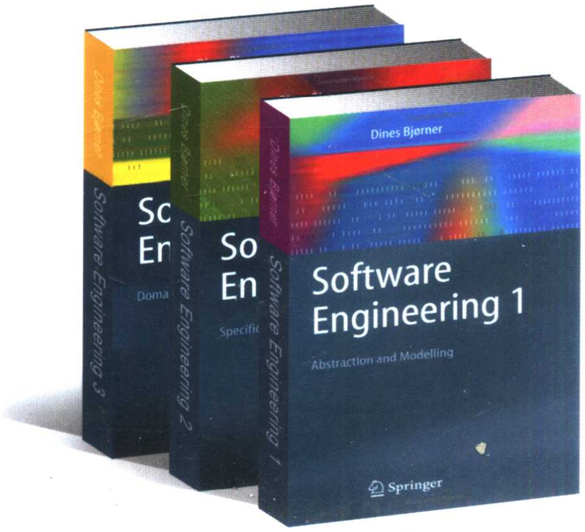
Springer

大学计算机教育国外著名教材系列（影印版）

SOFTWARE ENGINEERING 1
ABSTRACTION AND MODELLING

软件工程 抽象与建模
卷1

Dines Bjørner 著



清华大学出版社

大学计算机教育国外著名教材系列（影印版）

Software Engineering 1

Abstraction and Modelling

软件工程 卷 1

抽象与建模

Dines Bjørner

清华大学出版社

北京

图书在版编目(CIP)数据

软件工程 卷 1：抽象与建模＝Software Engineering 1: Abstration and Modeling: 英文 / (丹) 比约纳 (Bjørner, D.) 著. —影印本. —北京： 清华大学出版社, 2007.7
（大学计算机教育国外著名教材系列）
ISBN 978-7-302-15428-0

Ⅰ. 软… Ⅱ. 比… Ⅲ. 软件工程—高等学校—教材—英文 Ⅳ. TP311.5

中国版本图书馆 CIP 数据核字（2007）第 086860 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http:// www.tup.com.cn　　**邮　　编**：100084
c-service@tup.tsinghua.edu.cn
社 总 机：010-62770175　　**邮购热线**：010- 62786544
投稿咨询：010-62772015　　**客户服务**：010-62776969
印 刷 者：北京密云胶印厂
装 订 者：三河市金元印装有限公司
发 行 者：全国新华书店
开　　本：185×230　**印张**：47.5
版　　次：2007 年 7 月第 1 版　2007 年 7 月第 1 次印刷
印　　数：1~3000
定　　价：79.00 元

出 版 说 明

进入 21 世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的竞争。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从 1996 年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书（影印版）”等一系列引进图书，受到国内读者的欢迎和支持。跨入 21 世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高校本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列（影印版）”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列（影印版）”做得更好，更适合高校师生的需要。

清华大学出版社

中 文 序

信息化社会的发展越来越依赖于软件。如何在期望的时限内、以可承受的开销、开发具有满意质量的软件成为人们特别是软件开发人员所关心的问题。自 1969 年 NATO 会议首次提出“软件工程”一词以来，软件工程已成为了一门学科。IEEE 的 SWEBOK 和 SEEK，以及 ACM/IEEE 的计算教程 2005 相继发布，标志着软件工程的学科内涵及其教育逐步走向规范与成熟。

软件形式化开发方法是软件工程的重要组成，荟萃了软件工程在原理和科学上的许多精华，被认为是开发可靠、安全软件的重要途径。Dines Bjørner 教授所著的《软件工程》（卷 1～3）系统地介绍了软件形式化开发方法，包括数学基础、形式化模型与建模、形式化规约、领域与需求分析以及软件设计等内容。该书对如何运用形式化方法进行软件开发进行了比较全面的阐述。Dines 是软件工程领域的著名专家，是著名的 VDM 方法和 RAISE 方法的创建者之一，因其在形式化软件开发方法及其在工业界的应用的突出贡献和倡导作用而获选 ACM Fellow 和 IEEE Fellow。这本书是他长期研究和实践的总结和提炼，内容丰富，想必会令中国的软件工程人员开卷有益。

很高兴得知《软件工程》（卷 1～3）的英文影印版和中文翻译版即将出版，应老友之邀，提笔作序，以资庆贺。

陈火旺

于梦泽园

Preface — to Vols. 1–3

This preface covers the three volumes of *Software Engineering*, of which this volume is the first.

