SEI 软件工程丛书 · 影印版

用商业组件构建系统

Building Systems from Commercial Components

库尔特 · C · 瓦尔诺 [Kurt C. Wallnau]
斯哥特 · A · 希萨姆 [Scott A. Hissam] 著
罗伯特 · C · 塞克德 [Robert C. Seacord]

清华大学出版社

SEI 软件工程丛书 · 影印版

用商业组件构建系统

Building Systems from Commercial Components

库尔特 · C · 瓦尔诺 [Kurt C. Wallnau]
斯哥特 · A · 希萨姆 [Scott A. Hissam] 著
罗伯特 · C · 塞克德 [Robert C. Seacord]

清 华 大 学 出 版 社

（京）新登字 158 号

内 容 简 介

商业组件的广泛使用给软件工程学带来了全新的挑战。商业组件的复杂性和商业市场中的不确定因素使得软件人员必须适应从制定组件规范到集成现有商业组件的转变。本书全面、深入地介绍了各种扩展的软件工程思想、方法，并结合实例详细分析、阐述了各种方法、技术在实际开发中的运用过程。

本书适合所有从事或希望从事软件开发工作的人士阅读。

书　　名：用商业组件构建系统
作　　者：[美] 库尔特・C・瓦尔诺，斯哥特・A・希萨姆，罗伯特・C・塞克德
责任编辑：尤晓东
出 版 者：清华大学出版社（北京清华大学学研大厦，邮编 100084）
　　　　　http://www.tup.tsinghua.edu.cn
印 刷 者：世界知识印刷厂
发 行 者：新华书店总店北京发行所
开　　本：787×960 1/16　　印张：26.75　　插页：1
版　　次：2002 年 8 月第 1 版　2002 年 8 月第 1 次印刷
书　　号：ISBN 7-302-05754-0/TP・3401
印　　数：0001～4000
定　　价：49.00 元

作者简介

本书3位作者均为卡内基·梅隆大学软件工程研究所（SEI）的资深专家，具有深厚的理论底蕴和丰富的实践经验。

库尔特·C·瓦尔诺
Kurt C. Wallnau

SEI的COTS系统项目小组负责人，负责使用确定组件进行可预测装配项目，他使用基于组件的开发方法设计并教授CMU/MSE课程，从事研究和开发工作20余年。

斯哥特·A·希萨姆
Scott A. Hissam

匹兹堡大学的副教授，具有15年的软件开发经验，曾在Lockheed Martin航空航天工业公司和Bell Atlantic通信公司担任项目负责人。

罗伯特·C·塞克德
Robert C. Seacord

至少有17年软件开发经验，深谙Enterprise JavaBeans, CORBA及Web技术，曾经是X Consortium（X协会）和IBM技术小组的成员。

出 版 说 明

1984 年，美国国防部出资在卡内基·梅隆大学设立软件工程研究所(Software Engineering Institute，简称 SEI)。SEI 于 1986 年开始研究软件过程能力成熟度模型(Capacity Maturity Model, CMM)，1991 年正式推出了 CMM 1.0 版，1993 年推出 CMM 1.1 版。此后，SEI 还完成了能力成熟度模型集成(Capability Maturity Model Integration，简称 CMMI)。目前，CMM 2.0 版已经推出。

CMM 自问世以来备受关注，在一些发达国家和地区得到了广泛应用，成为衡量软件公司软件开发管理水平的重要参考因素，并成为软件过程改进的事实标准。CMM 目前代表着软件发展的一种思路，一种提高软件过程能力的途径。它为软件行业的发展提供了一个良好的框架，是软件过程能力提高的有用工具。

SEI 十几年的研究过程和成果，都浓缩在由 SEI 参与研究工作的资深专家亲自撰写的 SEI 软件工程丛书（SEI Series In Software Engineering）中。

为增强我国软件企业的竞争力，提高国产软件的水平，经三联四方工作室和清华大学出版社共同策划，全面引进了这套丛书，分批影印和翻译出版，这套丛书采取开放式出版，不断改进，不断出版，旨在满足国内软件界人士学习原版软件工程高级教程的愿望。

前　　言

在基于组件软件设计的理论和实践之间确实存在着差异，并且这种差异正在日益扩大。

市面上也有一些关于基于组件的设计方面的书籍，但这些书都假设设计任务就是为软件组件开发设计规范。实际上，绝大多数基于组件的设计工作都依赖于现有的组件。这两种观点都有自己的市场。但现有组件带来了新的挑战，同时，使用这些组件也变得日益重要起来。现有组件意味着现有的组件标准，这对于设计者来说是一种自然的约束。

当前基于组件的设计方法集中在某些特定的问题上，这些问题让我们提不起兴趣，出现这些问题的情况也很少。设计方法中更通用、更令人感兴趣的东西是那些设计人员不再能够把握的方面。

- 与设计组件标准不同，使用现有的组件会带来完全不同的另一类设计上的问题。现有的组件使得设计者面临选择决策，而自由定义组件接口的方式又使得设计者面临优化决策。对于软件工程师来说，这些开发方式间的差异是逐渐显现出来的，但是设计方法并没有得到足够的重视。

