

Introduction

- The **prerequisites** for studying this chapter are that you have academic training in programming, that is, in algorithms and data structures, say using two or more of the **Standard ML**, **Java** and **Prolog** programming languages.
- The **aims** are to set the stage for the entire set of volumes, to introduce the “triptych” concept of domain engineering, requirements engineering and software design, to emphasize the importance of documentation and of descriptions, to preview the concepts of formal techniques, methods and methodology, and to introduce the concepts of syntax, semantics and pragmatics.
- The **objective** is to guide you in the direction of what we think are to be the important aspects of software engineering; that is, to set, with respect to the aims and objectives of this book, your “spinal chord” to as close as possible a “state” as that of their author.
- The **treatment** is informal and discursive.

This chapter has been written so as to be read, if not in excruciating detail, then at least such that the reader is hopefully “tuned” to somewhere near the “wavelength” of the author of this chapter. The present chapter may thus be read in between the study of most subsequent chapters.

1.1 Setting the Stage

Characterisation. *Engineering* is the mathematics, the profession, the discipline, the craft and the art of turning scientific insight and human needs into technological products. ■

The sciences of software engineering are those of computers and computing.

Characterisation. *Computer science* is the study and knowledge of what kind of “things” may (or can) exist “inside” computers, that is, *data* (i.e., *values* and their *types*) and *processes*, and hence their *functions*, *events* and *communication*. ■

Characterisation. *Computing science* is the study and knowledge of how to construct those “things”. ■

These volumes will provide material for teaching you some of the core aspects of the mathematics, the profession, the discipline, the craft and the art of software engineering. The engineer walks the bridge between science and technology, creating technology from scientific results, and analysing technology to ascertain whether it possesses scientific values. These volumes will teach you some of the science of computing, exemplify current software technologies, and help you to become a professional engineer “walking that bridge”!

Students of these volumes are not expected to have any acquaintance with the disciplines in the following list of computer science topics: automata, formal languages and computability [296,319], programming language semantics [183,252,443,454,497,521], type theory [1,241,407], complexity theory [319], cryptography [363], and others as covered in, for example, [344]. The topics of the above list, other than the first, will either be introduced in these volumes or can be studied after having studied the present text.

Students of these volumes are expected to possess some fluency in the following computing science topics: functional programming [261], logic programming [295,351], imperative programming [20,243,290], parallel programming [449], and algorithms and data structures [7,161,326–328].

The keywords *art* [326–328], *discipline* [194], *craft* [441], *science* [245], *logic* [275], and *practice* [276], are also prefix terms of the titles of seminal textbooks on programming, as referenced. In a sense these references also serve to indicate our basic approach to programming. But software engineering goes beyond what has been implied by the above listings of computer and computing science topics. Software engineering goes beyond the algorithm and data structure, cum programming language skills. These computer and computing science skills can and must first be reasonably mastered by the individual, by the professional, academically educated and trained programmer. Software engineering is as much about making groups of two or more programmers work productively together.¹ And software engineering is about producing software which can be further deployed in the development of larger computing systems by other developers.

To fulfill these latter aspirations, software engineering must augment the knowledge of computer and computing sciences with such disciplines as project and product management. By *project management* we colloquially mean: How do project leaders plan (schedule and allocate) development resources, how

¹However, the principles, techniques and tools covered by these volumes are also required to be used even by the “lone” programmer developing her “own” software.

do they monitor and control “progress”, and so on? By *product management* we colloquially mean: How does a software house determine a, or its, product strategy and tactics, that is, which projects to undertake, which products to market, how to price, service and extend them, and so on?

We detail a number of project management issues: (1) choice and planning of development process, (2) scheduling and allocation of resources, (3) monitoring and control of work progress, (4) monitoring and control of quality: assurance and assessment, (5) version control and configuration management, (6) legacy systems, (7) cost estimation, (8) legal issues, etc. There are other issues, but listing just these shows, up here, early in these volumes, the large variety of development concerns.

(1) *Process (choice and) modelling* is a project management issue. How do the engineers proceed, what does one do first, then after that, etc.? There is not just one right way of doing things, of proceeding in phases, stages and steps, rather there are many eligible process models. First, the development process is determined by the problem frame; second, by the novelty of the problem; third, by the experience of the programmers and of management; and so forth.

(2) *Planning, scheduling and allocation of resources* is another project management issue. In planning we decide on which things to do. In scheduling we decide on when to do these things, and in allocation we decide on which resources (monies, people, machines, etc.) to deploy.

(3) *Monitoring and control of work progress* extends the list of project management concerns. Once the project proper starts, after planning, one needs to regularly and continuously check what has been achieved. And, if what has been achieved is according to plan, then just continue. But if plans are not being followed, then control must be asserted by possibly changing the plan, rescheduling and/or reallocating development resources.

(4) *Monitoring and control of quality assurance and assessment* further extends our project management concern list. The web of application domain knowledge that goes into a software product, the maze of hundreds of mostly unrelated requirements that are expected fulfilled from the software product and the “Babylonian towers” of software design techniques and tools (languages, etc.) all necessitate careful formulations of what is meant by product quality, as well as close scrutiny of the development process, in order to ascertain whether quality objectives are at risk or are being met.

(5) *Version control & configuration management*: In the development of software the programmers usually construct several versions, or “generations”, of code. One must monitor and control these generations and versions. This is called *version control*. It can be a sizable undertaking when, as is often the case, there exist hundreds, if not up towards thousands, of such alternative and complementary versions. Some of these versions may enter into one release of a product, while other subsets of versions enter into other releases of related products. Combining such versions into software products is called *configuration management*.

(6) *Legacy systems*: At any time customers (users, acquirers, buyers) of software operate computing systems composed from often “age-old” parts, and these have to be maintained: *adapted* to new hardware and to new software, *perfected* to offer relevant performance, and *corrected* (by removing “bugs”). All three maintenance aspects become increasingly problematic as the original software is either programmed in languages for which there are no longer adequate, let alone “recent” compilers and related support tools, or is documented in a style basically unfamiliar to new generations of programmers, or not documented at all. This kind of software and these kinds of problems constitute the concept of *legacy software*.

(7) *Cost estimation*: Two issues of cost estimation may be relevant: estimating the cost of developing new (or maintaining old) software, and estimating competitive, profitable prices for software. The problem of *cost estimation* is intertwined with the problems of software development process models, project and product management, quality assurance, version control and configuration management, legacy systems, etc.

(8) *Legal issues related to software*: There are many legal issues related to software. There are software patents, which establish intellectual, and property rights. There is *software curriculum accreditation*, that is, the approval of a university or college curriculum in software engineering. And there is *software house accreditation*: the approval (usually, typically by, or through some ISO-related agency), generally, of a software house as a trustworthy developer of software. There is *software engineer certification*: the approval (usually by some national engineering society) of a person being a bona fide professional. Finally there is *software product certification*: the approval (usually by some international agency, such as Lloyd's Register of Shipping, Bureaux Veritas, Norwegian Veritas, TÜV, or others) of a specific software product to meet certain standards of quality.

• • •

Software engineering is anchored in programming: (1) in the design of software, (2) before that in constructing the software requirements, (3) and before that in understanding the application domain.

These volumes spend most of their pages on the development aspects of software engineering: on principles and techniques for developing proper application *domain* understandings, on principles and techniques for developing proper software *requirements* and on principles and techniques for developing proper *software designs*. These volumes unfold these principles and techniques based on the tools of both informal and formal languages for *describing domains*, *prescribing requirements* and *specifying (designing) software*.

1.2 A Software Engineering Triptych

It is a definite new contribution of Vol. 3 that it focuses, in a “special way”, on the triptych² of *domain engineering*, *requirements engineering* and *software design*. That way emphasises that domain engineering, “*ideally and logically speaking*”, precedes requirements engineering, which (and there is nothing new in this), *ideally and logically speaking*, precedes software design. The new contribution is the central role given to domain engineering.

1.2.1 Software Versus Systems Development

Although these volumes are primarily about the engineering of software, we cannot avoid getting involved, to a nontrivial degree, in the more general engineering of computing systems.

Characterisation. By a *computing system* we mean a combination of hardware and software that together implement some requirements. ■

Typically a computing system is distributed, over local areas as well as globally, and thus very typically requires extensive data communication hardware and software. When, in the following, we say ‘software’ or ‘system’ we can usually substitute the more general term ‘computing system’.