- **Software engineering — art/discipline/craft/science/logic:** Software engineering is the art [326–328], discipline [194], craft [441], science [245], logic [275] and practice [276] of
  - ⋆ **synthesizing** (i.e., building, constructing) software, i.e., technology, *based on scientific insight*, and
  - ⋆ **analysing** (i.e., studying, investigating) existing software technology *in order to ascertain and discover its possible scientific content.*

To succeed in this,

- **Software engineering — abstraction and specification:** Software engineering makes use of abstraction and specification.
  - ⋆ **Abstraction** is used to segment development into manageable parts, from high-level abstractions in phases, stages and steps to low-level abstractions, i.e., concretisations.
  - ⋆ **Specification** records and relates all levels of abstraction.

Volumes 1 and 2 of the three-volume book cover abstraction and specification in detail.

- **Software engineering — the triptych:** Software engineering composes analysis of *application domains* with synthesis and analysis of *requirements* (to new software) into *design* (i.e., synthesis and analysis) of that *software*. Hence software engineering consists of
  - ⋆ **domain engineering,** which, as these volumes will show you, is a rich field of many disciplines, etc.,
  - ⋆ **requirements engineering,** which, as we shall again see, in these volumes, has many aspects and facets not usually covered in textbooks, and

- ⋆ **software design,** with concerns of software *architecture*, *component* composition and design, and so on.

Volume 3 of the book covers this triptych in detail.

- **Software engineering — practical concerns:** Software engineering, besides, consists of many *practical concerns:* Project and product management; principles, techniques and tools for making sure that groups of possibly geographically widely located people work effectively together, for choosing, adapting, monitoring and controlling work according to one of a variety of development process models; planning, scheduling and allocating development resources (people, materials, monies and time); and related matters, including cost estimation, legacy systems, legalities, etc.

We shall not be covering these management-oriented facets of software engineering in this book.

• • •

Each chapter of this volume and its companion volumes starts with a synopsis. An example — relevant for this preface — follows:

- **Assumptions:** You have taken this book into your hands since you are interested in knowing about, and possibly learning a new approach to software engineering.
- **Aims:** The main aim of these volumes is to introduce you to a new way of looking at software: One that emphasises (I) that software engineering is part of informatics, and that *informatics* is a discipline otherwise based on (i) mathematics, (ii) the computer & computing sciences, (iii) linguistics, (iv) the availability of the hard information technologies (computers and communication, sensors and actuators) and, last but not least, (v) applications. Furthermore (II) that informatics "hinges" on a number of philosophical issues commonly known under the subtitles — epistemology, ontology, mereology, etc.
- **Objectives:** To help you become a truly professional software development engineer in the widest sense of that term, such as promulgated by these volumes.
- **Treatment:** Nontechnical, discursive.

To develop large-scale software systems is hard. To construct them such that they (i) solve real problems, (ii) are correct and pleasing and (iii) will serve well in the acquiring organisation is very hard.

This series of volumes offers techniques that have proven (i) to make the development of large-scale software systems much less hard than most current software engineers find it, (ii) to result in higher-quality systems than normally experienced and (iii) to enable delivery on time.

Thus we emphasise the software engineering attributes aimed at in this series: Trustworthy and believable methods, higher-quality software products, higher-quality software development projects, and the personal satisfaction of developers and acquirers, that is, the software engineers and their management, respectively the users and their management. We aim at much less, if any, frustration, and much more fascination and joy!

Reasons for Writing These Volumes

A number of reasons[1] can be given for why these volumes had to be written:

- *Formal techniques apply in all phases, stages and steps of software engineering, and in the development of all kinds of software.* But there was no published textbook available that covered software engineering, such as we shall later characterise that term, from a basis also in formal techniques (besides other, "non-formal" bases).
- Formal development (that is, specification, refinement and verification) books were more like monographs than they were textbooks, and they covered their topic from a rather narrow viewpoint: usually just specification of software, that is, of abstract software designs and their concretisation. *Formal specification, in these volumes, applies not just to software, but also to their requirements prescription, and, as a new contribution (in any book or set of lecture notes), also to domain descriptions.*
- The author of these volumes has long been less than happy with the way in which current textbooks purport to cover the subject of software engineering.
  - ⋆ "All" current textbooks on software engineering fail[2] with respect to very basic issues of programming methodology, in particular with respect to (wrt) formal techniques. If they do, as some indeed do, bring material on so-called "Formal Methods", then that material is typically "tucked away" in a separate chapter (so named). In our mind, the interplay between informal and formal techniques, that is, between informal descriptions and formal specifications, informal reasoning and formal verification, and so on, permeates all of software engineering. The potential of (using) formal techniques shapes all phases, stages and steps of development. Classical software engineering topics, such as software processes, project management, requirements, prototyping, validation (not to speak of verification), testing, quality assurance & control, legacy systems, and version control & configuration management, these auxiliary, but crucial, concerns of software engineering, can be handled better, we show, through a judicious blend of informal and formal techniques. Needless to say, these volumes will redress this "complaint".

