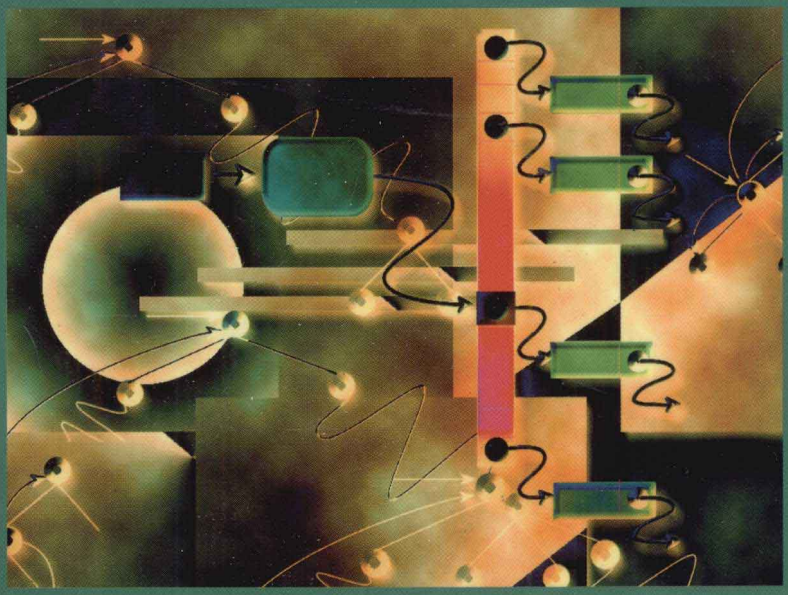
SEI SERIES IN SOFTWARE ENGINEERING

# Building Systems from Commercial Components



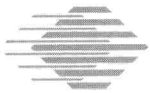
Kurt C. Wallnau
Scott A. Hissam
Robert C. Seacord

# Building Systems from Commercial Components

Kurt C. Wallnau
Scott A. Hissam
Robert C. Seacord

Addison-Wesley

Boston • San Francisco • New York • Toronto
London • Munich • Paris • Madrid
Capetown • Sidney • Tokyo • Singapore • Mexico City



**Carnegie Mellon**
**Software Engineering Institute**

The SEI Series in Software Engineering

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

CMM, Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, and CERT Coordination Center are registered in the U.S. Patent and Trademark Office.

ATAM; Architecture Tradeoff Analysis Method; CMMI; CMM Integration; CURE; IDEAL; Interim Profile; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Personal Software Process; PSP; SCAMPI; SCAMPI Lead Assessor; SCE; Team Software Process; and TSP are service marks of Carnegie Mellon University.

ANY MATERIAL FURNISHED BY CARNEGIE MELLON UNIVERSITY AND THE SOFTWARE ENGINEERING INSTITUTE IS FURNISHED ON AN "AS IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

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.

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

Pearson Education Corporate Sales Division
One Lake Street
Upper Saddle River, NJ 07458
(800) 382-3419
corpsales@pearsontechgroup.com

Visit AW on the Web: www.awl.com/cseng/

*Library of Congress Cataloging-in-Publication Data*

Wallnau, Kurt C.
Building systems from commercial components / Kurt C. Wallnau, Scott A. Hissam, Robert C. Seacord.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-70064-6
1. System design. 2. Component software. I. Hissam, Scott. II. Seacord, Robert C. III. Title.

QA76.9.S88 W345 2002
005.1'2dc21

2001022846

Copyright © 2002 by Addison-Wesley

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

0-201-70064-6
Text printed on recycled paper
1 2 3 4 5 6 7 8 9 10—EB—0504030201
First printing, July 2001

# Preface

There is a real *and growing* gap between the theory and practice of component-based software design.

There are, of course, books on component-based design. However, these books assume that the design task is to develop specifications for software components when most component-based design relies on *preexisting* components. There is room for both perspectives. However, preexisting components introduce new and novel design challenges, and their use is becoming increasingly prominent. Pre-existing components mean preexisting component specifications, and these are constraints on—not artifacts of—a design.

Current component-based design methods are focused on the *less interesting* and *less encountered* design problem. The more common and more interesting aspects of the design process are those that are no longer under the control of *the designer.*

- Use of preexisting components involves a completely different class of design problem than arises from component specification. Preexisting components involve the designer in *selection decisions*, while the freedom to define component interfaces involves the designer in *optimization decisions*. The difference between these classes of design problem are only gradually becoming evident to software engineers, and design methods have not yet caught up with this growing awareness.
- Use of preexisting components involves a significant loss of control over fundamental design decisions: how a system is partitioned into components, what functionality is provided by components, and how components coordinate their activities. In software engineering theory, these are architectural (that is, design) decisions. This leads to the mistaken conclusion that aggressive use of preexisting components is antithetical to, or at least incompatible or disjunctive with, software design.