1.2.2 Motivating the Triptych

We motivate the roles of the three triptych constituents as follows: *Before we can (3) design software we must understand the (2) requirements put to this software. And before we can prescribe the (2) requirements we must understand the application (1) domain.* What is discussed, again and again in these volumes, is how we interpret the “*ideal and logical*” precedences mentioned above. But first we will take a look at the three triptych components, or, as we shall also refer to them in these volumes, *the three phases of software development*.

1.2.3 Domain Engineering

Characterisation. By *domain engineering* we mean the engineering of domain descriptions. ■

²*Triptych*: (i) From Greek ‘triptychos’, having three folds, (ii) an ancient Roman writing tablet with three waxed leaves hinged together, (iii) a picture (as an altar-piece) or carving in three panels side by side, (iv) something composed or presented in three parts or sections. Same as trilogy.

Characterisation. By a *domain* we mean (i) an area of human activity, (ii) and/or an area of semi- or fully mechanised activity, (iii) and/or an area of nature that can be described, and parts or all of which that can potentially be subject to partial or total computerisation. ■

Example 1.1 Three Domains: Examples of (respective) domains, related to the above enumeration (i–iii), are: (i) book-keeping; (ii) the sending of freight from a harbour of origin, on ships via other harbours, to a destination harbour; and (iii) the planetary movements, i.e., celestial mechanics [494]. ■

We understand a domain when we can describe it in an objective way.

Characterisation. By a *domain description* we mean an indicatively expressed description of the properties of the following domain facets: the *in-trinsics* (the basic, invariant, and core), the *enterprise* (business, institution) *processes*, the *technology supports*, the *management and organisation*, the *rules and regulation*, the *human behaviour*, and possibly of other facets of the domain. ■

Domain descriptions explain the domain *as it is*. No reference can be made to any requirements to desired software — that comes later! Furthermore, no reference can be made to the desired software — that also comes later! So, a domain description really has nothing to do with information technology (IT) or software — other than what is already installed and deployed in the domain, and then only if reference to such existing IT and software is deemed relevant.

Example 1.2 A Logistics Domain: We are not describing the example domain, only informing about it, but in almost descriptive terms: A logistics domain consists (a) of senders and receivers of freight; (b) of logistics firms which arrange for senders and receivers to send or, respectively, receive freight; (c) of hubs (like harbours, railway stations, truck terminals and airport air cargo centres) where freight may be loaded onto or, respectively, unloaded from conveyors; (d) of conveyors (such as ships, freight trains, trucks, respectively air planes) that are owned and/or operated by transport companies; (e) of transport companies (like cargo liners, railway operators, trucking companies, airlines); and (f) of the networks of transport routes (shipping lanes, railway lines, highways or, respectively, air corridors).

Some further descriptions can be hinted at: A conveyor path³ is a connection between two hubs. A conveyor route is a sequence of one or more connected paths. Some hubs are of two or more kinds, viz., harbours and railway stations, air cargo centres and truck terminals, etc. Conveyors travel their routes according to fixed time tables. A conveyor fee table prescribes costs of transporting freight, per cubic meter, between hubs. This example is

continued in Example 1.3. Notice that there were no references to either requirements or to possibly desired software (i.e., computing system), let alone to such a system. ■

A domain description, to repeat, describes the domain as it is. Chapter 5 of Vol. 3 covers principles, techniques and tools for describing any *universe of discourse*, whether domain, requirements or software. Part IV (Chaps. 8–16) of Vol. 3 covers principles, techniques and tools for proper domain description. Domain knowledge need be acquired, that is, elicited from those who work in and are affected by the domain.

1.2.4 Requirements Engineering

Characterisation. By *requirements engineering* we mean the engineering of *requirements prescriptions*. ■

Requirements arise as a natural consequence of a *contractual relation* between a *client* who procures (who is to acquire) some desired software (i.e., software to be delivered), and the *deliverer* or the *developer* of that software. By *requirements* we mean a list of one or more putatively expressed statements as to which *properties* are expected from the software to be developed. Requirements must be acquired, that is, elicited from those who may be affected by the eventually acquired software.

Example 1.3 *Some Logistics Requirements:* This example continues Example 1.2. We do not exemplify a proper requirements prescription, we just hint at what it might deal with. A logistics system needs software support for (at least) the following kinds of activities:

First we exemplify some *domain requirements*. These are requirements that solely pertain to the domain, and whose professional terms are domain terms. Examples are: Software support for handling inquiries, from potential senders, with logistics firms, as to possible routing of freight, schedules and costs; software support for handling requests, from actual senders, to logistics firms, for the dispatch of freight, and hence the issuance of bills of lading (waybills) and the handling (passing on) of freight to be sent; software support for logistics firms tracing the whereabouts of freight at hubs or with the owner transport companies of scheduled conveyors; software support for the hub management of conveyors in and out of hubs, the unloading and loading of conveyors, and the receipt of freight from, and delivery of freight to logistics firms.

Then we exemplify some *machine requirements*. These are requirements that primarily pertain to the machine to be built, that is: the software+hardware of the desired computing system, in other words, whose professional terms additionally include information technology terms in general.

³Examples of paths: Sea lanes, rail lines, roads, and air corridors.

Examples are: The computing system shall have a mean time between failures of two years; when the system is “down” it must at most be so for two hours, and so on.

Finally, we exemplify some *interface requirements*. These are requirements that pertain both to the domain and to the machine to be built, to the interface between the machine and the domain, human users of the domain as well as (other) natural phenomena and man-made equipment of the domain. Interface requirements are about the phenomena that are shared between the domain and the machine. Examples are: senders and receivers shall be able to ascertain the transport status of their own freight from their own, home PCs based on standard Internet browsers; the computing system shall display, for logistics firms, the route networks in some “zoom-able” manners, and so on.

This example is continued in Example 1.4. ■

Notice how Example 1.3 introduced three notions of requirements: domain requirements, interface requirements and machine requirements.

This decomposition represents a pragmatic separation of concerns. Domain requirements, to repeat, are requirements that pertain solely to domain phenomena, i.e., they are requirements whose professional terms are domain terms. Interface requirements, to repeat, are requirements that pertain both to the domain and to the machine to be built, to the interface between the machine and the domain, human users of the domain as well as (other) natural phenomena and man-made equipment of the domain. That is, to phenomena shared between the environment and the machine. Machine requirements, to repeat, are requirements that primarily pertain to the machine to be built, that is, the software + hardware of the desired computing system. In other words, the professional terms of machine requirements additionally include information technology terms in general.

Notice how we, in rough sketching some requirements, relied on domain terms having been previously described. We did, however, not precisely describe those terms. But we hinted at how it is the purpose of a domain description to explicate all such domain specific terms. We likewise relied on machine (hardware + software technology, that is: IT) terms also having been precisely specified, elsewhere!

Notice also how we “sneaked” the crucial concepts of *domain*, *interface* and *machine requirements* into the example! Part V (Chaps. 17–24) of Vol. 3 covers principles, techniques and tools for the proper prescription of requirements.

A popular view of requirements makes the following distinctions: user requirements, system requirements, and non-functional requirements. How are we to take these? User requirements form one entire set of requirements: domain, interface and machine requirements. So do system requirements. Non-functional requirements are what we refer to as some interface and most, if not all machine requirements. How does this work? User requirements do not need to be complete, they can be, as we shall call them, rough-sketches, although they are typically well-structured and carefully cross-referenced, and

they form input for the development of system requirements. System requirements must be consistent and relatively complete: they “improve” upon the user requirements, and they form input to software design.

1.2.5 Software Design

Software: Code and Documents

Characterisation. By *software* we mean not only the *code* based on which computers can act, but also all the *documentation* that is necessary for the proper deployment of the code. This includes the *business process reengineering manuals* that are necessary for the enterprise (the institution) acquiring the computing system to function most optimally when using this system, the *installation manuals* that are necessary when initially installing the computing system, the *user training and daily use manuals* that are needed in preparatory training of future system users as well as in their daily use of the system as installed, the *maintenance manuals* that are needed during the daily facilities management of the installed system (for (adaptive) up- or downgrades, for performance (perfective) enhancements, and for error corrections), and the *disposal manuals* that are needed when dismantling the system. Ideally software also includes a precise record of the software validation and verification history: stakeholder responses, verification and tests, including test suites and the results expected from, and actually recorded during, actual tests using these test suites. By a *test suite* we mean a collection of data serving as input to a test. ■

Software Design, I

Characterisation. By *software design* we mean the implementation of (required) software, not just coding, but its stage and stepwise development and documentation. ■

Phases, Stages and Steps of Development

Characterisation. By *software development* we mean the combined development of domain descriptions, requirements prescriptions, and software designs. ■

Software, as well as domain descriptions and requirements prescriptions, is usually rather complex. Hence these need be developed according to the principle of *separation of concerns*, i.e., of *divide and conquer*. Therefore we divide the development phases of domain descriptions, requirements prescriptions and software design into stages and steps. A first development, one that is reasonably illustrative of a multistep development, is given in Examples 16.10 to 16.21. Part VI (Chaps. 25–30) of Vol. 3 covers software design.