[1]Usually, when more than one "excuse" is given for some "mistake", none apply. This series of volumes, however, is no mistake.

[2]With the notable exception of [240].

- ⋆ All current textbooks, in our mind, fail in not properly taking into account the issue of the software developer not having a thorough understanding of the domain in which the software is to be inserted, that is, the domain from which sprang the desire to have "that new software"! As mentioned above, a major new "feature" of our books is the separation of concerns illustrated in the software development process — when the developer initially spends much time and effort to understand and document an understanding of the application domain.
- ⋆ All current textbooks, in our mind, fail in not systematically, i.e., methodically, presenting principles, techniques and tools that "carry through" and "scale up". By carry through I mean principles, techniques and tools that are shown, by extensive examples, to cover all the major phases, stages and steps of development. By scaling up I mean principles, techniques and tools that can be applied to the largest-scale software development projects.
- ⋆ Some current textbooks, in our mind, fail the programming, that is, the design issues completely. There is no assumption on any methodological approach to the development of software from the point of view of programming methodology.[3]
- ⋆ Other current textbooks, in our mind, fail the stepwise refinement, that is, the implementation relation development point of view.[4]
- ⋆ And yet other current textbooks fail the design point of view.[5]
- ⋆ Finally all current textbooks fail, we believe, in not properly integrating the above, albeit more theoretical, points of view, with the points of view of mundane, engineering issues such as (i) development process models ("waterfall", "spiral", "iterative", "evolutionary", "extreme programming", etc.), (ii) quality management, (ii) testing & validation, (iv) legacy systems, (v) software re-engineering, and so on.

Shortcomings of These Volumes

The major shortcoming of the current set of three volumes is our all too brief coverage of correctness issues, that is, of the verification (theorem proving, model checking) of properties of single and pairs of (development-step-related) specifications.

[3]By the `programming` methodology point of view we mean a view that concerns itself with such issues as establishing invariants when specifying loops, as securing proper programming abstractions in terms of routines (procedures, functions), etc.

[4]By the stepwise refinement point of view we mean the concern that abstractions, even when informally expressed, are rendered into correct concretisations — when expressed as code.

[5]By the design point of view we mean the programming concern for choosing appropriate algorithms and data structures, for their justification and validation.

Elsewhere, and where appropriate in these volumes, we explain why we have not introduced substantial material on verification.

The reader, seeking this knowledge, is referred to an abundance of texts (books, and articles in journals and in proceedings), or may have to wait till we feel competent to write a textbook of sufficient generality on this topic. Current texts are very much linked to a specific notational system (i.e., specification language).

• • •

Obviously we do not know all there is to know about how to develop all possible kinds of software, and not all that we know is in these volumes. To develop software, in general, takes a diverse range of techniques and tools.

Whatever special techniques and tools we cover, we cover them to some non-trivial depth, but not to the depth that is sufficient for a professional engineer in the relevant field. For example:

- **Development of compilers:** We cover quite a lot, but not all that is necessary for the really professional compiler developer. We cover what we believe all software engineers ought know. And we cover it in a way that we find is sorely missing from all compiler textbooks. We refer to Chaps. 16–20 of Vol. 2.
- **Development of operating and distributed systems:** We cover only general principles and techniques of specifying concurrent systems.
- **Development of embedded, safety-critical and real-time systems:** Basically the same coverage as for operating and distributed systems development: We emphasise that Vol. 2 covers techniques for specifying embedded, safety-critical and real-time systems. These techniques and their underlying notations are those of Petri nets [313, 421, 435–437], message [302–304] and live sequence charts [171, 270, 325], statecharts [265, 266, 268, 269, 271], temporal logics [205, 360, 361, 400, 429] and the duration calculi [537, 538].