We have described briefly the state of component-based design methods today, but have not yet supported the assertion that there is a growing gap between the theory and practice of component-based development. In fact, the gap does exist and is self-evident, once you know where to look for it.

The trend toward component-based development has been well under way for more than fifteen years, and has its roots in the commercial software marketplace. Software products, such as relational database management systems, transaction monitors, message brokers, event managers, encryption services, Web

browsers and servers, geographic information systems, product data management systems, *ad infinitum*, all satisfy the essential criteria of software component, at least as this term is coming to be understood by industry. That is, they all are implementations of functionality, are in binary form, are independently deployed, are described by a programmatic interface, and support third-party integration.

The commercial marketplace is the primary source of software components. This is true today, and will remain so for the indefinite future. Indeed, we believe that components and the software component marketplace are inextricably linked. Szyperski, in his influential book, shares this belief by observing that a component must be defined to fill a market niche [Szyperski 98]. However, Szyperski's notion of market was largely (although not completely) metaphorical. In contrast, our use of the term *component market* refers to something that demonstrably exists today, complete with component suppliers, component infrastructure providers, third-party component integrators, and, ultimately, consumers.

Ignoring the effects of the marketplace on software engineering would be analogous to ignoring the effects of friction on mechanical engineering. In particular, there are three qualities of commercial software components that together account for a significant share of the challenges posed by software components.

1. Commercial software components are *complex*. This complexity is needed to justify and sustain a component market. Many components are sufficiently complex that even experts in their use do not know all their features. There are invariably unknowns about component features and behavior.
2. Commercial software components are *idiosyncratic*. Standards are useful, but innovative features attract consumers. This means component knowledge is vendor-specific, and integration difficulties arise due to mismatches among innovative (that is, nonstandard) features.
3. Commercial software components are *unstable*. New features must be introduced to motivate upgrade, and are needed where competitors have copied successful features. Component knowledge has a short half-life, and design assumptions based on component features are fragile.

These qualities of software components, as they are found in the practice of building real systems, confound the assumptions of an orderly process that underlie traditional software design methods. However, these new complexities require a methodological response, since all component-based roads lead to the commercial component marketplace.

## Methodological Response

A central proposition of our approach is that a principal source of risk in component-based design is a lack of knowledge about how components should be integrated, and how they behave when integrated. To mitigate this risk, component-based

此为试读,需要完整PDF请访问: www.ertongbook.com

design inherently involves exploration and discovery. Acquiring and sustaining technology (component) competence is a principal motivation for this exploration.

This proposition may appear to some to be a heretical departure from the canons of software process improvement, which emphasize management skills over technical skills, and collective behavior over individual contributions. Indeed, phrases such as "that's just plumbing" in reference to component integration details, and "we need to get beyond individual heroics" in reference to reliance on software engineers with extraordinarily deep technology competence, are indicative of a mismatch between *perceptions* of what is important in software process, and the *reality* of what is needed in component-based development. In fact, the feasibility of a design is often dependent on "plumbing." Moreover, the overall design conception often depends on these low-level details. And there is no escaping the fact that deep technology competence is essential if these details are to be mastered.

The following are core elements of our methodological response:

1. We introduce component *ensemble* as a fundamental design abstraction. Ensembles expose component dependencies, and shift the emphasis from selecting individual components to selecting sets of components that work together (that is, ensembles).
2. We introduce *blackboards* as a fundamental design notation. Blackboards depict what is currently known about an ensemble and, just as important, what remains to be discovered. Blackboards serve to document a design and known areas of design risk.
3. We introduce a risk-driven discovery process, called $R^3$, for exposing design risk, and for defining ensemble feasibility criteria. We also introduce a prototyping process, called *model problems*, for generating situated component expertise, and for establishing ensemble feasibility.
4. We introduce the *design space*, defined in terms of ensemble relations and predicates. The design space captures dependencies among ensembles that arise in response to anticipated market events such as new component releases, and design hedges where ensemble feasibility is in doubt.

The methodological challenge is to meet the challenge posed by the commercial component market without allowing a) the design process to degenerate into an exercise in hacking, and b) innovative but unstable technology features to dominate a design and result in excessive and unnecessary design risk. The approach we prescribe, we believe, meets this challenge.

## About This Book

### GOALS OF THIS BOOK

Our goals are straightforward. Our first goal is to show that software components pose new methodological challenges for software engineering. In making this argument, we hope to clarify the nature of these challenges, with particular emphasis on those challenges rooted in the dynamics of the component market. Our second goal is to describe, in detail, processes and techniques that respond to these challenges. We believe these processes and techniques are a necessary foundation for any methodological response to software components. Our final goal is to illustrate, in a realistic case study drawn from our own experience in developing a large enterprise system, the complexity of component-based design, and the efficacy of our proposed processes and techniques.