- 使用现有组件使我们对底层设计决策失去了控制。这些决策包括：系统是怎样分布到组件中的，组件都提供了什么样的功能，组件之间如何协作。在软件工程理论中，这些都属于构架决策方面的内容。这将导致错误的结论，即广泛使用现有组件是违反或至少不适应软件设计方法的。

我们简要地讨论了基于组件的设计方法的现状，但还没有得出结论说在这种设计模式的理论和实践中存在着日益扩大的差异。事实上，一旦你明白如何找到它，这种差异的存在就是不言自明的了。

基于组件的开发模式已经流行了 15 年，并且已经在商业软件中扎下了根基。许多软件产品，例如：关系数据库管理系统、事务监测器、邮件代理程序、事件管理器、加密服务、WEB 浏览器和服务器、图形信息系统、产品数据管理系统以及其他很多产品，它们都符合软件组件的标准，或至少为业界所了解。这就是说，这些软件都实现了某种功能，表现为二进制形式，能独立分发，由一个程序接口所描述，并且支持第三方的集成。

商业市场是软件组件的主要来源。现阶段是这样，在将来无法预期的一段时间内也将保持这种现状。我们认为组件以及软件组件市场之间存在着明显的联系。Szyperski 在他很有影响的著作中也认同这个观点，他注意到组件必须切合市场的需要。Szyperski 关于市场的说法在很大程度上（但并不完全是）有些晦涩。与他相反，我们这里所提及的组件市场是指现在已经存在的状况，这其中包括组件供应商、组件基础构架供应商、第三方组件集成商以及最终的用户。

忽视市场在软件工程中的作用就如同在机械工程中忽略了摩擦一样。特别指出的是，商业软件组件具有三个方面的特点，这三个特点共同作用，导致了由软件组件带来的挑战。

（1）商业软件组件是复杂的。对于组件市场来说，这种复杂性也是必须的。很多组件非常复杂以至于使用它们的专家也不了解它们全部的特性。对于这些组件的特性和行为总是有不了解的地方。

（2）商业软件组件是与众不同的。标准有用，但是吸引顾客的往往是组件的一些新特性。这意味着关于组件的知识是由顾客所定义的，并且这些新特性（也是非标准的特性）之间的不匹配造成了集成的难度。

（3）商业软件组件是不稳定的。必须引入新的特性以使其升级，并且当竞争者也模仿了这些成功的特性后，组件也必须引入新的特性。组件知识的半衰期很短，并且设计上基于组件特性的假设也是靠不住的。

在部署现实系统时发现了软件组件的这些特点。这些特点使得关于软件设计是一个有序过程的假设也变得让人迷惑起来，而且这个假设是传统软件设计方法的基础。既然所有基于组件的开发方法都会涉及到商业组件市场的问题，所以我们需要从方法学上解释这些新的复杂性。

方法学上的回应

有关方法的一个中心问题就是在基于组件的软件设计中，我们缺乏某些知识，这些知识涉及到组件是如何被集成起来的，以及集成后是如何运作的。要减少这种风险，基于组件的设计模式中自然地就包括着探索以及发现的机制。获取和维持技术（组件）能力是这种探索活动的主要目的。

这种观点看起来像是对标准软件开发方法所取得的进步的背叛，这种进步强调管理经验而不是技术经验，强调群体行为而不是个体的贡献。其实，一方面，在组件集成细节方面，我们提倡仔细推敲；另一方面，在软件工程方面，关于额外的高

深技术能力，我们提倡的却是超越个人英雄主义。这些都是与我们对软件方法中一些重要原则的理解，以及基于组件的开发现实相违背的。事实上一个设计能否完成常常依赖于仔细推敲。进而，整个设计概念常常依赖这些底层的细节。一个无法回避的事实是，如果需要把握细节的话，精通技术是十分重要的。

下面是我们在方法学上改进的一些核心要素。

（1）我们把组件集成块看作是一个基础的设计抽象。集成块提出了组件独立性的要求，并且将我们工作的重点由选择单个的组件转移到选择能协调工作的组件集合上来（这就是集成块）。

（2）我们将黑板理解为一个基础的设计符号。黑板描述了有关集成块已知的知识，并且，更重要的是，描述了将被发现的事物。黑板用于将一个设计项目及已知的设计风险文档化。

（3）为了暴露设计风险及定义集成块的可行性标准，我们引入了一个风险驱动的发现过程，名为 R^3。为产生合适的组件技术以及确立集成块的可行性，我们还引入了一个原型化方法，名为模型问题。

（4）根据集成块的关系及断言，我们引入了设计空间的概念。设计空间的概念用于提供集成之间的独立性，它被用来作为对预期市场行为的响应，例如新组件的发布。同时当集成块的可行性遇到问题时也能处理设计上的一些障碍。

