软件工程丛书
Software Engineering Series

英文版

基于重用的软件工程
——技术、组织和控制

Reuse-Based Software Engineering:
Techniques, Organization, and Controls

[美] Hafedh Mili Ali Mili Sherif Yacoub Edward Addy 著

WILEY
PHEI
電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

软件工程丛书

基于重用的软件工程
——技术、组织和控制

（英文版）

Reuse-Based Software Engineering:
Techniques, Organization, and Controls

[美] Hafedh Mili
Ali Mili
Sherif Yacoub
Edward Addy 著

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

本书主要介绍了基于重用的软件工程的实践模型和其他有关的基础问题，全面分析了基于重用的软件工程的当前状态和未来发展，重点讨论了软件重用的关键技术、管理和组织问题，详细论述了这些理论知识在基于组件的软件开发生命周期和产品线工程中的运用问题。全书的结构清晰，内容全面，并且提供了相关练习，可以使读者对软件开发有更深刻的认识。

本书适用于计划推进本单位软件重用实践的管理和技术人员，同时对计算机应用专业的研究生和高年级本科生也有很好的参考价值。

Hafedh Mili, Ali Mili, Sherif Yacoub, Edward Addy: **Reuse-Based Software Engineering: Techniques, Organization, and Controls.**
ISBN 0-471-39819-5
Copyright © 2002, John Wiley & Sons, Inc. All Rights Reserved.
AUTHORIZED REPRINT OF THE EDITION PUBLISHED BY JOHN WILEY & SONS, INC., New York, Chichester, Weinheim, Singapore, Brisbane, Toronto. No part of this book may be reproduced in any form without the written permission of John Wiley & Sons, Inc.
This reprint is for sale in the People's Republic of China only and exclude Hong Kong and Macau.
English reprint Copyright © 2003 by John Wiley & Sons, Inc. and Publishing House of Electronics Industry.

本书英文影印版由电子工业出版社和 John Wiley & Sons 合作出版。此版本仅限在中华人民共和国境内（不包括香港和澳门特别行政区）销售。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号：图字：01-2003-0601

图书在版编目（CIP）数据

基于重用的软件工程——技术、组织和控制 = Reuse-Based Software Engineering: Techniques, Organization, and Controls /（美）米利（Mili, H.）等著. -北京：电子工业出版社，2003.4
（软件工程丛书）
ISBN 7-5053-8624-7

Ⅰ.基… Ⅱ.米… Ⅲ.软件工程-英文 Ⅳ. TP311.5

中国版本图书馆 CIP 数据核字（2003）第 022576 号

责任编辑：冯小贝
印 刷 者：北京天竺颖华印刷厂
出版发行：电子工业出版社 http://www.phei.com.cn
北京市海淀区万寿路 173 信箱 邮编：100036
经 销：各地新华书店
开 本：787 × 980 1/16 印张：41.5 字数：930 千字
版 次：2003 年 4 月第 1 版 2003 年 4 月第 1 次印刷
定 价：59.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077

Preface

Software productivity has been steadily increasing since 1970, but not enough to close the gap between the demands placed on the software industry, and what the state of the practice can deliver. Today, as software costs continue to represent an increasing share of computer system costs, and as software faults continue to be responsible for many expensive failures, nothing short of an order-of-magnitude improvement in software quality and development productivity will save the software industry from its perennial state of crisis. Reuse-based software engineering has been touted, since the late 1960s, as the only practical and realistic approach that can deliver such improvements "in the short term." This book provides a state of the art and the practice of all aspects of reuse-based software engineering, and attempts to explain why the "short term" keeps eluding us, and how to catch up.

Despite decades-long intensive research in software engineering and artificial intelligence, software generation remains an elusive goal. Progress has been made, of course, but the target has continued to move even faster, outdistancing the kind and scale of software that techniques can deliver. Hence, short of generative sufficiently understanding the process we use to build software from imprecise and contradictory user needs, subject to all the contradictory constraints of cost, reliability, time to market, and portability—to name a few—to encode it into a generator, we could try to reuse the products or repeat the development processes, of previous developments efforts. The underlying hypotheses are that (1) the computer systems that we develop today have a lot in common and (2) by reusing the process and products of previous development efforts to solve new problems, we increase productivity and improve the quality of the resulting system.

Reuse is not only natural but may be the key to progress. Psychologists and cognitive scientists have long argued that we humans seldom solve problems from first principles. When faced with a problem to solve, we first perform a "rote recall," just in case this problem was already solved. When that fails, we perform "approximate recall," with the hope of identifying an already solved problem that is so close that its solution can be adapted, locally, to address the new problem. Only when that fails do we fall back on analytic problem solving, at least as far as decomposing the initial problem into a set of more manageable subproblems. The key to progress, some would argue, is to reuse (learn from past experiences), and to communicate (be able to transmit knowledge to new generations).