### INTENDED AUDIENCE

This book is intended for individuals participating in a component-based development effort, and for students of software engineering. Although the whole of the book provides useful information for all of these roles, emphasis may vary.

**System Architect.** The lead designer will find ensembles, and the techniques for reasoning about ensemble repair and feasibility, welcome additions to his or her repertoire. The design space provides the system architect the conceptual language for managing the many layers of contingency and repair that characterize complex component-based systems.

**Chief Engineer.** While the system architect is responsible for the conceptual integrity of a design, the chief engineer is responsible for demonstrating its feasibility in practice. The chief engineer will find the $R^3$ and model problem processes essential to exposing latent design risks that are otherwise masked by the complexity of components and their interactions.

**Project Manager.** Project management is concerned first and foremost with identifying and mitigating project risk. The aggressive search for technical risk that drives $R^3$ (one of the Rs is **R**isk Identification) meets these concerns. The design space provides a concise snapshot of the status of a design, and provides a structure for allocating and tracking engineering effort versus project objectives.

**Chief Technology Officer (CTO).** Modern enterprise systems are universally composed from commercial components. Such large-scale and long-lived systems never leave the design phase and, in fact, inhabit all phases of the development life cycle at all times. The CTO will find all of the concepts and techniques we describe useful for managing technology refresh.

**Software Engineers and Programmers.** The frontline developer is the true unsung hero of component-based development. Project success depends upon developers to remain current with technology trends. This book provides ammunition for developers who wish to convince their management to invest in technology training in addition to the usual process training.

## HOW TO READ THIS BOOK

This book has three parts, as follows:

- Part I explores the engineering challenges posed by commercial components. We describe engineering techniques that meet these challenges, and describe, wherever possible, workflows for incorporating these techniques into an enclosing development process.
- Part II presents an extended case study of a project that we were involved with starting in 1998. Each chapter illustrates the challenges posed by commercial components and the techniques used to meet these challenges.
- Part III provides advice on how to get started using the techniques described in this book. We also dust off our crystal ball and make predictions about the future of component-based development.

Chapter 1 introduces the problems inherent in component-based development. Chapters 2 through 4 explain why it is necessary to abandon as unworkable some of the more staid precepts of software process. Chapter 5 describes component ensembles and blackboards, both essential concepts in their own right and for the material presented in this book. Chapter 6 defines process models for exploratory design and design risk reduction. Chapters 7 and 8 describe how design documentation developed by these processes can be managed and reused, respectively. The remaining chapters in Part I describe specific techniques (really, families of techniques) for developing component-based systems. These can be read in any order; you can also skip these and head straight for the case study and return to the techniques as needed.

The case study describes a chain of events and so these chapters are linked by a running narrative. However, the chapters are designed to be relatively standalone, although the motivation for the work described in each chapter may be less than clear if you read them out of order. Chapter 14, which provides a mini-tutorial on public key infrastructure (PKI) and security, is one exception. If you already understand PKI, skip this chapter. Otherwise, you will need to read it to understand the details of the case study.

## ACKNOWLEDGMENTS

First, the authors wish to express their gratitude to Daniel Plakosh, David Carney, and Fred Long for their contribution of chapters in this book. We also owe a debt

of gratitude to our manager, John Foreman, for his strong support for this book, without which we would not have succeeded.

We are also grateful to the reviewers of this book whose insightful comments are reflected throughout our work: Santiago Comella-Dorda, Judith Stafford, Paul Clements, Tom Shields, Hans Polzer, Will Tracz, Alan Brown, and John Dean. We also happily acknowledge the intellectual contributions of members of the SEI COTS-Based Systems project not already mentioned: Howard Slomer, Wilfred Hansen, Patricia Oberndorf, Cecilia Albert, Lisa Brownsword, Edwin Morris, John Robert, and Patrick Place.

Special thanks go to our in-house editor, Len Estrin, for his excellent editing under a tight deadline. Also, Peter Gordon from Addison-Wesley deserves our thanks for agreeing to publish this work, and for his timely interventions to keep things on track.

Last, the authors are indebted to the Software Engineering Institute (SEI) for providing an unparalleled environment for conducting research in software engineering practice. In particular, we want to acknowledge our tireless librarians Karola Yourison, Shiela Rosenthal, and Terry Ireland. We offer special thanks to Steve Cross, the Director of the SEI, for his enthusiastic endorsement of the ideas expressed in this book.

# Contents

此为试读,需要完整PDF请访问: www.ertongbook.com

此为试读,需要完整PDF请访问: www.ertongbook.com