Software Design, II

Conventionally we think of establishing, in *stages of software design*, first the *software architecture*,⁴ which in a sense explained, in Chap. 26 of Vol. 3, implements a “high-level design” of the domain requirements, the interface requirements and the machine requirements. In the second stage we establish the *program components* which in a sense, explained in Chaps. 27 and 28 of Vol. 3, designs the gross and detailed modular structure of the software. The final or *implementation stage*, which usually consists of many steps, includes *platform reuse design* in which available *software components* are examined for their possible reuse in the implementation, *modularisation* or *objectivisation*, in which a fine grained decomposition of the program organisation into modules takes place, and finally the *coding* itself in which final lines of code are specified. That is, the instructions to the computer as expressed in some programming languages and in calls to run-time system facilities and (other platform) components.

In Example 1.4 we give an informally expressed software architecture design.

Example 1.4 *A Logistics System Software Design:* This example continues Examples 1.2 and 1.3. We do not exemplify a proper software design specification. We just hint at what it might deal with. A logistics computing and communication system is implemented as follows: Each sender or receiver, each logistics firm, each transport firm, each hub and each conveyor (of a transport firm) is implemented as a separate, concurrently operating process with its own state. None of the processes share global state components, but instead operate based on synchronised and communicated messages. Freights are not implemented as objects, i.e., as independent processes. Shared data is implemented as a separate process whose state represents the shared data (i.e., a database). ■

1.2.6 Discussion

General Issues

This ends our exposition of core concepts of the software development triptych. In summary we emphasise two sets of relations between the three software development phases. The three kinds (cum phases) of engineering development can be summarised as follows: In domain engineering we describe the domain as *it is*. In requirements engineering we prescribe the requirements to software (i.e., a computing system) for the support of activities in the domain as we

⁴Wherever we say software architecture we could say computing systems architecture.

would like to have them. In (the early stages of) software design we specify the software such as we *have decided it shall be*.

The relations between the three kinds of documents arise from respective development phases. Domain descriptions are *indicative* [308], as we seriously believe the domain essentially is. We must make sure to describe all possible behaviours of the domain, including as we normally expect well-functioning actors to perform, but to also include erroneous, faulty, less diligent, sloppy, or even outright criminal behaviours. Requirements prescriptions are *putative* [308], as we would mandate the software to behave. A requirements prescription would naturally focus on well-functioning behaviour and try to ensure correct behaviour of all actors, whether men or machines. Software specifications are *imperative* [308], that is, mandatory.

When a domain description is formalised, the hedge ‘may’ is lost. And when a requirements prescription is formalised, the hedge ‘must’ is likewise lost. Formal domain descriptions, requirements prescriptions and software (design) specifications have in common a certain “authoritative air” which the domain description can never have. A domain description is only an abstraction, or a model of some reality, but it is not that reality, whereas a requirements prescription is intended to be a precise exact model of the software to be implemented.

The triptych approach to software engineering is central to these volumes. We shall endeavour to enunciate clear principles, techniques and tools for the development of domain descriptions, requirements prescriptions and software specifications. Within domain descriptions we find such concepts as domain attributes, stakeholders and their perspectives, and domain facets. Within requirements prescriptions we find such concepts as domain requirements, interface requirements, and machine requirements. Independently of these we find such requirements techniques as domain projection, instantiation, extension and initialisation. Within software design we find such concepts as software architecture, program organisation and structure, and modularisation.

1.3 Documentation

This section is a precursor for a later chapter, Chap. 2 of Vol. 3, which includes many examples and enunciates many documentation principles, techniques and tools. Since documentation is all pervasive and is all important in software engineering, we shall this early in these volumes “lift the curtain” on documents enough that we can refer broadly and generally to the document types in the text that follows between this section and Chap. 2 of Vol. 3 in which we finally dispose of the subject.

We saw, in the previous section, that software development entails three major phases, possibly several stages within phases and possibly several steps within stages. Carrying out each of the steps results in documents. These are

documents on domains descriptions, requirements prescriptions and software specifications.

There is nothing else⁵ emanating from steps, stages and phases than documents, on paper or electronically. So the question is: What kind of documents? In this section we will briefly overview three kinds of documents that result from the engineering of the steps, stages and phases. It is important that the reader keeps the *universe of discourse* in mind, either the domain, the requirements, the software, the two first (domain and requirements), the two last (requirements and software) or all three (an entire development). That is, the various documents, even the informative ones, all have a specific *universe of discourse* in mind. It must first be clearly stated, lest one of the “parties” to a development contract gets confused from the very start!

1.3.1 Document Kinds

There are basically three kinds of documents that emerge from the development process, and which the developer hence should be aiming at. These are: (1) *informative documents*, or document parts, such as partners and current situation, needs and ideas, product concepts and facilities, scope and span delineations, assumptions and dependencies, implicit/derivative goals, synopsis, design briefs, contracts, logbook; (2) the *description documents*, or document parts, such as rough sketches (records of “brainstorming”), terminologies, narratives, and formal models; and finally (3) the *analytic documents*, or document parts, such as description property verifications, verification of correctness of development transition (i.e., development step), and validation of formal and informal descriptions.

We will briefly review these kinds of documents, both as concerns their pragmatics: why they are necessary, and as concerns their multitude: why there are so many seemingly different kinds of documents.

1.3.2 Phase, Stage and Step Documents

A development phase results in a comprehensive, definitive set of informative, descriptive and analytic documents. A development stage results, similarly, in a comprehensive set of informative, descriptive and analytic documents, or in a set of relatively complete domain, interface or machine requirements prescriptions.

The boundaries between a subphase and a stage, and the comprehensiveness of either, are not sharp. It serves no purpose here, or for the approaches advocated in these volumes, to try sharpen such distinctions. The stage and

⁵Strictly speaking: Understanding also emerges, and so do closer relations between client (acquirer, customer) and developer (deliverer, provider), etcetera. But, contractwise, unless, for example, education and training is also part of a project, documents are the only tangible goods delivered!

step concepts are simply pragmatic. One could go on defining sub-steps, etc., but we refrain. Let the actual project determine a need for finer granularities!

If a distinction need be made between a phase and a stage, then the comprehensive set of stage documents represents one of more than one “stage” of development within the phase.

A step of development produces only a part of a comprehensive set of documents, for example: a comprehensive set of informative, descriptive or analytic documents or document parts, or just, as a substep, one of these documents, or document parts. More will emerge as we progress deeper into these volumes.

1.3.3 Informative Documents

Characterisation. By an *informative document* we mean a document, or a document part, which informs, it does not necessarily describe a designatable, manifest phenomena or concept. ■

As the name implies, informative documents give information which takes many forms. Informative documents include those of perceived or already enunciated needs, product concepts and facilities, scope and span delineations, assumptions and dependencies, implicit/derivative goals, synopsis, contracts, design briefs, and so on.

Current Situation Documentation

Need for software development, or for requirements prescription, or for domain description usually arise out of a *current situation*. A current situation may be that the domain is not well-understood, or that software is required. Professional software development projects therefore produce an informative document — two–three pages — which inform of the current situation that leads to needs.

Needs Documentation

Needs refer to perceived or actual needs for the product being desired, whether a domain description, a requirements prescription, a software design (i.e., specification), or just plainly, as is most often the case, the software itself. Needs can be expressed in many ways: We must understand the domain; we must establish requirements; “*So ein Ding muss Ich auch haben*”⁶; software to automate humanly menial, boring processes; software to speed up slow processes; and so on. Needs must be quantified, if possible.

⁶“I must also have such a ‘thing’” (i.e., software).