Chapter 28 in Vol. 3, Domain-Specific Architectures, will, however, go into some depth, showing which principles, techniques and tools apply in the development of translation systems (interpreters and compilers), information systems (database management systems), reactive systems (i.e., embedded, real-time and safety-critical systems), workpiece systems (worksheet systems), client/server systems, workflow systems, etcetera. Our treatment in that chapter is novel, and is inspired, strongly, by Michael Jackson's concept of Problem Frames [310].

Thus we cover what we believe all software engineers, whatever their specialty is, should know. And we believe they should know far more than most textbooks in software engineering offer.

As explained elsewhere, these volumes suggest that education and training in the specialised fields mentioned above can follow after having studied Vol. 3.

And much of the textbooks of those specialised fields really, then, ought be rewritten: be adapted to formal specification, and so on.

Methods of Approach

Our didactics seeks to go to the "roots of the matter". We see these roots to be formed from basic understandings of such issues as (i) the linguistics of "how to describe", (ii) the near-philosophical issues of "what to describe", (iii) the linguistic, i.e., *semiotic* issues of *pragmatics, semantics and syntax*, and (iv) the issues of constructing concise, objective formulations in terms of mathematics, i.e., of using formal specification languages (and, in turn, understanding their pragmatics, semantics and syntax — independent of the pragmatics, semantics and syntax of the application phenomena).

Thus this book begins by exploring the above four issues. In Vol. 2 we then take up this theme of semiotics (pragmatics, semantics and syntax) in four separate chapters (Chaps. 6–9 incl.).

Also this is new: Existing textbooks on software engineering completely avoid any mention of these issues. For a modern, professional software engineer to graduate from any reputable academic institution without a proper grasp on these four didactic bases (i–iv) is, to this author, unthinkable! Alas! It is today the rule rather than the exception: That they do not even see these issues at all!

A New Look at Software

These volumes will provide the reader with a new way of looking at software and at the process of developing software. They will provide the reader with an altogether dramatically different approach to understand and to develop software. That "new look" can perhaps best be characterised as follows: Software is seen as intellectual artifacts, as the product of a rather intellectual process of thinking (analysing), of describing (of synthesising) and of contemplating (of reasoning). Software, as a product, has less material, quantitative measures by which to be grasped (no cheaper, faster, smaller, etc., catchwords) than it has intellectual, qualitative measures — such as affinity to application domain (it is, or is not, the right product), fitness for human use (computer–user interaction), correctness (the product is, or is not, right), etc.

Grasping abstraction — a major issue of these volumes — affords any developer a far better chance of getting the right product and the product right than not grasping abstraction — even when these same people do not use many of the formal techniques of these volumes. Most practicing software engineers do not grasp abstraction. Yet software, by its very nature is and must be abstract: When supporting the automation of what used to be human work processes, the automating software is not "those human processes", it is only a model, an approximation, an abstraction of them.

We wish to perpetrate a view of software development as something that proceeds in phases, stages and steps of development and for which there are now available clear techniques of relating these phases, these stages, these steps to one another. Yet such development is hardly covered in standard textbooks on software engineering. We wish to perpetrate a view of software development where the specification of the phases, stages and steps can be done formally, and where the relations can be formalised and, in cases where warranted, can even be formally verified. This view has been possible, at least in the small to medium, for at least 20 years. Yet such development is hardly covered in standard textbooks on software engineering. We wish to further a view of software development where the developers create, nurture and deploy abstractions. Where the programmers at all levels take pride and have fun in "isolating", as it were, beautiful abstractions and let them find their way into programs. In the end these programmers let those abstractions determine major structures of systems, and beauty: Simplicity and elegance, as felt by users, arises! Such development is scalable to large systems. It is now possible, manageable and affordable. It can be taught and it can be learned by most academically trainable students.

Formal Techniques "Light"

Many practicing programmers abstain from and some academics express reservations about formal reasoning[6] or just formal specification.[7]

Our approach is a pragmatic one. We allow for a spectrum from systematic via rigorous to formal development. By a systematic development we mean one which specifies some of the steps of development formally. By a rigorous development we mean one which expresses and formally proves some of the proof obligations of a systematic development. By a formal development we mean one which formally proves a significant majority of proof obligations as well as other lemmas and theorems of a rigorous development.

