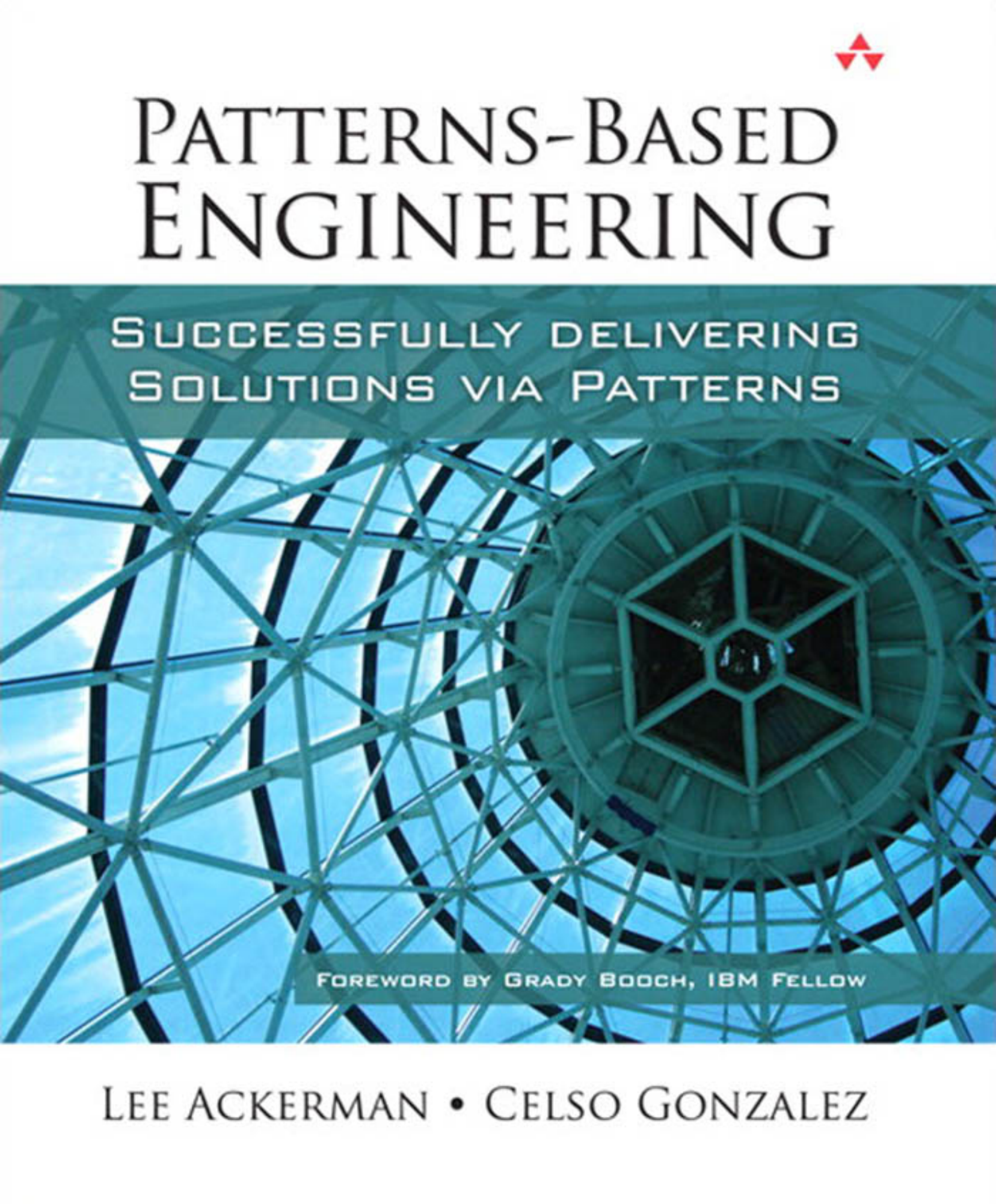
PATTERNS-BASED ENGINEERING
SUCCESSFULLY DELIVERING SOLUTIONS VIA PATTERNS
FOREWORD BY GRADY BOOCH, IBM FELLOW
LEE ACKERMAN • CELSO GONZALEZ

Figures 13.7 and A.3 are reproductions of Figure 29-1 from page 531 of Martin, R. C., D. Riehle, and F. Buschmann. 1998. "Context-Setting Patterns." In *Pattern Languages of Program Design 3*. Boston: Addison-Wesley. Reproduced by permission of Pearson Education.

Library of Congress Cataloging-in-Publication Data

Ackerman, Lee, 1971–
Patterns-based engineering : successfully delivering solutions via patterns / Lee Ackerman, Celso Gonzalez.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-57428-2 (hardcover : alk. paper) 1. Computer software—Development. 2. Software patterns. I. Gonzalez, Celso, 1969– II. Title.
QA76.76.D47A255, 2010
005.1—dc22

2010013630

ISBN-13: 978-0-321-57428-2
ISBN-10: 0-321-57428-1
Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing, June 2010

Foreword

Developing, deploying, operating, and evolving software-intensive systems are problems of engineering: One must devise a solution that reasonably balances the forces upon that system. Every individual system faces its own unique set of forces, and thus every system presents a unique engineering problem. Nonetheless, these are not all problems of singularity: Over time, common solutions to common problems emerge, and these become part of the institutional memory of that system space. Insofar as we can make those patterns manifest, we can improve the manner in which we develop, deploy, operate, and evolve systems. Indeed, this too is the nature of engineering: For every new system, we look back on things that didn't work (and try alternatives) as well as things that did (and improve them).

Lee and Celso have considerable industrial experience in delivering software-intensive systems, and in this book they bring the best practices they have learned to the problem of engineering software-intensive systems. If you are are unfamiliar with the nature of patterns, this book will help you understand how to discover, design, create, package, and consume these common solutions to common problems. Their extended case study demonstrates how to pragmatically apply these ideas; their guidelines offer patterns and antipatterns for engineering systems using patterns. Finally, Lee and Celso attend to the softer issues of Patterns-Based Engineering: its value, its risks, and its economic return.