Product Concepts and Facilities

Product concepts and facilities refer to “brainstorming” or ideas (“dreams”). That is, what the universe of discourse “contains”, or is to contain, what aims and objectives the proposers have for the “product”, what roles, in a larger socioeconomic context, the product is to serve (or fulfill). That is, what are the strategic or tactical objectives of the developer and/or customers, how it might complement earlier products, and/or how it might open the way for, or be, a next-generation product.

Design Briefs

Design briefs refer to documents which state what kind of project is to take place: for which universe of discourse, specifically (aiming at a very specific client), or generally (aiming at a largest class of such clients), or something in-between. Whether the project is an ordinary development, or a research, or some advanced project encompassing both R&D. Finally it also encompasses what general deliveries are expected, the time frame, costs, institutions involved, and so on.

Usually a scope and span delineation is part of or strictly adjoins the design brief. To this we turn next.

Scope and Span Delineations

Scope and span delineations refer to the more specific subjects of the universe of discourse to be dealt with in the project, that is, the target and modal scope, for example: *railways*, or *health care*, or *financial services*; respectively new development (incl. R&D), or maintenance, or other. The target and modal span, for example, *rolling stock monitoring and control*, or *electronic patient journals*, or *stock trading*; respectively off-the-shelf commercial, one-of-a-kind, or other product.

Synopsis

Synopsis refer to a “capsule” (i.e., short overview) characterisation of the product being desired, whether a domain description, a requirements prescription or a software design. A synopsis is like a movie “trailer”. It tells, in a few words, what the whole thing (domain, requirements or software) is all about. A synopsis is not a description (a prescription, a specification), “but almost”. It mentions all the most important phenomena of the universe of discourse, their entities, types, values, actions, events and behaviours. It mentions their semantics and syntax, but it does so incompletely. And a synopsis “links” these phenomena components to their pragmatics, that is what role they serve, and so on.

Synopses often form an important introductory part of contracts.

Contracts

A contract describes *parties to the contract, the subject matter and considerations*.

Contracts refer to the legal documents that name contractors (the parties: clients and developers); and that define what is to be developed: If software, then the contract would normally refer to an already existing requirements prescription; if requirements, then the contract would normally refer to an already existing domain description; or if a domain description then the scope and span delineation would be an important document part. In addition (the considerations) contracts prescribe the development costs (estimates): If software is to be developed, then the estimate should be rather binding. If requirements are to be developed, then costs could be based on fixed hourly rates and some usually negotiable rough time estimates. Precise numbers cannot be given since much, unforeseeable interaction needs to take place between the contracting parties. Or if a domain description is to be developed—in which case the project is basically a joint research effort—then the costs are usually negotiable, and billed on a, say, monthly basis. A contract would (further considerations) refer to legal conditions. Many other considerations may be part of a contract document.

Discussion

We have outlined essential informative documents. We emphasise that the developer (and/or client) may, in the extreme, have to “repeat” such documents for each phase, stage and, in a few cases, step of development and their transitions. That is, informative documents may be needed for each and all of the triptych phases: domains, requirements and software design.

We have chosen the wording *documents* (and documentation) so as to indicate that one may view each of the listed informative document types as designating instantiation of individual, separately “bound” documents. For the next category of documents, the descriptive ones, we choose a wording that allow their various types to designate document parts that can be “mingled” (woven together) into larger documents.

1.3.4 Descriptive Documents

Characterisation. By a *descriptive document* we mean a document, or a document part, which describes a manifest phenomenon or a concept. ■

The term describe, and hence the terms description, and descriptive, are here used in a rather specific, narrow sense. A description designates (i.e., is some text that sets forth, in words) either some physically existing part of nature (one that centres around physical behaviours usually governed by laws of physics) or some man-made part of the world (one that centres around human

activities, including their interaction with artifacts) or some combination of these two classes of worlds.

Thus a description, such as we shall deploy the term, tends to focus on what might eventually “fit within a computer”. It may well be that what we describe concerning a domain is not computable and cannot be “mimicked” by a computer. A requirements prescription, however, “cuts down” on its underlying domain description and makes sure that what is required is also computable. Hence opinions, emotions, metaphysical, political or such other similar subjective texts are not here considered descriptions.

It can be seen from the above, and it will reappear, again and again later, that it is not a simple, straightforward matter to delineate precisely when something is a description (a prescription, a specification), and what can be described, that is, what can exist. Chapters 5, 6 and 7 of Vol. 3 focus on principles and techniques for forming proper descriptions (specifications) and touch on the philosophical issues of being.

We (thus) consider three kinds of descriptions: domain descriptions, requirements prescriptions, and software designs. We point out that we use three different terms synonymously: descriptions, prescriptions and designs (specifications). Domain descriptions are about what already exists, “the world as it is”.⁷ Michael Jackson [308] refers to domain descriptions as indicative. Requirements prescriptions are about what we expect from software, “the world as we would like it to be”. Michael Jackson [308] refers to requirements prescriptions as putative. Software (design) specifications then outline the design structure of software, that is, specifications of specific types, values, functions, events and behaviours. Michael Jackson [308] refers to domain descriptions as imperative.

Descriptive Document Kinds and Types

We see basically two kinds of description documents: informal and formal. And we see basically four types of description documents: rough sketches (documents which record results of “brainstorming”), terminologi.e., narratives and formal models. One could consider the latter two types (narratives and formal models) to stand for one type, the type of ‘proper description documents’, both informal and formal. We shall stick with the above compartmentalisation.

Rough Sketches

Characterisation. By a *rough sketch document* we mean a descriptive document which is a draft and whose description is incomplete, and/or is not well structured. ■

⁷From an epistemological point of view we may have to say: a world as we subjectively observe it.

When we first, as an initial act of proper development, attempt to develop something, we then “brainstorm”. Recording the ideas that arose during “brainstorming” results in a rough sketch. We are told either to develop a domain description or a requirements prescription or a software design. And we are not quite sure where to begin in the chosen universe of discourse. So we “doodle”, or we rough sketch. A rough sketch is basically an unstructured nonsystematic effort at describing whatever has to be described (prescribed, specified).

A rough sketch serves the purpose — in the style of explorative, experimental work — of coming to grips with the concepts that are central to the universe of discourse, and from there with the derivative concepts. A rough sketch shall then serve, as it is being developed, i.e., as a means to identify the core concepts, and their relations. This identification process is of utmost importance. It is of analytic nature, and is further discussed in Section 1.3.5. Section 2.5.1 of Vol. 3 presents examples, principles and techniques of rough sketching.

Terminology

Characterisation. By a *terminology document* we mean a description document which, in a systematic, but not necessarily a complete or exhaustive manner, lists and briefly explains a number of terms. ■

The rough sketch descriptive step together with the concept formation analytic step serves to identify and consolidate the important concepts (i.e., abstractions of phenomena, whether in domains, requirements or software). This identification contains an element of naming these concepts. A list of all these concept names and their characterisation (description, explanation, definition) is what call a *terminology*. We could also call the list a *glossary* or a *dictionary* or even an *ontology*. We refer to Sect. B.1 for discussions of these four and the related terms of encyclopedia and thesaurus.

We consider it to be a very important and indispensable part of every phase of software development to perform the following four terminology-related actions: (1) to *establish* a (phase-oriented) terminology; (2) to *use* and hence *adhere* to such a terminology; (3) to *update*, i.e., *maintain* such terminologies and let changes be reflected back in all the documents where referenced terms are used; (4) and to *make available* such terminologies.

Failure to do as advised above usually has dire consequences.

Section 2.5.2 of Vol. 3 will present examples, principles and techniques for creating a terminology.

Narrative

Characterisation. By a *narrative document* we mean a description document which systematically and reasonably comprehensively, in natural, yet

most likely (application domain-specific) professional language, explains the entities, functions and behaviours (including events) of a designated universe of discourse. ■

To narrate is to “tell a story”. The story (the narration) to be told here is that of the chosen universe of discourse, be it a domain, or part of a domain, a requirement, or a software design. The narrative must be such that the listener (i.e., the reader) as well as, of course, the narrator, can formalise the story: That is, we put down as a constraint upon the narratives that they can be given mathematical, i.e., computing science, models or otherwise be characterised mathematically. It is not a constraint on domain descriptions that what is described is computable: that it can be “mimicked” (mechanised, simulated) by a computer. It is indeed a constraint on domain requirements prescriptions as well as on software design specifications that they constitute computational models.