这种方法学上的改进将用以面对由商业组件市场带来的挑战。它一方面防止将设计过程变为一项随意性的工作，另一方面防止某些具有创新意义但不稳定的技术主宰设计过程，导致额外但没有必要的设计风险。我们相信，这里所讨论的方法可以面对这种挑战。

本书的目标

我们的目标非常明确。第一个目标就是展现这个事实，即软件组件对软件工程提出了新的方法学上的挑战。围绕这个问题，我们希望阐明这些挑战的本质，并特别地将重点放在由组件市场变化所带来的挑战上。我们的第二个目标是详细阐述用以应付这些挑战的方法和技术。我们认为对于任何软件组件的方法学改进来说，这些方法和技术都是必要的基础。我们的最终目标是：对一个现实意义中的案例进行研究，这个案例来自于我们在开发一个大型企业系统时积累的经验。通过这些研究，解释组件设计的复杂性，以及推荐我们的方法和技术所带来的效应。

本书的读者对象

本书适合那些从事基于组件进行开发的人阅读，同时也适合于软件工程专业的学生。尽管本书所有的章节都为所有的读者提供有用的信息，还是应该根据需要确定不同的重点。

系统设计师：首席设计人士，可以从本书中找到有关集成块、集成修正和集成块可行性方面的技术，以及和他或她专业技能相关的附加知识。通过对设计空间的学习，系统构架者将掌握概念性的语言。通过该语言，他们从多个层面处理紧急情况，并且修正那些各具不同复杂特点的基于组件的系统。

总工程师：系统设计师主要把握设计概念上的完整性，而总工程师则要在实践中把握系统的可行性。如果不加以重视，复杂的组件以及它们之间的相互作用将掩盖系统中潜在的风险。总工程师通过使用 R^3 及模型问题方法能够将这些风险暴露出来。

项目经理：项目管理者最先考虑到项目的风险，以及如何降低这种风险。来自 R^3 过程（R^3 之一即风险识别）的主动搜索技术将满足这种需要。设计空间为项目管理者精确地提供了对有关设计状态的即时描述，并且提供了一种构架，通过这种构架可以为设计工作提供对于项目目标的定位以及跟踪。

首席技术官：现代企业级系统普遍由商业组件构成。这种大规模、长期使用的系统的成败与设计阶段密不可分。事实上，在任何时候，这种系统都蕴涵着开发周期中的各个阶段。首席技术官将发现这里讲述的所有概念和技术都适合于管理技术上的更新。

本书也适合于软件工程师和程序员，在基于组件的系统开发中，一线的开发人员是真正的无名英雄，项目的成功依靠开发人员保持技术趋势的正确方向。这本书使得开发人员有充足的理由说服管理者重视除了开发方法培训之外的技术培训。

本书阅读方法

本书包括三个部分：

- 第 I 部分讨论了由商业组件带来的挑战；解释了用以应对这些挑战的软件工程技术；就如何将这些技术整合到现有的开发过程中去，讨论了工作流技术。
- 第 II 部分将一个项目作为案例进行了研究，这是我们在 1998 年开发的一个项目。这一部分的每一章都阐述了

商业组件带来的挑战以及应对这些挑战所用到的技术。

- 第III部分就如何运用本书提出的技术给出了一些建议。就将来基于组件的开发模式，我们也做出了一些预见。

第 1 章介绍了由基于组件开发模式所带来的问题。第 2 章到第 4 章解释了为什么有必要放弃软件方法中某些呆板的规则。第 5 章讲述了组件集成块和黑板。这些概念都非常重要，并且将贯穿本书。为进行探索式设计和减少风险，第 6 章定义了过程模型。第 7 章和第 8 章讨论了如何分别管理和使用由这些方法开发出来的设计文档。第一部分的余下章节讨论了一些具体的技术（确切地说应该是一些技术族）用以开发基于组件的系统。这些章节可以以任意的顺序阅读。您也可以先跳过这些章节，等到学习案例研究时再根据需要返回来学习这些章节。

在案例研究中描述了一系列的事件，这些章节也按照事件的顺序编排。假如您没有按照顺序阅读这些章节，可能不会很了解这些章节内容的写作动机。但是这些章节又是相对独立的。第 14 章是个例外，它提供了有关公共密钥基础设施（PKI）和安全问题的简短介绍。如果您已经了解有关 PKI 的知识，可以略过该章，否则，您还是需要阅读该章以便清楚地了解案例研究中的细节。

To my wife, Jeannemarie, and to my children, Zachary, Victoria, Gemma, and Bryn. You were all beginning to suspect that I would never finish the book. So was I. Without your love and support, I never would have.

—KW

I deeply want to thank my wife Jackie for her patience and support in the authoring of this book, she had to endure some of the early chapters and now knows more about PKI than she ever wanted to. I also want to thank my sons, Derek and Zachery, for showing me how to build systems from Legos! Love and thanks to them as they give reason to everything I do.

—SH

I would like to acknowledge my family for supporting my efforts in writing this book. My wife Rhonda, whose encouragement and support was essential, and my daughter Chelsea and my son Jordan without whom everything would be meaningless.

—rCs

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 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.