As software-intensive systems continue to grow in complexity and in their importance to the world, it is our responsibility as software engineers to deliver systems of quality. Lee and Celso's work will help you along that path.

—Grady Booch
IBM Fellow
February 1, 2010

Preface

Increasing and unending pressure exists in software development to finish more quickly, to produce higher-quality solutions, and to do so with fewer resources. We can use patterns, proven best-practice solutions to known problems, as a powerful tool to help address these challenges. However, if the answer were as simple as "Just use patterns," we would already have dealt with these challenges.

There is complexity, depth, and nuance to using patterns, and succeeding with them requires knowledge, expertise, and guidance. And not only do we want to succeed with patterns, we want to do so in a fashion that is scalable, repeatable, and predictable. This book introduces an approach known as Patterns-Based Engineering (PBE) that provides guidance on how to successfully incorporate and leverage patterns in software development. We don't just use patterns; we think, evaluate, create, innovate, collaborate, abstract, simplify, justify, automate, and reuse.

Patterns-Based Engineering

PBE is a specialized approach to asset-based development that focuses on patterns, a specific type of reusable asset. PBE provides guidance and support for using patterns in a systematic, disciplined, and quantifiable way. With PBE an organization uses patterns in multiple forms, for numerous purposes, and in a number of ways. More specifically, we look at two specific types of patterns: pattern specifications and pattern implementations. We use these types of patterns to support design, testing, deployment, and other aspects of the software development lifecycle. In performing these tasks, we use patterns in many ways such as documenting, generating, refactoring, and harvesting. As a result, we are able to use patterns to boost productivity, improve quality, leverage expertise, simplify, and improve communication within an organization. The goal is to ensure that as we use and create patterns, we are doing so in a way that adds value and boosts the agility of our projects and organization.

An important aspect of PBE is that it goes beyond just the technology. Success on a project has never been and will never be just about the technology. We need to ensure that the team is able to work together; that we all know the roles we are to play, the tasks to be done, the work products to create along the way; and that we can communicate with one another.

How to Read This Book

This section starts with an overview of the book's structure. We then provide a guide to reading the book based on role, and we finish with suggestions for the background needed to get the most out of the book and a list of learning objectives.

Book Structure

This book is divided into four parts. Part I provides an introduction to PBE. Within this part, Chapter 1 starts by defining PBE and Chapter 2 follows by providing some examples of pattern implementations, as this is a new concept for most and is an important aspect of PBE. Chapters 3 through 7 then show an example of PBE in action through a case study. Chapter 8 concludes Part I with a discussion of the process aspects of PBE and how it could fit into existing software development processes, including coverage of Extreme Programming, Scrum, and OpenUP.

Part II describes some of the best practices related to PBE in the form of patterns and guidelines. Chapter 9 explains the organization and summarizes each of the patterns and guidelines. Chapters 10 through 16 detail the patterns and guidelines, each chapter focusing on a specific category.

Part III covers additional topics that provide a deeper examination of PBE, particularly the nontechnical aspects. We detail some of the PBE benefits in Chapter 17, move to the economic aspects of PBE in Chapter 18, and finish Part III with Chapter 19, which takes a look at some of the PBE misconceptions that may be faced in rolling out PBE within an organization.

Part IV wraps up the book with a set of appendices that provide supporting materials and references. Appendix A summarizes the main PBE definitions to provide quick access to some of the terms and concepts used throughout the book. Appendix B takes a look at PBE in comparison to other software development approaches. Appendix C provides a nonexhaustive list of tools available to help in applying PBE within an organization. Appendix D provides a set of PBE Patterns and Guidelines overview diagrams. Appendix E provides the pattern specification for the Subsystem Façade pattern created and used in the case study. Appendix F serves as a companion to Chapter 8, adding more details to support our understanding of the PBE Practice.

To get a high-level understanding of PBE, read all of Part I and Chapter 9 from Part II, which provides a high-level summary of the PBE Patterns and Guidelines.

For a deeper understanding of PBE you should go at least once through the PBE Patterns and Guidelines to get a better idea of what problems they address and the associated solutions. And if you are interested in some of the nontechnical aspects of PBE, Part III is a must read.

Who Should Read This Book

Patterns surface and are applicable throughout the development lifecycle and for multiple purposes. Thus, there is a wide audience for this book, including

- Software architects, designers, and developers: Read all parts.
- Project managers: Read Part I, at least Chapter 1, and Part III.
- Process engineers: Read all parts with a focus on Chapter 8 and Appendix F.
- Analysts, including those responsible for testing, requirements, and business: Read Part I, at least Chapter 1.

Suggested Background

To get the most from this book, we suggest that you have basic familiarity with the following topics:

- Object-oriented programming with a language such as Java or C#
- Patterns
- Unified Modeling Language (UML)
- XML

Learning Objectives

Upon completion of the book, you will be able to

- Describe ways in which patterns can be leveraged in delivering software
- Describe the roles, tasks, work products, and best practices defined within PBE
- Describe the factors that help decision making about investments driven by PBE as well as the expected implications of such decisions
- Describe the value and purpose of using patterns (implementations and specifications)
- Successfully identify, specify, and implement patterns
- Understand that patterns are for all roles and projects

Why This Book?

So, why this book? Why does the world need another patterns book? Aren't there enough already? One way to answer these questions is to say that we do not like solving the same problems over and over again. As a matter of fact, we struggle a bit dealing with the mechanical and mundane aspects of writing software. We are big fans of trying to be creative and solving new and unique problems. And we like to leverage automation to help us minimize and avoid having to work on those mechanical and mundane tasks.

Another way to answer these questions is that it's not enough to be passionate. It's not enough to be creative. It's not enough to use tooling and automation. There's already content in many forms—books, courses, and articles—that touches upon these topics. However, we were not able to find content that brings these ideas together. A holistic approach is needed to discuss how we can use patterns more strategically, more systematically. We need to be able to scale the use of patterns across the organization and ensure predictability and repeatability in our pattern-infused projects.