This insistence on formalisation can be justified as follows: The domain requirements must imply something computable. After all, they are about a computing system. The software design certainly must also imply something computable.

But why insist on the domain description being formalisable? First, we must accept that domain requirements, as mentioned in Example 1.3, are derived from domain descriptions, and we would like the derivation operations to be formally well understood. Second, we must accept that the original role, as well as the successful pursuit of this role over the last two and a half millennia, has been to formalise phenomena of the actual world, first the physical ones, and now the human-made ones. So why not also attempt this for domains — essential parts of which cannot be said to be understood unless we indeed have a formal model. Third, it must be understood that we shall only attempt to formalise the semantic and the syntactic aspects of domains, not their pragmatic imports.⁸ Finally, we must accept that we today, November 2, 2005, do not quite know how to formalise all aspects of domains and requirements! That last caveat applies in particular to domain descriptions and to interface and machine requirements prescriptions.

Thus the task is clear: describe, principally, what can or what ought be formalised. The style of the informal narrative follows from this dogma: Present first text on the classes of entities (i.e., types: abstract type (sorts) and concrete types). Then postulate any fixed, i.e., constant, instantiations (i.e., values), if and when needed. Then postulate all the functions that apply to entities (i.e., observers, generators, predicates, auxiliaries), and characterise these functions: Start by stating to which types of entities they apply (the input) and the type of the resulting, the yielded (the output) entity; then characterise the functional relationship between inputs and outputs. Similarly identify the

⁸For a discourse on pragmatics, semantics and syntax we refer to later material in Sect. 1.6.2 and in Part IV (Chaps. 6–9 inclusive) of Vol. 2.

behaviours (i.e., processes); and their interaction (i.e., their shared events, such as synchronisation and communication).

We are guided in the task of informally describing something when we follow the above “recipe”, the above “narration” dogma — which leads on to the formalisation itself.

Chapter 2 of Vol. 3 (Sect. 2.5.3) presents examples, principles and techniques for the construction of proper narratives. These principles and techniques emerge from most chapters in Vols. 1 and 2. Specific domain, requirements, and software design narration principles and techniques are then covered in Parts IV–VI, respectively, of Vol. 3.

Formal Model

Characterisation. By a *formal document* we mean a document which expresses a model (of some universe of discourse) in a formal language. ■

A formal model is a model expressed in some mathematical notation or in some formal language. A mathematical expression permits conventional, albeit precise reasoning, such as is normally done in textbooks on mathematics. A formal language is one with a precise syntax, a precise semantics and a mathematical logic proof system, that is, a set of proof rules that allow formal reasoning, such as is done in textbooks on mathematical logic but here with a twist! The informal narrative and a formal model may be intertwined, textually, such as we often see in mathematics and physics textbooks. The relation between the informal narrative and its formal model is necessarily informal. That is, is one that can never be proven correct, it must be validated.

Volumes 1 and 2 contain many chapters which present examples, principles and techniques for the construction of proper formal models. Specific domain, requirements and software design formalisation principles and techniques are then covered in Parts IV–VI, respectively, of Vol. 3.

Discussion

The informal rough sketch, the more structured, but still informal narration, and the formal model, may be manifested in separate documents or may be combined and intertwined with the analytic documents. Usually the rough sketch is not documented in a manner suitable for release other than to the directly involved client and developer staff, and then usually only to the development staff. We say that the informal narratives, the terminologies and the formal models may constitute deliverables. And we normally assume that the rough sketches remain proprietary documents of the development enterprise.

1.3.5 Analytic Documents

Characterisation. By an *analytic document* we mean a document whose subject is a descriptive document. The text of an analytic document analyses a descriptive document. ■

As the term indicates, analytic documents are documents whose content represents analyses of other documents, here the descriptive documents. We consider four kinds of analytic documents: those that represent (i) formation of concepts from rough sketches (during brainstorming), (ii) validation of formal and informal description documents, (iii) description property verifications, and (iv) verification of the correctness of development transitions (i.e., development steps).

There may be other analytic documents. Examples: documents whose content analyses behavioural aspects of the intended computing system, such as expected interface response times based on queueing theoretic studies; expected machine computation times based on complexity theoretic studies; details of dictionary or database hashing algorithms based on statistical studies of reference patterns; and so on. Also included may be documents whose contents analyse pragmatic issues such as, production line flow (congestion), based on statistical studies, for a project and production planning, monitoring and control computing system; company cash flow, based on similar studies, for a financial services or an electronic trading computing system; and so on. Further kinds of analytic documents can be imagined. We shall, in these volumes, only cover those just mentioned.

Rough Sketch Analysis and Concept Formation

The most important task in describing a domain, prescribing some requirements or specifying some software design is to identify the core concepts around which the universe of discourse evolves. On one hand are the phenomena in the domain, the facilities that are desired from the software or the software program constructs (data structures, procedures, etc.). On the other hand these phenomena, in the actual world, these facilities (to be made manifest in the required software), or these program code constructs are to be conceptualised (as for the domain) or are indeed concepts, that is, abstract ideas, once captured as requirements or in software code.

Thus we see a transition from a concrete, manifest, actual world of usually tangible phenomena to an abstract, intellectually perceivable, but usually intangible world of concepts. It is this transition, from what is perceivable, via what is conceivable, to that which is “made into” software, that we need to record.

We do so for the domain by first brainstorming, that is by sketching rough domain descriptions and, from those, through analysis, identifying domain concepts. Then for the requirements by conceiving. In that case by sketching

rough requirements “prescriptions” and, from those, through analysis, identifying requirements concepts. And finally we do this for the software by “casting”, that is, by sketching rough software “designs” and, from those, through analysis, identifying proper software constructs.

Analysis with the aim of forming concepts is an art. Perhaps the hardest thing to learn is to do it right, or at least to do it in such a way that pleasing, elegant and “economic” concepts emerge. But reading lots of analysis examples might help. Chapters 13 and 21 of Vol. 3 therefore present analysis and concept formation examples, principles and techniques that are found useful in conducting the analyses hinted at above.

Validation of Descriptions, Prescriptions and Specifications

Characterisation. By a *validation document* we mean an analytic document which validates the text of a description document ($\mathcal{E}c$.) with respect to the stakeholders of the described universe of discourse. ■

By $\mathcal{E}c$. we mean: prescription and specification document.

Domain descriptions must be validated, they are, most likely, written by a small group of primarily developers, aided by a likewise small group of client staff. But larger, more definitively representative groups of client staff need review domain descriptions in order to concur. The same holds for requirements prescriptions.

Domain description and requirements prescription validation is necessarily a process of interaction between client staff and developers, and is necessarily a process based on informal narrative and terminology descriptions. This kind of validation is a crucial one: It is necessarily an informal, human process, and it serves the role of getting the right product. Chapters 14 and 22 of Vol. 3 present validation examples, principles and techniques that are found useful in conducting the analyses hinted at above.

Verification of Properties of Specifications

Characterisation. By a *verification document* we mean an analytic document which proves, model checks, or tests statements made about the properties of a description or a prescription or a specification. ■

A domain description denotes a theory. The description is only a model of the domain, not the real domain. Expressed in precise English, and especially expressed in some formal language, the model designated by a domain description possesses some properties. The sum total of all these properties is a theory for the domain. The same is true for requirements prescriptions and software design specifications.

We can informally reason about such properties when given a consistent and relatively complete description (or prescription or specification). And we

may record this reasoning formally when we also have a formal description (formal prescription, formal [design] specification). The usefulness of formal models is that such theorems may be proven. Proof of such theorems affords a higher trust in the descriptions.

Example 1.5 *Towards a Domain Theory:* Assume that we have described a railway system, its network of lines and stations, its train timetables and the actual train traffic according to timetables. Let us further assume that the train timetables, and hence the traffic is modulo 24 hours: repeats itself daily and is always on time. Now a property that transpires only very indirectly from the train timetables (and hence the train traffic) could be the following variant of Kirchhoff's Law: For any station in the network, the number of trains arriving, over any 24 hour period, at that station, minus the number of trains ending their journeys at that station, plus the number of trains starting their journey at that station, equals the number of trains departing from that station, all over the same 24 hour period. ■

Informatics models of domains can be made into theories, just as were models of physical phenomena such as Newton's Theory of Mechanics, Thermodynamics, etc. Chapter 15 of Vol. 3 presents domain theory examples, principles and techniques that are useful in establishing domain theories as above.