In the early days of the profession, because of the scale of the systems to be built (small), the expressiveness of the programming languages (low level), and the cost distribution of computer systems (machines and machine time predominant), it was felt that much benefit would accrue from reusing *executable programs*. Much research effort in software engineering in general and software reuse in particular went into software packaging issues, and more specifically, language features that support modularization and abstraction. A lot of progress has been made along these dimensions, but again, the target has continued to move; as the size and complexity of typical software systems kept increasing, organizational aspects of both the software product and the teams building it began to dominate the software development process, in terms of complexity, cost, and impact in general. This meant that software reuse research needed to focus on ways to encode, package, and organize software artifacts of a granularity that is bigger than the procedure or routine, and at a stage of development that is earlier than code. Simultaneously, as the number of stakeholders increased, and the financial stakes grew higher, there was a need to focus on the organizational and economic aspects of reuse.

Since the 1970s, a number of organizations have recognized the potential for software reuse to improve productivity and enhance the quality of the products being built. In the early 1980s, the (U.S.) federal government in general, and the Department of Defense in particular, have launched a number of initiatives to help understand, organize, and promote software reuse, within its software providers community, and throughout the software industry in general. A number of organizations had embarked on software reuse initiatives since the 1970s. The results varied widely, and in those cases where reuse was successful, the approach used was sometimes not repeatable, not scalable, or both, and its benefits were nonmeasurable. Nowadays, it is widely recognized that in order to attain worthwhile, predictable, and repeatable reuse levels at the scale of the enterprise, we must address software reuse at three levels:

- A set of techniques for developing and packaging high-quality software artifacts that are widely applicable, and cost-effectively usable
- An organization that has the mandate, the discipline, the skills, and the resources for producing, *consuming*, and managing a shared repository of reusable artifacts

此为试读，需要完整PDF请访问：www.ertongbook.com

- A set of control and management tools for planning, controlling, and evaluating the degree to which such an organization attained its objectives

This book covers all three sets of issues, and the breadth and depth of coverage that we strove for in all three areas has made this book worth writing and we should hope will make it worth purchasing, and perhaps even reading!

There has been much research, and a growing body of codified practical knowledge in all three areas. However, the three areas have not attained the same level of maturity, and the contents of this book reflect that reality. Perhaps the most mature area is the technical field; research on reuse-enabling software packaging technologies has been ongoing since the late 1960s, with practice increasingly closing the time lag—and sometimes reversing it, by producing techniques and tools faster than researchers can conceptualize. Here the challenge is simply to keep up! In this area, we built on our collective experiences from learning about, doing, consulting, mentoring, and teaching various reuse-enabling technologies. For this knowledge to have a shelf life that is longer than that of the latest fad, be it a language, tool, technique, or method, we have to look for, and try to codify, some of the fundamentals of modeling and programming for reuse. We hope that this material will help complement the more practical skills and techniques that this book contains, by tooling the "introspective professional" with some of the skills to consume, if not develop, the next generation of techniques.

The organizational aspects of software reuse have received increasing attention since the late 1980s, both as a distinct body of knowledge and in relation to more general issues such as process maturity frameworks and issues related to process reengineering and change management. Input for organizational models has come both from (1) synthesizing known success stories of reuse organizations in the industry and (2) by analytically building such models by drawing on general knowledge from process reengineering, risk management, and change management. The result has been a set of more or less elaborate organizational models drawn up by public (e.g., government), parapublic [e.g., Software Engineering Institute (SEI)], and industry consortia [e.g., Software Productivity Consortium (SPC)]. Each one of these—relatively similar, and invariably commonsensical—models would warrant a full book in its own right. We chose to synthesize this knowledge, and focus on the commonalities, but provide pointers to the individual sources, as well as to known cases where such organizational models—or variants thereof—were successfully used.

The area of economic modeling of software reuse is perhaps the least mature of the three. Central to economic modeling is the issue of reuse measurement. Software measurement has, for a long time, been the lot of a small community of determined software engineering researchers and professionals, but not afforded the sexy status of the more mainstream research subjects. In practice, it is one of the main reasons why most software organizations are not above level 1 of the capability maturity model (CMM); very few software organizations measure software and the processes that produce it, and even fewer know what to do with the data. The area of software measurement is witnessing a boom of sorts, mostly in terms of software quality measurements, and in relation to reengineering and maintenance. However, this boom is not likely to benefit the economic modeling of software reuse in the short run, because in economic modeling, we are interested in functional size measurements, to which software reuse adds layers of complexity. The material related to the economic modeling of software reuse is a mixture of a tutorial and a literature survey, although the models that we describe can be valuable simulation tools for the reuse strategic planner and/or manager.