Downloadable Content

There are three downloads associated with this book, available from www.Patterns BasedEngineering.net. These downloads include

- Source artifacts for the pattern implementation that is discussed in the case study
- The source plug-in for a PBE Practice to be used with Eclipse Process Framework Composer
- A published configuration of the PBE Practice composed of a set of HTML pages that can be viewed with a standard web browser

Writing Style

Software development is a team sport; we present the rest of the book as a team effort. We are in this together, having a chat about how we can better leverage patterns in our projects.

Chapter 1

Defining Patterns-Based Engineering

Beginnings are tough. Where to start? Is the necessary expertise available? Will we finish on time? Will the quality be there? Will everyone on the project follow best practices? Will the team get a chance to be creative while using and improving their skills? Are the requirements really known and understood?

As software engineers, we find ourselves asking such questions whether we are working on a greenfield project or maintaining a legacy application. Often the answers are not to our liking. As we start the project, we often joke about missing deadlines and are anxious about whether we will get the job done. Experience has taught us that the road ahead is going to be difficult and frustrating. To quote Grady Booch: "Software development has been, is, and will remain fundamentally hard."[1]

We know that we are not alone. We need to improve how we deliver our software projects. We need to improve productivity, enhance quality, hasten time to market, have better governance, and do all of this while dealing with a challenging set of constraints,[2] such as not enough expertise, daunting timelines, ambiguous and changing requirements, and ever-increasing solution scope and complexity.

Over the years we have tried to take steps to address these issues. We've adopted Agile processes, as who wouldn't want their projects to be more agile? We've tried model-driven development (MDD); as they say, "A picture is worth a thousand words." We've incorporated the leading industry frameworks, including .NET and Java EE, as well as the frameworks within these domains that further support our efforts, including Spring, Hibernate, and JavaServer Faces (JSF). We've adopted the best approaches to development as they've emerged, such as object-oriented (OO), component-based development (CBD), and service-oriented architecture (SOA). We've outsourced and off-shored, looking outside our organization for support,

1. Booch (2007).

2. For a look at a selection of notable software failures, refer to Charette (2005).

skills, and cost management. However, we continue to come up short—all while the complexity of what we are asked to build continues to advance.

This book discusses Patterns-Based Engineering (PBE), an approach to software development. It is not the silver bullet; it is not the magic elixir that will cure all that ails our projects. However, PBE, as demonstrated in real-world projects, takes a systematic and disciplined approach to using patterns—proven, best-practice solutions—to deliver software. A key and unique aspect of this approach is that in addition to using existing patterns from the community, we identify and create patterns within the organization, codifying, automating, and leveraging our own best practices. Organizations that have adopted this practice have seen improved productivity, increased quality, better utilization of expertise, and improved governance.

Asset-Based Development

A good place to start in gaining an understanding of PBE is to look at asset-based development (ABD). There is a strong connection between PBE and ABD. ABD is focused on how to leverage investments made in software artifacts in future projects. However, the guidance related to ABD is typically focused on assets in general, which is useful when the focus is on promoting reuse across many types of artifacts. PBE builds on the foundation provided by ABD and provides guidance for how we can succeed with a specific type of asset—specifically, patterns. With this relationship in mind, let's take a more detailed look at ABD.

ABD includes four major areas—process, standards, tooling, and assets—all of which are focused on how to successfully reuse and benefit from assets. An asset is "a collection of artifacts that provides a solution to a problem. The asset has instructions on how it should be used and is reusable in one or more contexts, such as a development or a runtime context. The asset may also be extended and customized through variability points."[3] A variability point is a part of the asset that is purposely provided by the creator of the asset and allows for later configuration or extension of the asset. Variability points are key to success with ABD, and in turn PBE, as they allow us to take a proven solution and easily tailor, customize, and adapt it to the specifics of our situation.

Generally a team produces numerous different types of artifacts as they look to deliver software solutions, ranging from requirements, to models, code, tests, and even deployment scripts. Each of these investments could potentially become a reusable asset. We need to evaluate specific instantiations of these artifacts to determine which would warrant an investment.

3. Larsen (2006).

As shown in Figure 1.1,[4] when following an ABD approach, we look at four areas of effort related to the assets, including

- **Asset identification.** We need to identify potential assets and determine which are suitable for investment.
- **Asset production.** After we have identified candidate assets, we need to produce those assets.
- **Asset management.** As assets are produced, we take appropriate steps to manage them and make them available for others to reuse. This includes support for searching, reviewing, and providing feedback. An asset repository is typically used to assist with this effort.
- **Asset consumption.** Once a set of assets is made available, the team accesses the asset repository and reuses the assets in their projects. Users of the assets are expected to provide feedback to the asset producers. This feedback is used to improve the assets and increase their value to the organization.

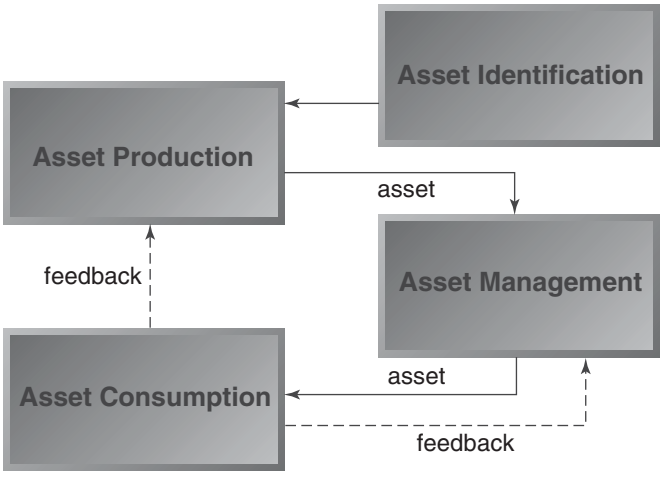

Figure 1.1 *Overview of major areas of effort related to asset-based development*
Credit: Grant J. Larsen, IBM.

Typically, we consider ABD from two perspectives. In one perspective we are concerned with the available tools, processes, standards, and assets. In the other perspective

4. Larsen (2003).

we focus on the efforts that we put into the identification, production, management, and consumption of assets.