Correctness of Development Phase, Stage or Step Transition

When we make the transition from the phase of describing a domain to the phase of prescribing requirements to software for support of activities in that domain we correctness-relate that transition, from the latter to the former. When we make the transition from the phase of prescribing requirements to software to the phase of specifying the required software we correctness-relate from the latter to the former. These correctness relations, when stated properly (and so they must be if we are to have trust in the development), can be informally reasoned about. And, if the descriptions, prescriptions and (design) specifications are formally expressed and the relations likewise, then the reasoning may be formally supported: Formal proofs of correctness may be made.

Phases can be decomposed into stages of development, and transitions between stages may be correctness-related and argued about. Stages can similarly be decomposed into steps, and transitions between steps may be correctness-related and argued about.

Note that we sometimes used the term 'can', and sometimes 'may'. We can always try reason informally, as do mathematicians. But it is not always possible today to formally prove properties and transition correctness. Reasons for this may be of the following: We may have constructed some unwieldy models that make the proofs too cumbersome. Or computing science, cum specification language designers, may not yet have researched and developed

appropriate specification language constructs and proof systems. Or we, the developers, are simply not good enough at stating and proving auxiliary lemmas and theorems. Or we are trying to prove a non-theorem, something that is false.

Discussion

We have surveyed the analytic documents that may arise during software development. There are at least four kinds of analytic document parts: concept formation, description (prescription and design specification) validation, property verification and correctness verification. Some analytic work is “inspiration-guided”, such as concept formation seems to be. Other analytic work is guided by human interaction, such as validation is. And yet other analytic work is formalisable, such as property and correctness verification can be.

To give a proper, comprehensive presentation of these three kinds of analytic work is, however, not a goal of these volumes. Instead we refer to specialised texts and monographs on software verification.

1.4 Formal Techniques and Formal Tools

Reading of this section can be skipped till the reader has read Chaps. 2–9 of the present volume. The section may to some lay readers appear a bit esoteric.

The aim of this early section is to make the reader aware of the fact that the languages in which one expresses domain descriptions and requirements prescriptions are not programming languages, but are specification languages. These specification languages need allow the expression of abstractions, so as to make easy the expression of essential properties, while allowing freedom of software design implementations.

1.4.1 On Formal Techniques and Languages

Characterisation. By a *formal technique* we mean both of the following: a technique that has a mathematical foundation, and thus can be explained mathematically, and a technique by which its user expresses descriptions, prescriptions and (design) specifications formally and is able to reason formally about what is expressed. ■

Thus a formal technique implies: Formal specification using subsidiary techniques and the possibility of formal verifications, with their subsidiary techniques. Therefore a formal technique requires a formal specification language.

Characterisation. By a *formal specification language* we mean all of the following: a language which has: a formal, mathematical syntax; a formal, mathematical semantics; and a formal, mathematical logic proof system. ■

In Chap. 9 of this volume we explain what is meant by a proof system. In Vol. 2, Part IV we will explain what is meant by formal syntax and formal semantics.

Normally, in conventional software engineering, only the last step of development uses an almost⁹ formal language, namely the coding (i.e., the computer programming) language. We shall advocate the use of formal languages from the very beginning, for all phases, stages and steps of development. In conventional software engineering many different kinds of informal description, prescription and (design) specification languages are deployed, some with one form of diagrammatic constructs, others with other constructs, but all without a proper syntax, let alone any discernible semantics.

1.4.2 Formal Techniques in SE Textbooks

The aims and objectives of these volumes hinge crucially on the ideas of formal techniques and formal tools. The purpose of this section is to motivate this central role of formality. Most, if not all, existing textbooks on software engineering shy away from propagating these ideas of formalism. If other textbooks on software engineering bring any material on what they call ‘formal methods’, it is usually in the form of a separate chapter appearing somewhere in the book. In these volumes formal techniques permeate all technical chapters. Formal techniques are deployable, and are hence to be taught in connection with all technical aspects of software engineering.

1.4.3 Some Programming Languages

A language, when seen as the means for expressing an engineering objective, can be considered a tool. As such, formal languages represent one class of software engineering tools. As for all crafts, many tools are needed, different size hammers, different size saws, different size screwdrivers, different size planners, etc., are needed for carpentry. That is, the artifact to be constructed, that is, its “nature” or its attributes (properties), determines exactly which of many different tools are to be deployed.

We have very many different kinds of programming languages, “past” and “current”¹⁰: functional programming languages such as LISP [370], • Standard ML [261, 389], • Miranda [502], and • Haskell [498], to mention a few; logic programming languages, including • Prolog [295, 351], and CLPR

⁹Usually most programming languages still do not possess a proof system.

¹⁰‘Current’ programming languages are marked with a bullet: •.

[312]; the imperative¹¹ programming languages of Fortran [14], Cobol [12], Algol 60 [24], Algol 68 [510], Pascal [522], • C [321]; object-oriented programming languages, such as Simula 67 [54], • C++ [489], Modula 2 and Modula 3 [262,401,525], • Eiffel [377,378], Oberon [434,526,528–530], and • Java [10,20,243,348,470,511]; and finally the parallel programming languages of PL/I [13,37], CHILL [145], Ada [128], and • occam [301,364,449].

1.4.4 Some Formal Specification Languages

We can also expect to have many different kinds of formal specification languages that are model-oriented or property-oriented.

On Model-Oriented Specification Languages

Some specification languages are model-oriented:¹² • VDM-SL [120,121,226,317], • Z [281,476,477,533], and • B [3].

Characterisation. By a *model-oriented specification language* we mean one which expresses whatever it specifies in terms of mathematical constructions (i.e., models) such as sets, Cartesians, lists, functions, etc. ■

On Property-Oriented Specification Languages

Other specification languages are property-oriented (algebraic semantics) specification languages:¹³ OBJ3 [233], • CafeOBJ [190,232], and • CASL [49,397,399].

Characterisation. By a *property-oriented specification language* we understand one which expresses whatever it specifies in terms of logical properties of what is specified. ■

¹¹An imperative programming language is one which primarily focuses on assignable variables, hence assignments, and hence has statements, and usually therefore statement labels and GOTOS. Statements, in a sense, prescribe: *Do this, then do that* — “imperially”.

¹²A model-oriented specification language allows for the expression of models in terms of mathematical entities such as sets, Cartesians, lists, maps, functions, etc. Chaps. 12–16 (of the present volume) will make the first presentations of model-orientedness.

¹³A property-oriented specification language allows for the expression of models in terms of logically expressed algebras. Chapters 9 and 12 will make the first presentations of algebras and property-orientedness.

On Property-Oriented + Model-Oriented Specification Languages

Other specification languages are “mixed” property- and model-oriented specification languages: • RSL [236, 238, 239].

In these volumes we mostly use the RAISE Specification Language, RSL. But, really, nothing prevents a lecturer from using, for example, VDM-SL or Z instead.

More on Programming Languages

One selects a programming language according to what one wishes to express, that is, the values one wishes to speak of. Different programming language categories, as listed above, favour different value spaces.

In *functional programming* we handle functions, their definition, application and composition, because functions (including ordinary operator/operand expressions) are thought to best capture the problem at hand.

In *logic programming* we express propositions and predicates, i.e., handle logical values, because it is thought that one can best express certain computing problems by characterising their properties.

In *imperative programming* we establish, initialise, update and read states, i.e., assignable variables, because states and state changes are thought to best capture the problem to be solved.

In *object-oriented programming* we establish, initialise, update and read special clusters of state components called objects, because dividing the problem up into a set of such objects and solving the problem by expressing the interaction between objects is thought to best capture the problem at hand.

In *parallel programming* we establish, initialise and compose processes, and select among processes in various deterministic or nondeterministic ways. In addition we express cooperation among processes through their synchronisation and communication because it is thought that one can best express certain computing problems by their decomposition into cooperating and concurrently operating processes.

Specification Languages Resumed

The situation is not that simple with formal specification languages. Indeed, there is the distinction between model-oriented and property-oriented formal specification languages mentioned above. So one can choose one from either category depending on what it is one wishes to express, and how.

Purists might choose either the Z (since 1980) or the B (since around 1990) specification language paradigm. Both are based on simple set theoretic notions, are utterly elegant and can traditionally handle what one would consider simple state-oriented sequential problems. Z has been extended in various ways: to express concurrency, or to express objects beyond its own basic, elegant modularity concept.