In order to follow the principles and techniques of these volumes, we advise going "light": Start by being systematic. Specify crucial facets — of your application domain, your requirements and your software designs — formally. Then program (i.e., code) from there!

It seems, from practice [155], that by far the most significant improvements in correctness of software development accrues from being systematic. And these

[6]Example: Proving, in some mathematical logic, some lemma about program properties.

[7]Example: Describing, in addition to informally, but concisely, some domain, or prescribing some requirements, or specifying some software design formally, in some formal specification language.

volumes are primarily, possibly almost exclusively, focused on being systematic. Certain kinds of applications warrant higher trust, and it then seems that being rigorous achieves the next higher step of believability. Finally, a few customers are willing to accept today's rather high cost of formal development: heart pacemakers, hearing-aid implants, hybrid controllers for nuclear power plants, driverless metro trains, and the like.

Volume 3, Chap. 32, Sect. 32.2 discusses a rather large number of dogmas, misconceptions and myths about so-called "formal methods". Section 1.5.3 of this volume and Vol. 3, Chap. 3, Sect. 3.1 discuss why methods cannot be formal, but that some techniques can.

The "Super Programmer"

Many practicing programmers and some academics believe strongly in the unchecked individualism of the programmer: They are worried that having to adhere to a number of method principles and formal techniques may squash the creativity and productivity of "super programmers". We are not worried. We have generated well over a 100 MSc thesis candidates. Most work in fewer than eight software houses in Denmark. All follow, more-or-less, many of the principles and techniques of these volumes. Most of them are super programmers.

The following has been expressed by other academics and most of my former students and likewise those of my colleagues around the world who similarly teach and propagate principles and techniques like those of these volumes. I emphasise it here:

> The principles and techniques of these volumes, even when adhered to only "lightly", even when hardly followed explicitly, are such that if you have grasped them, while studying these volumes, they will have changed your attitude to software engineering. It will never be the same.

We are sure that you will, from then on, enjoy far more doing "super programming", being a super programmer, and *"being clever in many small ways, devising smart tricks to do things better and faster."* We shall not deny a central role[8] for being low level clever, for being smart. We will augment whatever skills you may have in this direction with a number of teachable engineering principles and techniques. *"The successful programmer is both beast and angel."*

We claim that we can also point to several medium-scale software development projects where knowing or being aware of the principles of these volumes seems to have helped significantly in devising elegant, beautiful products. And *'Beauty is our Business'* [224].

[8]The two *slanted* "quotes" of this paragraph are from an e-mail, Sunday, January 20, 2002, from Prof. Bertrand Meyer [5,375,376], ETH Zürich, Switzerland, and ISE, Santa Barbara, California, USA.

What Is Software Engineering?

We continue the characterisation of software engineering that we began on the very first page of this preface.

- **Software engineering:** To us, in a most general sense, 'software engineering', as are all kinds of engineering, is a set of professions which based on scientific insight construct technologies, or which analyses technologies to ascertain their scientific content (including value), or, usually, do both.
- **"Software Engineer":** Thus the software engineer (but see the following for a critique of this term) *"walks the bridge"* between computer and computing science, on one side, and software artifacts (software technologies), on the other side, and constructs — or studies — the latter based on insight gained from the body of knowledge established in the many disciplines of computer and computing sciences.

In a more mundane way, software engineering embodies general and specific principles, techniques and tools (i) for analysing problems amenable to solution or support through computing; (ii) for synthesising such (program, such as software) solutions; (iii) for doing this analysis and synthesis in large projects, that is, projects involving more than one developer, and/or projects for which the resulting software is to be used by other (people) than the developer(s); and (iv) for managing such projects and products (including planning, budgeting, monitoring and controlling the projects and the products).

But because we can term a subject software engineering does not necessarily mean that we can speak of "software engineers". As formulated above, and this must be understood clearly by all readers of these volumes, software engineering is a body of principles, techniques and tools available to such people as we may otherwise have wished to label "software engineers". But for any one person to be labeled a software engineer without further, more "narrowing" qualifications seems problematic. It would give the "recipient" of the message *that person is a software engineer* the belief that the person in question is able to professionally tackle the development of well nigh any software. With Jackson [307] we claim that there are no software engineers! There are compiler engineers, there are embedded systems (software) engineers, there are information (cum database) systems (software) engineers, there are banking software engineers, and so on, just as we speak of automotive engineers and of electrical power engineers rather than mechanical or electrical engineers.