If an organization already has an ABD program in place, PBE is a quick addition to the effort. If an organization has not yet adopted ABD, PBE is a very good place to start learning, adopting, and succeeding with assets.

Patterns

Let's take a look at what a pattern is and why it is important. A simple definition that we can start with is this: A **pattern** is a proven best-practice solution to a known, recurring problem within a given context. This definition still leaves a bit of ambiguity about patterns. There are many best-practice solutions out there that apply to a context but may not be considered to be patterns. To help further refine and expand our understanding of what a pattern is, we can refer to some of the work done by Christopher Alexander. The patterns movement started with his book *The Timeless Way of Building*. In the book Alexander looks at how we can build better architectures. Although the book addresses civil architecture and not software architecture, its ideas and guidance can be adapted and applied to software. In a subsequent book, titled *A Pattern Language*, Alexander states that a pattern

> . . . describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.[5]

At this point we are starting to get a more precise definition. A key idea that surfaces from Alexander's statement is that a pattern can be used many times. However, when applying it, we can adapt it so that it suits the needs of a particular situation. Although we may use the pattern multiple times, each instance of the pattern is unique.

Building on this definition, we can add a few ideas provided by John Vlissides.[6] He points out that a pattern needs to support teaching and have a name. In this way we can refer to the entire pattern by just its name and still convey meaning. The detail provided with the pattern can also be used to teach the pattern user about the best practices associated with the pattern, when to use the pattern, and the implications of doing so.

In the software world, the most-referenced book on patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* written by Erich Gamma, Richard

5. Alexander (1977).

6. Vlissides (1998).

Helm, Ralph Johnson, and John Vlissides—known as the Gang of Four (GoF).[7] This book contains 23 design patterns that are widely used and referenced, including patterns such as Abstract Factory, Bridge, and Observer. These patterns and others are often embedded within the frameworks that we depend upon for building our solutions, like the Model-View-Controller pattern, which is the basis of Struts and JSF. Odds are that we've already been using patterns in our projects.

Engineering

As we build a definition of PBE, let's next define engineering. **Engineering** is "the application of a systematic, disciplined, quantifiable approach to structures, machines, products, systems, or processes."[8]

As we are dealing with software, it makes sense to look at a definition that is a little more targeted to the work we do. So **software engineering** is defined as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."[9]

With definitions of patterns and engineering as they relate to software development in mind, we are now in a position to consider a definition for Patterns-Based Engineering.

Patterns-Based Engineering

Let's start with a short definition. PBE is

> a systematic, disciplined, and quantifiable approach to software development that leverages the use of pattern specifications and implementations throughout the software development and delivery process.[10]

7. Gamma et al. (1995).

8. IEEE Standard Glossary of Software Engineering Terminology, www2.computer.org/portal/web/seonline/glossary.

9. Ibid.

10. This definition reflects this book's focus on the use of patterns for building software solutions. However, we could replace the term *software development* in this definition with *enterprise architecture* or *operational modeling*, for example. The use of patterns is applicable across many aspects of the IT domain.

When we dig into this definition, a number of key ideas surface:

- PBE augments the overall software development and delivery process. We take a wide view of how patterns support us in developing and delivering software.
- PBE unifies the use of patterns in their different forms, using pattern specifications as blueprints as well as using pattern implementations to automate the application of those blueprints.
- PBE focuses on the systematic, disciplined use of patterns while enabling us to quantify the impact of using the patterns.

Pattern Specifications and Pattern Implementations

Design patterns from the GoF book were the first patterns to gain significant attention. Since that time, many other patterns have been identified and documented, as evidenced by the large number of results from online pattern searches. Generally, these patterns are presented in formal, written documentation that explains the pattern. We refer to these documents as **pattern specifications**.

There is some variation in how pattern specifications appear, but the following information is usually included:

- The *name* of the pattern
- A description of the *problem* the pattern solves
- A description of the *solution* that the pattern provides
- A discussion of the *consequences*, the advantages and disadvantages, of applying the pattern

Pattern specifications provide a great deal of value, as they

- Support learning about the best-practice-based approach to a recurring problem
- Simplify communication, since the name of the pattern can be used in place of repeating all of the pattern details
- Enable people to easily read and learn about a pattern
- Detail a best-practice approach to solving a recurring problem

However, there are a number of limitations to just using pattern specifications:

- Pattern Users need to be aware of what patterns exist and how they can be applied.

- Because patterns are tailored to the context in which they are used, it is highly likely that each person who uses a pattern will create slightly different solutions.
- If a pattern needs to be reapplied to a solution, it is a manual effort to update all the areas of the solution that leverage the pattern.
- Because the pattern is applied manually, human error is likely to creep into the application of the pattern. Even a small percentage of errors become significant over a large number of applications.
- It is difficult for individuals and teams to use a selection of patterns together. All the complexity of the combination surfaces as we are unable to easily encapsulate and shield Pattern Users from such complexity.

With the limitations of pattern specifications in mind, we are left to wonder how tooling and automation could assist. To that end, we look to the idea of **pattern implementations**. A pattern implementation automates the application of a pattern in a particular environment. Thus, patterns become tools themselves, concrete artifacts within the development environment.

From a Pattern User perspective, there are a number of different ways in which a pattern implementation can be manifested. A pattern implementation may surface as a wizard, a model transformation, a UML pattern, a web page, or even something as simple as a right-click with the mouse.

Benefits of using pattern implementations include

- **Increased productivity.** Using pattern implementations simplifies and accelerates how we deliver software. We are able to automate our best practices, which allows us to dramatically reduce repetitive and manual efforts. In addition, we reduce the skill requirements for working with and applying a pattern correctly.
- **Increased quality.** Pattern implementations allow us to consistently create solutions that adhere to architectural, design, and coding standards. In addition, as the pattern implementation embodies our best-practice approach to solving a problem, we are by definition increasing the quality of our solutions.
- **Better leveraging of skills and expertise.** With a pattern implementation we are able to capture our best practices and then make them available to the rest of the team for reuse. Others easily reuse the expertise that goes into the pattern, without the need for the experience, trial and error, and research that went into creating the pattern.
- **Improved governance.** Not only are we able to use tooling to apply our best practices, but we can also check that the resulting solution adheres to these best practices.