VDM [120, 121, 226] represents possibly the first full-fledged formal specification language concept (since early 1970s), and is still flourishing in the form of the ISO standardised VDM-SL. The RAISE [236, 238] Specification Language (RSL) was conceived, in the mid-1980s, as a successor to the VDM specification language, then colloquially known as *Meta-IV*.

RSL, which we primarily use in these volumes, features both property-oriented and model-oriented means of expression, has a somewhat sophisticated object-oriented means of compositionality, and borrows from CSP [288, 289, 448, 456] to offer a means of expressing concurrency. Extensions to RSL have also been proposed, for example with timing [535], and with Duration Calculus, that is, temporal logic ideas [274].

1.4.5 Insufficiency of Current, Formal Languages

The story as told above may give you the impression that the formal (programming as well as specification) languages offer sufficient expressibility to handle all situations, but this is not so. Few, if any, professionally supported programming languages offer means for expressing temporal notions such as absolute times, relative time (intervals), delays, etc. The same is true for specification languages. Accordingly we see a bevy of very fascinating programming languages focusing on expressing synchrony: *Esterel* [47, 48], *Lustre* [256] and *Signal* [248]. We also see specification languages involving temporal notions: *Timed Automata* [9], *TLA* (Temporal Logic of Actions) [331] and *Duration Calculus* [537, 538]. We also find some which provide for the expression of state transitions: *Petri Nets* [313, 421, 435–437], *MSCs* (Message Sequence Charts) [302–304] and *LSCs* (Live Sequence Charts) [171, 270, 325], and *Statecharts* [265, 266, 268, 269, 271]. We shall have more to say about Petri nets, sequence charts, statecharts and the duration calculi [537, 538] in Vol. 2's Chaps. 12–15.

What does this plethora of programming and specification languages signify? First, it tells us that we are still in the early days of computing science, and hence software engineering. Proposals for new and better languages, or for altogether different language paradigms, are being put forward continually. It also probably tells us that we should not seek “universal” languages, that could handle all the “things” that one wishes to express. We shall probably have to settle for using combinations of different languages when specifying and when implementing problems.

More generally, it tells us that we shall, in these volumes, be content with the formal specification languages that are available today, while recognising their (and our) shortcomings. That is, there are situations in these volumes where we would like to show a formal specification of a problem, but where that would entail a longer introduction of a “new” notation, or where we simply have to give up because no pleasing or adequate or even known such language can be found!

1.4.6 Other Formal Tools

The most well-known formal tool for software development is a compiler: It accepts programs in a formal language, the source programming language, it checks that input programs satisfy a wide variety of static properties, and if so, it generates an output program in a target coding language, such that the meaning of the input program is preserved in the meaning of the output program. To do this properly a compiler embodies a number of instantiations of *theoretical artifacts*. These include a *finite state machine* which processes (ASCII) character strings into either keyword or identifier tokens, and other symbols into appropriate delimiter or operator tokens; a *push-down stack machine* which processes strings of tokens and creates, while checking, suitable internal representations of the input program (dictionaries, a parse tree, etc.); a *rewrite system* that transforms these internal representations into other, sometimes claimed optimised representations; and another *rewrite system* that finally transforms possibly resulting internal representations into output code.

Other formal tools are possible and exist: type checkers for abstract specifications; general data or control flow analysers, proof checkers, proof assistants; model checkers, theorem provers, and program interpreters. These, together with compilers, are all examples of what we in general call *abstract interpreters*, or *partial evaluators*. The current understanding of the role and possibilities of *abstract interpretation* is far from complete [163,164,215,231,320].

1.4.7 Why Formal Techniques and Formal Tools?

Some Rationale

Engineering, in its classical forms, civil, mechanical, electrical, all deploy calculations in one form or another. They do so in order to determine structural properties and design parameters, for example, for reinforced concrete or steel constructions, aircraft wing design, electrical transformer design, and so on. When we drive over a bridge, fly in an aircraft, or use some electrical appliance, we do so with some confidence that the classical design engineers have been properly trained in how to, and, when required, can, and indeed do, perform such calculations.

When we use an ordinary text processing system, yes, even when we send otherwise “innocent” (read: unimportant) e-mails, then we do not bother much about the “error-freeness” of that software. But when we fly an aircraft, or live next to a nuclear power plant, or receive our monthly paycheck (calculated from a myriad of interdependent tax regulations), or follow instructions from a medical doctor, and when we are told that any of these, the aircraft, the power plant, the paycheck processing and the medical advice, are monitored and partly or fully controlled by a computer, we may wonder about the correctness of the relevant software! But are the software engineers

comparatively well trained in the many calculi that do indeed exist today for securing trust in the software, and, if so, are they actually deploying such calculi? The answer is, wrt. current practice, sadly, no! These volumes will teach you some, but certainly far from enough, such calculi, i.e., formal techniques.

The answer to the rhetorical question of this section, *Why formal techniques and formal tools?* is therefore: *Because we need the highest possible degree of trust, given today's knowledge, in our software!* Since it can be done, namely, *ensuring highest possible degree of trust*, it must be done. Not ensuring so would be tantamount to cheating the customer — also known as criminal neglect!

Anecdotal cum Analogical Evidence

Until the mid-1700s most ships' captains (and their ships' mates) did not know how to reckon the longitude¹⁴. The chronometer was first fully available and known by the last quarter of the 1700s. Samuel Pepys¹⁵ commented on the pathetic state of navigation:

It is most plain, from the confusion all these people are in, how to make good their reckonings, even each man's with itself, and the nonsensical arguments they would make use of to do it, and the disorder they are in about it, that it is by God's Almighty Providence and great chance, and the wideness of the sea, that there are not a great many more misfortunes, and ill chances in navigation than there are.

We bring that story here for analogical purposes.

We claim that developing software without using formal techniques is like sailing the high seas without knowing how to compute the current longitude. We claim that nobody can become a ship's mate, much less a captain, if they do not know how to compute the longitude.

It is as simple as that, but the problem itself is not simple. It was, perhaps, more obvious, that the chronometer had indeed solved the longitude problem. To some it is still not obvious that formal specification and related techniques (verification, etc.) have brought us a long way towards having solved the software development problem.

1.5 Method and Methodology

We refer to Vol. 3's Chap. 3 for a more thorough treatment of the concepts of method, methodology, principles, techniques and tools. Suffice it here to give a brief account of these terms.

¹⁴Those "funny" lines (on a map of the world, or, as here, more appropriately, of the seas) which stretch between the arctic poles.

¹⁵From a trip as a high official of the British Royal Navy, 1683, from England to Tangier.

1.5.1 Method

Characterisation. By a *method* we understand a set of *principles* for *selecting* and *applying* a number of *analysis* and *synthesis (construction)* *techniques* and *tools* in order *efficiently* to *construct* an *efficient* artifact, here software (i.e., a computing system). ■

The above will be our guiding characterisation of the concept method. It will flavour these volumes. We will endeavour to enunciate such principles, techniques and tools that will guide the software engineer in where to start, how to proceed and where to end.

In the above characterisation we have also *emphasized* the things about or to which the principles, techniques and tools are concerned or apply, *selecting, applying, analysis, synthesis (construction)* and *efficiency*. Humans select the principles, techniques and tools. Hence choices of selection form a crucial aspect of a method. We, humans, or machines, i.e., *tools, apply techniques*. Hence *modes of application* form a crucial aspect of a method, likewise for *analysis* and *construction*. *Efficiency*, as a concept, applies both to the development process and to the developed artifact. We have added *efficiency* as an attribute of the concept of a method.

1.5.2 Methodology

Characterisation. By *methodology* we understand the study of, and the knowledge about one or more *methods*. ■

These volumes also cover *methodology*: We will contrast several *methods*, including several alternative *principles, techniques* and *tools*. No one *method* suffices for all software. There are a number of *principles, techniques* and *tools* that can help us. But for any one *method* there are still *principles, techniques* and *tools* to be identified, studied and tried out.

1.5.3 Discussion

The *principles* are to be interpreted by humans. The *selection* and *analysis* is to be mostly performed by humans. Some *techniques* and some *tools* can be used by machine, i.e., are formalised. But far from all. Hence it is a misnomer to refer to a concept of *formal methods*. It seems appropriate to refer to some techniques and some tools as being *formal*. So we conclude: Methods cannot be formal.