The intended audience for this book is technical. The material in this book is either irrelevant or too detailed for most managers. There are a number of excellent books out there whose contents and style of delivery is targeted at the hurried manager who needs bullet points to include on next day's plan review meeting.

We strove to make this book useful to both the academic and the professional by including foundational material as well as practical and tutorial material. We have also included review questions and exercises to support the teaching of the material in an academic (mostly) or professional training setting.

The Professional. The material covered in this book will be useful to developers and technical managers and leaders. A developer can skip right up to Part III, read Chapters 7 and 8, and then on to Part IV, where the introduction gives a detailed roadmap for Chapters 10–14. Chapters 15 and 17 in Part V are also a must read; Chapter 16 (on component retrieval) may be skipped. Curious developers can read the one chapter in Part VII (specialized forms of reuse) that is more relevant to their organizations or projects. Technical managers would benefit from Parts I–III (except Chapter 9), the last two chapters of Part IV (application frameworks, Chapter 13; architectural frameworks, Chapter 14), and all of Part VII. Technical leaders (e.g., architects) would benefit from Parts I and III–V (possibly skipping some of the foundational material in Part IV; see the introduction to that part), and the chapter that most corresponds to their development practice from Part VII.

The Academic. This book is suitable for a sixth- or seventh-semester advanced software design course. We recommend Chapter 1, Chapters 6 and 7 of Part III, the entire Parts IV and V, and, depending on how much general software engineering background the students has, perhaps Chapter 19 (component-based software engineering). The book is also recommended as a graduate course in computer science or management information systems (MIS). For computer science students, Part VI could be left out entirely, and some of the introductory material on object oriented (OO) techniques (Chapter 9) could be skipped. For MIS students, we could cover Parts I and II, all of Part III except Chapter 9, the introductions to Parts IV and V (Chapters 10 and 15), and Parts VI and VII. Some chapters include review questions and exercises. The uppercase letters in Exercise sections indicate level of problem difficulty or complexity: (A) easiest, (B) intermediate or medium difficulty, (C) hardest or maximum difficulty, and (R) research problem.

We hope that you will have as much fun using the material in this book as we have had writing it.

HAFEDH MILI, ALI, MILI, SHERIF YACOUB, AND EDWARD ADDY

August 2000

Acronyms and Symbols

AA	assessment and assimilation
ACT	annual change traffic
ADL	architecture description language
ADN	adaptive dynamic network
AFMC	Air Force (U.S.) Material Command
AI	artificial intgelligence
AOP	aspect-oriented programming
API	application program interface
APL	Array Programming Language
ARBV	average return on book value
ARC	Army (U.S.) Rescue Center
ARR	average rate of return
ASL	application-specific language
ASSET	Asset Source for Software Engineering Technology
ATA	architecture tradeoff analysis
ATM	Automated Teller Machine
AWT	abstract windowing toolkit
BIDM	basic interoperability data model
BS	behavioral sampling
CAD	computer-aided/assisted design
CARDS	Comprehensive Approach to Reusable Defense Software (proprietary to USAF–NASA)
CASE	computer-aided/assisted software engineering

Common abbreviations (i.e., CPU, IEEE, R&D, etc.) omitted here. Proprietary definitions are capitalized.

CBSD	component-based software/system development
CBSE	component-based software/system engineering
CCC	credit card company
CCL	Command Center Library
CCPL	command and control product line(s)
CIM	Center of Information Management
CLOS	Common LISP (list processing) Object System (language)
CM	Configuration Management (proprietary to SEI)
CMM	Capability Maturity Model
COCOMO	constructive cost model
COM	Component Object Model (Microsoft)
CORBA	Common Object Request Brokerage Architecture
COTS	commercial off-the-shelf [product(s)]
CRA	car rental agency
CRC	cyclic redundancy check
CT	coding and (unit) testing
CTA	Computer Technology Associates
C2AI	Command and Control Architecture Infrastructure
DADP	domain analysis and design process
DARPA	Defense Advanced Research Projects Agency
DBMS	database management system
DCE	distributed computing environment
DD	detailed design
DII	dynamic invocation interface
DISA	Defense Information Systems Agency
DLL	dynamically linked library
DM	design modification
DSRS	Defense Software Repository System
DSSA	domain-specific software architecture
EAF	effort adjustment factor
EJB	Enterprise Java Beans
ESC	Electronic Systems Center [U.S. Air Force (USAF)]
FAST	family-oriented abstraction, specification, and translation
FIFO	first in/first out
FODA	feature-oriented domain analysis
4GL	fourth-generation language
FSP	full-time software person/programmer
GUI	graphical user interface
IC	investment cost(s)
IDL	interface definition language
IM	integration modification
IMS	information management system
IIOP	Internet Inter-ORB Protocol
IRR	internal rate of return
ISO	International Standardization Organization