- **Reduced cost.** We can reduce the cost of the solutions we build as we are able to be more productive, increase quality, better leverage skills, and improve the governance associated with best practices.

A Model for Succeeding with PBE

With this background in mind, we can state that a pattern is a specific type of asset, and we can state that PBE is a specialized form of ABD. The main difference is that we focus on a specific type of asset, namely, a pattern. We still look at and work with all of the other types of artifacts that could be assets, such as requirements, models, code, tests, deployment scripts, and so on. Within PBE, these other types of assets either are used in association with patterns or are used as input into our efforts to create new patterns.

However, we still need to find answers to questions such as these:

- How do we perform PBE? That is, how can we take a systematic, disciplined, and quantifiable approach to using patterns to develop and deliver software?
- How do we succeed in taking on a PBE approach and improving how we deliver software?
- What are the best practices associated with PBE?
- What are the roles and tasks associated with PBE?
- How can we adopt and succeed with PBE as a team?

As we work our way through the rest of the book, we will discuss answers to these questions. As is the case with ABD, we can consider PBE from two perspectives. First, we are concerned with the available tools, processes, standards, and assets. Second, we look at the effort that we put into the identification, production, management, and consumption of patterns.

We can also leverage a model, as shown in Figure 1.2, to help us understand and position the content found in the rest of the book. The elements in the model build out from the base, leveraging the elements contained within. Starting with the innermost circle, we can see that there is a set of PBE Core Values. These core values form the basis of how we approach PBE. The goal is to ensure that we are able to quickly understand, remember, and relate a small and simple set of values that will influence all of our PBE efforts.

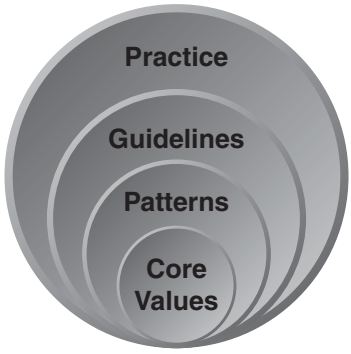

Figure 1.2 *A model bringing together the key elements that support our PBE efforts*

There is then a set of PBE Patterns that build upon the PBE Core Values. These patterns, as expected, provide a set of proven best-practice solutions to recurring PBE problems. There are patterns that support us in identifying, producing, managing, and consuming patterns.

Beyond the PBE Patterns there are PBE Guidelines to further assist us in performing PBE. The PBE Guidelines provide advice on PBE, including how to use the patterns and core values.

The final element shown in Figure 1.2 is the PBE Practice. In general, a practice is a process component; that is, it is a building block to help in building out a software development process. Typically a software development process is composed of a number of process components. Some of the components are focused on testing, others on deployment; in this case we will look at a process component focused on PBE. The PBE Practice looks at the tasks, work products, roles, artifacts, and associated guidance, patterns, and core values that we can use in successful PBE efforts.

The following sections take a closer look at each of the constructs from Figure 1.2.

PBE Core Values

With an overall model of PBE in place, we can now start to take a more in-depth look at each of the components within that model. We start with the PBE Core Values, as they serve as the basis for the other elements. These are the PBE Core Values:

1. **Patterns are best used in combination, rather than in isolation.** When building a solution, we expect to use many patterns. The patterns selected will vary in size and occur at multiple levels of abstraction, so we expect that the patterns will both connect and overlap. As stated by Christopher Alexander: "But it is also possible to put patterns together in such a way that many patterns overlap in the

same physical space: the building is very dense; it has many meanings captured in a small space; and through this density, it becomes profound."[11]

2. **Always identify and build new patterns.** We need to always be on the lookout for potential pattern opportunities across repetitive scenarios, repetitive code, repetitive solutions, and areas where we are just mechanically participating in the development effort.

3. **Patterns can be built and used within the same project.** A challenge that has traditionally surfaced in building reusable assets is determining when we should harvest them. Harvesting, whereby we identify and then extract reusable assets from existing solutions, is a significant effort and expense. We thus look for ways to justify the time and monetary expenditure for the asset harvesting. Often we decide that we should just wait until the end of the project and then harvest the assets for reuse on a later project. With PBE we can identify and build patterns within the current project. The assets thus pay for themselves during the current project and are then also available for other projects to use.

4. **Make your patterns live.** A good place to start with this principle is with a quote from Christopher Alexander: "You see then that the patterns are very much alive and evolving. In fact, if you like, each pattern may be looked upon as a hypothesis like one of the hypotheses of science."[12] Much like our development efforts, when we build patterns we leverage an iterative and incremental approach with a focus on always delivering value. In addition to building a better pattern over time, this approach also allows us the opportunity to look at ways in which we can increase the scope of the pattern. This also reduces pressure—we do not need to produce the perfect pattern on the first attempt. We are also able to collect feedback from the Pattern Users, leading to enhancements in future releases. As a result, the portfolio of patterns will grow and mature over time.

5. **Focus on making patterns consumable.** All the effort of identifying and building patterns is pointless if Pattern Users are unable to work with the patterns. Pattern consumability touches upon many aspects such as ease of use, enablement materials, and the ability to find the right pattern at the right time.

6. **PBE can fit into many different development processes.** PBE itself is not a process. It is a development practice that can be combined with and leveraged by other practices and processes. The PBE Core Values, Patterns, Guidelines, and Practice can be leveraged within most other modern software development processes.

---

11. Alexander (1977).

12. Ibid.

PBE Patterns and Guidelines

The PBE Patterns and Guidelines support us in identifying, producing, consuming, and managing patterns. The patterns and guidelines support one another. The guidelines help us to succeed with the patterns, and the patterns in turn help with the guidelines. Why patterns and guidelines? PBE has a set of patterns that have surfaced over the years. It makes sense to discuss patterns that can help us follow PBE; think of them as metapatterns. Also, it is important to see that new patterns are discovered and created; we are not restricted to using patterns that have been discovered by others. Patterns are for everyone; we all need to be on the lookout for pattern opportunities.