Thus the principles, techniques and tools of these volumes apply, we claim, across a broad spectrum of specialty software engineers. These volumes bring examples of applications of the principles, techniques and tools across the broadest possible spectrum. The fact that principles, techniques and tools are generally useful and can be deployed across a broad field of occupations and applications only means that the student must also, additionally, study special texts on the chosen profession, compiler development, development of safety-critical real-time software, database systems, etc., to become a proper specialty software engineer.

The Author's Aspirations

So these then were and are my aspirations: To provide you with a different kind of textbook; to bring more than 30 years of exciting programming methodological studies and controlled experimental practice into the larger arena of software engineering; to show you what a beautiful world software development can be when following the didactic cornerstones of linguistics, philosophy, semiotics and mathematics; and to unload more than 25 years of evolving lecture notes into a set of three coherent, consistent and relatively complete volumes.

> I have written these volumes because I wanted to understand how to develop large-scale software systems. When I started, some 25 years ago, writing lecture notes on this subject, I knew less than I do now. Meanwhile I have had the great pleasure of having many clever and eager students follow the practice. I have initiated the large-scale commercial developments of compilers for such unwieldy programming languages as `CHILL` [254,255] and **Ada** [128,129,155], and I have thus honed and corrected my thinking. Writing about software engineering while testing out the ideas has been a sobering experience. There are still many corners of software engineering that I have to write about, think and experience. Meanwhile, this is what you get!

These volumes thus represent my chef d'œuvre.

Role of These Volumes in an SE Education Programme

Who are the target readers of these volumes? That question is indirectly answered in the following.

What roles do we see these volumes serve in the larger context of an academic software engineering education, one that leads to a Master's degree in the subject? Figure 1 shall assist us in answering that question.[9]

[9]The labelled boxes of Fig. 1 designate topics that enter into the software engineer's daily practice, and which are therefore useful topics of learning. In Fig. 1 two-way arrows between boxes indicate that the designated topics can be studied simultaneously. Directed (one-way) arrows between boxes designate a suitable, proposed precedence relation between the learning of these topics. A "fan in" (multiple source) arrow shows that a topic may need (i.e., have as prerequisites) the knowledge of one or more (predecessor) topics. A "fan out" (possibly multiple target) arrow shows that the arrow source topic is a "must" for one or more successor topics.

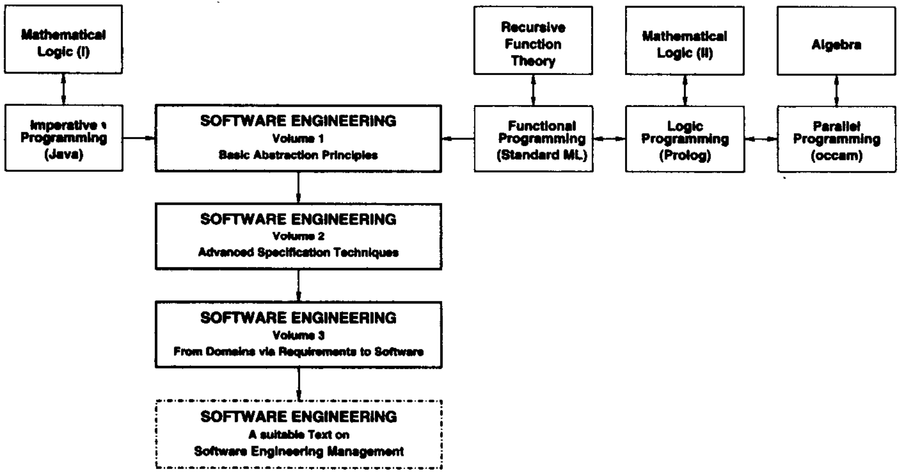

Mathematical Logic (I)
Imperative Programming (Java)
SOFTWARE ENGINEERING Volume 1 Basic Abstraction Principles
Recursive Function Theory
Functional Programming (Standard ML)
Mathematical Logic (II)
Logic Programming (Prolog)
Algebra
Parallel Programming (occam)
SOFTWARE ENGINEERING Volume 2 Advanced Specification Techniques
SOFTWARE ENGINEERING Volume 3 From Domains via Requirements to Software
SOFTWARE ENGINEERING A suitable Text on Software Engineering Management

Fig. 1. Courses based on these volumes: a first setting

We emphasise that we here place these volumes in the context of an academic Software Engineering MSc education programme — not to be confused with an academic Computer Science MSc education. The former aims at the production of industry programmers: developers of commercial software. The latter aims at theoreticians, useful in an academic institution of study. Another explanation, wrt. another diagram, would thus have to be given for an equally likely setting in the context of an academic programme for an MSc degree in (theoretical) computer science, and yet another one for an undergraduate course of an academic software engineering BSc education programme.