IT	information technology; integration testing
JDBC	Java database connectivity
JIAWG	Joint Integrated Avionics Working Group
JODA	joint object-oriented domain analysis
KAPTUR	knowledge acquisition for preservation of tradeoffs and underlying rationales
KLOC	kiloline(s) of code
LNCS	Lecture Notes in Computer Science
LI	library insertion
LM	labor-month(s)
LMFS	Lockheed Martin Federal Systems
LMTDS	Lockheed Martin Tactical Defense Systems
LOC	Lines of code
MIL	module interconnection language
MIS	management information system
ML	machine language
MVC	model view conroller
NPLACE	National Product Line Assessment Center
NPV	net present value
NTT	Nippon (Japan) Telegraph & Telephone Corporation
ODBC	object database connectivity
ODM	organization(al) domain modeling
OLE	object linking and embedding
OMA	Object Management Architecture (proprietary to OMG)
OMG	Object Management Group
OMT	object modeling technique
OO	object orientation
ORB	object request broker
ORRA	organizational engineering for reuse assessment
OSI	Open Systems Interconnect (protocol)
PASTA	process and artifact state transition abstraction
PBV	payback value
PCTE	portable common tool interface
PD	product design
PDL	Program Design Language
PI	profitability index
PLA	product-line architecture
PLAF	pluggable look and feel
PLE	product-line engineering
PLP	product-line practice
PRISM	Portable Reusable Integrated Software Module
PROLOG	programming in logic
PuLSE	Product-Line Software Engineering (proprietary to Fraunhofer Institute for Experimental Software Engineering)
QA	quality assurance

此为试读，需要完整PDF请访问：www.ertongbook.com

RA	requirements analysis
RAASP	Reusable Ada (language) Avionics Sofware Package(s) (U.S. Air Force)
RBP	relative blackbox price (default value 0.40; see Chapter 19).
RCA	relative cost of adaptation (default value 0.67; see Chapter 19).
RCDE	relative cost of domain engineering (default value 0.20; see Chapter 19)
RCM	reuse capability model
RCR	relative cost of reuse (of software) (default value 0.20; see Chapter 19)
RCWR	relative cost of writing for reuse (default value 0.15; see Chapter 19)
REBOOT	reuse based on object-oriented techniques
RIC	Reuse Information Clearinghouse
RICC	Reusable Integrated Command Center (proprietary program)
RLIG	Reuse Library Interoperability Group (also abbreviated RIG)
RLPM	reuse library process model
RMI	remote method invocation
ROI	return on investment
RPC	remote procedure call
RSL	reusable software library (a, generic)
RWP	relative whitebox price (default value 0.20; see Chapter 19)
SAAM	Software Architecture Analysis Method
SA/SD	Stucture Analysis/Design
SAIC	Science Applications International Corporation
SCAI	space command and control architectural infrastructure
SCM	service control manager
SD	start date; standard deviation
SEI	Software Engineering Institute
SEL	Software Engineering Laboratory
SLA	savings & loan association (a, generic)
SORT	Software Optimization and Reuse Technology (NASA)
SPARC	Scalable Processor Architecture
SPC	Software Productivity Consortium
SQL	Structured Query Language
SRI	Software Reuse Initiative (proprietary program)
SRSC	software reuse support center
STARS	Software Technology for Adaptive and Reliable Systems
SU	software understanding
SWSC	Space and Warning Systems Center
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling/Medical Language
URL	uniform resource locator
V&V	verification and validation
VBX	Visual Basic Controls (Microsoft)

WAP	Wireless Application Protocol
WVHTCF	West Virginia High Technology Consortium Foundation
$\rho(P)$	Cost of developing product P with reuse (see Chapter 19).
$\pi(A)$	Cost of purchasing reusable asset A (see Chapter 19).
$\sigma(S)$	Cost of developing product S from scratch (see Chapter 19).
$\theta(W)$	Cost of (whitebox) reusing asset W (see Chapter 19).
$\beta(B)$	Cost of (blackbox) reusing asset B (see Chapter 19).
$\tau(S,B,W)$	Cost of integrating components S, B and W (see Chapter 19).
$\omega(P)$	Labor overhead incurred by the development of project P as a result of practicing software reuse

Contents

此为试读，需要完整PDF请访问：www.ertongbook.com