Guidelines are also needed to provide advice on how to successfully apply PBE. Not everything needs to be or should be a pattern. We need to evaluate and review patterns to ensure that they are worthy of the name and add value. We don't want to get into a situation where everything is a pattern (think Maslow's Hammer[13]) and end up diminishing the value of the term and concept.

Additional details for each of the patterns and guidelines are provided in Part II. More specifically, Chapter 9 provides an overview of the entire set of patterns and guidelines. The following chapters then provide details on each of the patterns and guidelines based on categories, including foundational patterns, pattern discovery and identification, designing patterns, creating patterns, pattern packaging, using domain-specific languages (DSLs) and patterns, and consuming patterns.

PBE Practice

Typically, a software development process provides guidance regarding the roles, tasks, work products, and workflow needed to develop software. However, PBE is not a full-fledged process; it is a practice. In essence, a practice is a process component that is used in conjunction with other process components (practices) to create a process. The practice still looks at the roles, tasks, work products, and workflows needed; however, the focus is entirely on PBE and not all of the other things that you would normally do when developing software. If we view the software development process as a set of components—some on testing, others on writing code, and some on source code management—we will focus on the PBE component. To that end we will look at the roles, tasks, work products, and workflows associated with PBE.

13. The idea that if all you have is a hammer then everything looks like a nail is referred to as Maslow's Maxim or Maslow's Hammer. More information regarding Abraham Maslow and Maslow's Hammer is available at www.abraham-maslow.com/m_motivation/Maslows_Hammer.asp.

The PBE Practice is available in source form that can be modified, configured, and integrated with other practices. Figure 1.3 provides a view of a published default configuration of the PBE Practice. This published configuration is a set of interconnected HTML pages that can be viewed through a standard web browser. As seen in the figure, we are able to use the navigation tree on the left-hand side of the screen to quickly access information about the concepts, roles, tasks, work products, tool list, checklists, and templates that the practice provides. Chapter 8, "PBE and the Software Development Process," provides more details on the elements in the PBE Practice and how to integrate this practice with a software development process.

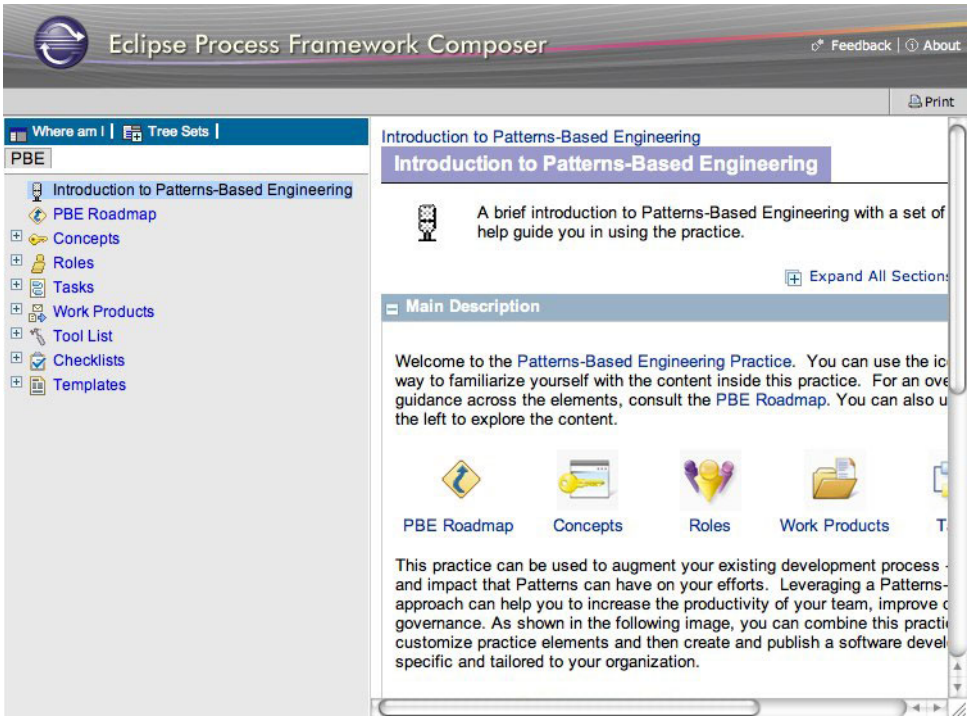

Figure 1.3 *A view of the published PBE Practice from within a standard web browser*

Examples of PBE Results

Chapters 3 through 7 present a case study in which a fictional organization leverages PBE in delivering a software solution. The case study is valuable because it pulls

together a range of real-world experiences, ideas, and guidance within the context of one example. However, it has the disadvantage of being a fictional example.

This section provides a brief discussion of real-world examples where teams and organizations have used PBE Core Values, Patterns, Guidelines, and aspects of the Practice in delivering solutions. Each of the examples comes from real-life projects; however, the names of the companies have been left out.

Services Team: Portlet Proof of Concepts

In the first example, a software vendor's services team needed to build custom portlets to support proof-of-concept engagements. Before they used patterns, the work was performed manually, and it would take about 40 hours to create and test a custom portlet. In a proof-of-concept environment, this is unacceptable as it significantly increases the cost and time to complete the proof of concept. This was identified as an opportunity to build a pattern as there was an existing and recognized best practice that was used many times. The pattern was responsible for creating a large number of files (approximately 95) needed for the portlet, code was added to handle common configurations and errors, and the pattern would also generate optimized and tested code. After the pattern was applied, the Pattern User would add business logic to the portlet. Using the pattern to build the portlet took approximately 20 minutes, and the majority of that time was spent on adding the business logic and then deploying the portlet. The pattern was used over a number of years and resulted in over 80 portlets being generated, saving over 3,000 hours of development time.