- **Prerequisite or "concurrent" courses:** We assume that the reader of these volumes is — or while following a course based on Vol. 1 of these volumes becomes — familiar with the general topics of imperative, functional, logic, parallel and machine programming. Teaching in these topics must cover both skill-learning and training wrt. specific languages such as, for example, SML (Standard ML) [261,389] for functional programming, Prolog [295,351] for logic programming, Modula-3 [262,401], Oberon [527] and Java [10,20,243,348,470,511] for modular (i.e., object-oriented) programming, and occam [364] and a machine language for some well-chosen, "current-technology" hardware (e.g., Intel-like) chip. Their teaching must also cover — to a basic extent — the knowledge acquisition wrt. the theoretical background for these programming styles and languages: recursive function theory [136,444], logic for logic programming [295,351], Hoare Logic for imperative programming [15,16] and process algebras for concurrency (CSP [288,289,448,456] and Petri nets [313,421,435–437]). The machine programming topic [379,501,511] is the only real hardware-oriented, but not hardware-design-oriented [279,418], course. Codesign [482], that is, design of combined hardware/software systems (typical, for example, for embedded systems, see below) is not covered. But one could "add other boxes"! Included in the above kinds of course, or additional to these, we expect the reader to have some working knowledge of algorithms and data structures, i.e., to be familiar with the classical as well as modern such algorithms and data structures and measures of concrete complexity [7,357,371,495,524].

- **Auxiliaries:** The reader is assumed to be — or to become, in conjunction with the software engineering study of which these volumes are part — comfortable with mathematics — to a Bachelor's degree level in the subjects listed. We suggest [534], a delightful "smallish" introduction, and the substantial introduction to discrete mathematics [213]. We find [213] to be an excellent textbook for an entirely separate, and major, course on that topic. One that every software engineer is assumed to take.

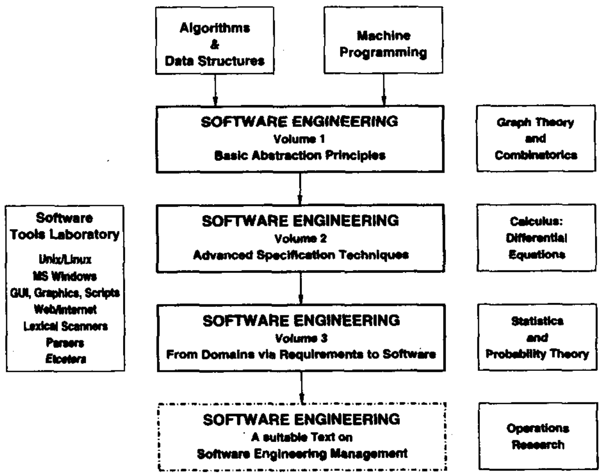

Algorithms & Data Structures
Machine Programming
SOFTWARE ENGINEERING Volume 1 Basic Abstraction Principles
Graph Theory and Combinatorics
Software Tools Laboratory
Unix/Linux
MS Windows
GUI, Graphics, Scripts
Web/Internet
Lexical Scanners
Parsers
Etcetera
SOFTWARE ENGINEERING Volume 2 Advanced Specification Techniques
Calculus: Differential Equations
SOFTWARE ENGINEERING Volume 3 From Domains via Requirements to Software
Statistics and Probability Theory
SOFTWARE ENGINEERING A suitable Text on Software Engineering Management
Operations Research

Fig. 2. Courses based on these volumes: a second setting

Similarly, but more thought of as part of term projects and other forms of laboratory (including self-study) work, we expect the reader to be reasonably comfortable with practical, existing platform technologies (the Software Tools Laboratory box).

- **Main course:** These volumes are then to serve in a main set of three courses on software engineering — and before the breadth and depth of the follow-on courses are attempted. We additionally would advise acquisition of the two books [236, 238], the first as supplementary, the second to fill out especially the verification (i.e., the design calculi) parts which are not developed in these volumes.