Software Vendor: Product Update

The second example is based on a team working for a software vendor. The application that they worked on needed to meet a hard deadline for a customer and ship with a set of 130 SOA-based services that adhered to the OAGIS message format. A significant number of steps were needed to create each of the services, resulting in over 100 pages of documentation. The team discovered that performing this work manually would push them three months beyond the customer-imposed deadline.

The team spent a day learning how to create pattern implementations and then went to work applying those skills. The resulting pattern generated a significant portion of the required code, including data objects, client code, server code, and even unit tests. When using the patterns, the team was able to reduce the documentation from over 100 pages to 2 pages. They were able to save over 1,400 hours in their project and deliver the solution to the customer in time to meet the deadline. They also ended up shipping the patterns with the product, enabling customers and business partners to benefit from the patterns.

Entertainment Industry: Enhancing MDD

Let's take a look at a couple of additional examples that go beyond companies that sell and service software.[14] First we'll look at a company in the entertainment industry. In this case the organization was following an MDD approach. More specifically, the team used a UML modeling tool for creating models that would then drive their development. The models were used for documentation, and a manual effort was required to follow through and create the solution. The organization had concerns about productivity, quality, and governance. Some patterns were used, but only manually, and they needed interpretation in the transition to development. Development phases were too long, and transcription errors occurred when moving from the models to code. In addition, adherence to corporate architecture standards was inconsistent, and best practices were not always followed.

The organization performed an analysis of its development approach, models, and resulting artifacts. As a result of this analysis a set of patterns was identified and then delivered as pattern implementations. By using these patterns, it was estimated that the architects in the organization were able to achieve up to 50% improvement in productivity, resulting in millions in development savings. In addition, they were able to eliminate many defects from the resulting solution by using automation in place of manual efforts.

Government: Integrating Departments

The last example we'll look at in this section is based on work completed for a European government department. The department was required to externalize its data via SOA-based services so that it could be shared and integrated across the government. Within the department there were multiple divisions, and each was expected to meet this requirement. The planned strategy was to use an enterprise service bus (ESB) to assist in externalizing the services and support integration and sharing. In creating a solution, the department required a development platform that all of its employees could use that would be simple, agile, and highly productive. In addition, they required a strong level of governance and high quality; each member of the team had to adhere to the corporate architectural standards.

During the analysis of the situation it was determined that supporting the integrations would take three days of effort from a highly skilled resource for each of the services. There were a significant number of services to create. Each of the services would be created according to a specific best-practice-based approach. Recognizing a

14. For another example from the software vendor realm, we encourage you to read Siddle and Draper (2008).

recurring best-practice approach, they leveraged the work done in the analysis and recognized the opportunity to create a set of patterns to generate the required integrations. Using pattern implementations, rather than performing the work manually, allowed each of the services to be completed in approximately ten minutes and required a much lower skill level. The organization expected to save over 50 person-years of effort by the switch to using pattern implementations.

Why Do We Need PBE?

As with most types of reusable assets, the more generic the asset, the lower the return that each reuse will provide. Many books since the original GoF book have provided guidance on these first 23 patterns. These books have mapped the patterns to numerous languages, detailed the designs of the patterns, and used the patterns to detail how to perform object-oriented development. In addition, many more patterns have been created and shared. A search of the web or a bookstore will lead you to numerous results that list patterns and pattern-related documents. For instance, the *Handbook of Software Architecture* website[15] has cataloged 2,000 patterns so far. So if we have all of these patterns available to us, you might leap to the conclusion that our problems in delivering software have been alleviated. Unfortunately, that is not the situation; the answer is not as simple as just saying, "Use patterns."

As patterns have been available for use in software development for quite some time now, one would expect that there would be a very high adoption rate and that we would have already reached a very high level of maturity in using them successfully. Unfortunately, we have run into a number of issues in using patterns. In practice, we see that

- There is little methodology that actually shows how to use and leverage multiple patterns within a solution. This leads to random and nonstrategic use of patterns.
- Most of the patterns that people use are the GoF patterns.
- In cases where other patterns are used, they are often used in isolation rather than being woven together as part of a larger solution.
- There is little skill in identifying and formalizing patterns that are unique within an organization. In addition, there is little focus on such patterns.
- If patterns are used, they are used only for forward-engineering a solution.

15. www.handbookofsoftwarearchitecture.com.

- Abstraction is put to limited use.
- There is concern that restricting the creativity of the developers within the organization will diminish job satisfaction.
- The patterns used are often invented elsewhere.[16]

Would the application of a systematic, disciplined, and quantifiable approach to pattern use solve some of the issues that have limited their success to date?

The Importance of Creativity, Constraints, Rules, and Assumptions

Popular business literature stresses the importance of creativity, and we expect and value it in new team members. However, unmanaged and unfocused creativity can be detrimental to a project. We need to channel and focus the creativity of the team; we want creativity with purpose, creativity that helps us to reach our goals.

We use platform decisions, architectural styles, architectural patterns, and so forth as mechanisms for narrowing the solution space in a collaboratively disciplined way. To support both the production and consumption of patterns, we seek to leverage constraints, rules, and assumptions. To ensure that we are all on the same page, let's take a quick look at the definitions associated with these terms:

- **Creativity:** "Characterized by originality and expressiveness; imaginative"[17]
- **Constraint:** "The state of being restricted or confined within prescribed bounds"[18]
- **Rule:** "A principle or regulation governing conduct, action, procedure, arrangement, etc."[19]
- **Assumption:** "A statement that is assumed to be true and from which a conclusion can be drawn"[20]

16. For additional discussion of some of the challenges regarding the use of patterns, see Manolescu et al. (2007).
17. *American Heritage Dictionary of the English Language, Fourth Edition*. Retrieved July 1, 2008, from http://dictionary.reference.com/browse/creativity.
18. *American Heritage Dictionary of the English Language, Fourth Edition*. Retrieved June 24, 2008, from http://dictionary.reference.com/browse/constraint.
19. *Dictionary.com Unabridged (v 1.1)*. Retrieved June 24, 2008, from http://dictionary.reference.com/browse/rule.
20. *WordNet 3.0*. Retrieved June 24, 2008, from http://dictionary.reference.com/browse/assumption.

In addition to using patterns to focus the creativity of the team, we also want to support the consumption of the patterns. Where's the value in cases where a Pattern User is unable to use a pattern? To help support consumption we provide constraints and rules. Constraints and rules ensure that the pattern is used correctly. In the case of a pattern implementation, the assumptions that are found within the pattern help to reduce the amount of information the user of the pattern needs to provide.

When we turn our focus to the building of patterns, we similarly need to look for ways to manage our creativity. We can use patterns to guide us in identifying, documenting, and building patterns. In addition, we need to leverage the creativity of the team to identify, document, and build the patterns that our organization needs.

Important Definitions

The terms defined in this section, as shown in Figure 1.4, are important to PBE and will be referred to in the rest of the book. This section provides brief definitions of these terms to ensure a common understanding. Additional details on these terms and supporting definitions are provided in Appendix A, "PBE Definitions."

Figure 1.4 also shows that there is a relationship between these elements. We use metamodels, DSLs, and patterns to represent solutions within a model. Models that are recognized as representing a best-practice solution are exemplars. With an exemplar in hand, we are able to create a new pattern that can be used to specify future solutions. With these relationships in mind, let's take a look at the definitions for each of these terms.

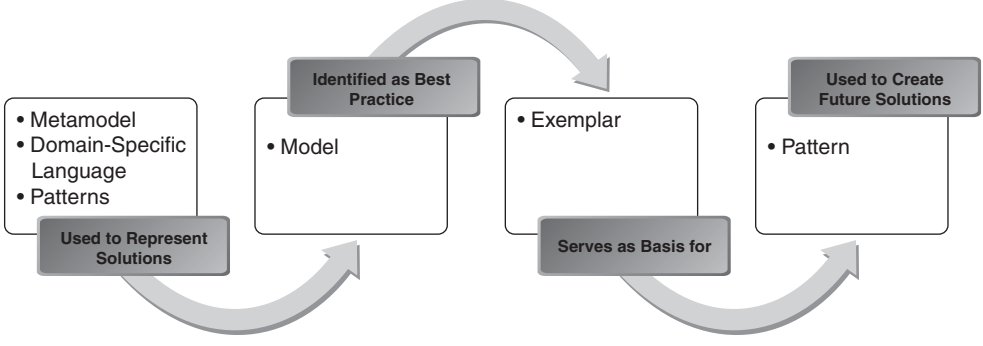

Figure 1.4 *Overview of key concepts related to PBE*

Model

Key to working with patterns is the use of models. Simply put, a **model** is an abstraction or a simplification of reality. A good model includes elements that are relevant at a given level of abstraction, while hiding or ignoring details that are not relevant. A model may be structural, emphasizing the organization of a solution, or behavioral, emphasizing the dynamics of a solution.

Metamodel

The language used within a model can be formalized textual, formalized graphical, or natural language. A formalized language—one that has a detailed and precise description—is helpful in creating models that support communication between people as well as communication with machines. A metamodel is a key mechanism we use in building a formal language. A **metamodel** is a special type of model that describes and specifies a modeling language. Essentially, we use a more abstract model to define the language that is used in another, more concrete model.

Exemplar

When we want to create a new pattern (whether a specification or an implementation), we need to keep in mind that patterns are discovered rather than invented. To this end, we often look for representative reference solutions that we can analyze and use as the basis for the pattern. We call such a reference solution an **exemplar**.

To identify a possible pattern, we are on the lookout for both exemplars and situations where the "Rule of Three" applies. The Rule of Three is used to judge where a possible pattern may exist; in this case we are looking for situations where the same problem/solution set has occurred in three unique situations.

As noted at the beginning of this chapter, starting a project is full of difficulties, including concerns about available expertise. However, what if we take the idea of exemplars and the Rule of Three into consideration? Can we use these ideas to help us find patterns to use on the project? Some of the sources to which we could apply these ideas include past project designs and implementations, as well as some of the artifacts of the current project. For instance, with the current project, we could examine the architectural mechanisms, key use cases, recurring aspects of the solution, and so on.

Domain-Specific Languages

A **domain-specific language** (**DSL**) is "a programming language or specification language dedicated to a particular problem domain, a particular problem representation

technique, and/or a particular solution technique."[21] More simply put, a DSL is a language that we can use to describe a solution that allows us to use terminology from the domain in which we are working.

There are a number of reasons to use DSLs along with PBE, some of which are these:

- We are trying to simplify the lives of pattern consumers and enable them to structure the input model to the pattern in the simplest manner possible.
- Ideally we are able to automate the use of the patterns via the creation and use of pattern implementations.
- DSLs and their underlying metamodels support communication between people as well as with machines.
- A DSL enables us to both speak in terms of the problem domain and to operate at higher levels of abstraction.

Summary

If there is one key idea that you should take away from this book, it is that patterns are for everyone. Whether you are working as an architect, developer, tester, or business analyst, you should be looking at what patterns can be reused and what opportunities present themselves for new patterns. You need to be aware of the best-practice-based solutions and repeating situations where you can apply these best practices. Don't worry about finding the perfect pattern that will work for everyone in all situations. As stated by Alexander:

> You see then that the patterns are very much alive and evolving. In fact, if you like, each pattern may be looked upon as a hypothesis like one of the hypotheses of science.
>
> In this sense, each pattern represents our current best guess as to what arrangement . . . will work to solve the problem presented.[22]

Patterns represent the best thinking currently available. Patterns are alive; they do not start out as perfect entities but improve in quality over time as the community of software engineers investigates and refines them.

21. http://en.wikipedia.org/wiki/Domain-specific_language.

22. Alexander (1977).

As you start your journey with PBE, you can leverage tools, processes, standards, and patterns, with a focus on how to identify, produce, manage, and consume patterns. Leveraging the PBE Core Values, Patterns, Guidelines, and associated Practice provides a set of materials you can use to guide you in adopting and succeeding with PBE.

If you already have started an ABD program, PBE makes a logical and easy addition to your efforts. If you have not yet started with ABD, PBE is a great initiative to use to get things going.

Index

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X