1.5.4 Meta-methodology

In this book, that is, in these volumes we shall highlight certain pieces of texts. These highlighted texts are concerned with

- *characterisations,*
- *definitions,*
- *principles,*
- *techniques,*
- *tools, and*
- *examples*

as follows. In the text the following kinds of highlighted texts will stand out. Please take appropriate note of these texts.

Characterisation. Characterisations are descriptive texts. They are not precise definitions. ■

Definition. Definitions are descriptive texts at the level of mathematical precision. We present definitions either as shown in the present definition, as numbered and highlighted paragraphs, or as mathematical texts or as RSL specifications. ■

Principles. Principles are here seen as comprehensive and fundamental laws, doctrines, assumptions or rules (codes) of conduct underlying the pursuit of software engineering. It is our principle to enunciate characterisations, definitions, principles, techniques and tools, and to bring many examples. ■

Techniques. Techniques are here concerned with the manner in which technical details are treated by the software engineer. The techniques of presenting highlighted characterisations, definitions, principles, techniques and tools are basically those used for descriptive texts. ■

Tools. Tools are here seen as intellectual (or even software) devices that aid in accomplishing a task, that is, are used in performing an operation or necessary in the practice of the profession of software engineering. The tool for presenting highlighted characterisations, definitions, principles, techniques and tools is that of English. ■

Example 1.6 The previous five **boldface** highlighted paragraphs together exemplified the ideas enunciated in this section. They all ended with the “■” symbol; and so does does this example. ■

1.6 The Very Bases of Software

This section previews the core issues of software engineering. The treatment here is, perhaps, a bit taxing, that is, it requires careful reading. You may wish to skip this section and return to read it after having studied, for example, the first half of this volume!

Before introducing types, functions and relations, algebras, and logic, we must, however, first cover some even more basic material: What is meant by didactics and paradigms, and what is meant by semiotics, that is, pragmatics, semantics and syntax. In other words, this section collects and presents a number of basic concepts, and as such it is a prelude to Part II of this volume.

1.6.1 Didactics and Paradigms

Life is rather a subject of wonder, than of didactics

Ralph Waldo Emerson 1803–1882

We are guided by *paradigms*, see Sect. 1.6.3. Good paradigms, we claim, reflect reasonably clarified *didactics*.

The Shorter Oxford English Dictionary [350] (*OED*) defines: didactics *having the character or manner of a teacher; characterised by giving instructions; instructive; preceptive; and systematic instruction*.

We shall, in these volumes, take the word *didactics* to mean *the basic ideas of practical or theoretical nature upon which the practice of a field of human activity is (best, or reasonably) pursued*. We claim that our rendition is commensurate with the *OED* explanation. There are other didactic and practical bases for software engineering than just types, functions, algebras and mathematical logic such as mentioned earlier. Although we shall in later volumes devote separate chapters to covering these other didactic bases in detail, we shall, in order that we may be able to refer to the very essence of these bases (before we reach those chapters), cover the concepts briefly. They are semiotics and descriptions.

1.6.2 Pragmatics, Semantics and Syntax

Semiotics can, for our purposes, fruitfully be understood as the study and knowledge of pragmatics, semantics and syntax of language. That is, respectively the use, meaning, and analysis and synthesis of language texts.

Pragmatics

Characterisation. By the *pragmatics* of a language we mean its use in social context: Why a particular expression used? What “ultimate” motive lies (seems to lie) behind an utterance, an expression. ■

We have some ulterior motives when specifying: What is it? What are they? Pragmatics, characterised somewhat convolutedly, is that *which cannot be formalised!* Pragmatics is the “real thing”. Syntax and semantics enable us to convey and, it is hoped, to understand, those “real things”!

Software specification languages and, more generally, computing systems specification languages serve to describe domains, prescribe requirements and

specify software designs. Thus their pragmatics, as well as the pragmatics of the individual domain, requirements and software design specifications, are that they are able to cover that spectrum, and that they, individually, allow for certain kinds of for example trustworthy and manageable development. Thus the design of any specification language, such as B, Cafe-OBJ, CASL, RSL, VDM-SL and Z, has taken into account which target applications that language best caters to. The main specification language of these volumes is RSL. As we shall see, RSL covers a rather broad spectrum. Two, amongst several more, important aspects of RSL are that it allows modular, reusable development and provably correct development.

Semantics

Characterisation. *Semantic* is about the meaning of what we express syntactically. ■

We shall later sharpen this characterisation, but first we express some deeply felt dogmas. Semantics, in some sense, *is what it is all about abstractly!* Pragmatics, in that sense, is what it is about concretely, in a specific social, human context. If we cannot express the essence abstractly, then we have not understood it. Then we can only have little trust in any software derived from such an incomplete understanding. Software is, by nature, abstract and is necessarily conceptual. Therefore it is more important to capture, mentally, the semantics before we search for a way to express it syntactically. Our best abstractions are those of mathematics. Mathematics is the science of abstraction.

So what is the semantics of RSL specifications? To appreciate and understand the choice made for the semantics of RSL, let us consider some very basic RSL specifications. Usually a specification names “things”.

Example 1.7 *Semantics of Class Specifications:* Our example is just that: It does not model anything “practical”, but illustrates, at a minimum cost of symbols, what we wish to say about semantics.

```
[0] scheme EXAMPLE =
[1]   class
[2]     type
[3]       A = Int, B = Nat
[4]     value
[5]       f: A → B
[6]     axiom
[7]       [ bijection ]
[8]        $\forall a:A, a':A \cdot a \neq a' \Rightarrow f(a) \neq f(a')$ 
[9]   end
```

Five things are named: (i) A class expression (**EXAMPLE**, lines [1–8]), (ii–iii) two types of values, **A** and **B** (lines [2–3]), (iv) a function, a value, **f** (lines [4–5]) that maps **As** (integers) into **Bs** (natural numbers), and (v) an **axiom bijection** (lines [6–7]) that expresses that **f** for distinct arguments yields distinct results.

Of the five things named only four designate specific mathematical entities. The **axiom** name, always enclosed in brackets, [...], may be put before the **axiom** keyword, and is there for a pragmatic reason so that we can refer to that **axiom**. Thus **axiom** names are optional and can be omitted.

Now what semantics does RSL ascribe to the identifiers **EXAMPLE**, **A**, **B** and **f**? We start “inside out”: **A** and **B** stand for the sets of integer, respectively sets of natural number values, and **f** for any function that satisfies the axiom. The class definition, **EXAMPLE**, etc. (lines [0–8]) now stands for a set of models, where a model provides a mapping from identifiers, such as **A**, **B** and **f**, into their meanings. All members of the set of models have **A** and **B** stand for the same universes of integers, respectively natural numbers, but each member of the set has **f** map into a distinct function from **A** into **B**, such that this set of models exhibits all such functions **f** in fact infinitely many! Hence **EXAMPLE** stands for an infinite set of models.

We summarise: Each **type** and **value** thing named by the specifier, e.g., you, in a specification, has a meaning. And that meaning may deterministically be a value, or a specific set of (typed) values, as for type names, or nondeterministically be one or another from amongst a possible infinity of values, as for the illustrated function name. So, functions can be values. The set of all values contains the set of all functions. Combining two or more such meaningful identifiers as here in a class expression, or just as a juxtaposition of definitions without the **class** keyword and class name results in a named, respectively unnamed set of (one or more) models. Axioms may be so constraining that there may be no model that satisfies the axioms. Or there may be a finite number of models, including just one!

Let us “display” the set of models for the class expression (lines [0–8]):

```
{
  [ A ↦ { ..., -2, -1, 0, 1, 2, ... },
    B ↦ { 0, 1, 2, ... },
    f ↦ λa • if a < 0 then
      3*(2*(-a)) else if a = 0 then 0 else 3*(1+2*a) end end,
  ... ],
  [ A ↦ { ..., -2, -1, 0, 1, 2, ... },
    B ↦ { 0, 1, 2, ... },
    f ↦ λa • if a < 0 then
      5*(2*(-a)) else if a = 0 then 0 else 5*(1+2*a) end end,
  ... ],
  [ A ↦ { ..., -2, -1, 0, 1, 2, ... },
    B ↦ { 0, 1, 2, ... },
    f ↦ λa • if a < 0 then
```

```

        7*(2*(-a)) else if a=0 then 0 else 7*(1+2*a) end end,
    ... ], ...
}

```

By $\lambda a:A \bullet E(a)$ we mean the function which when applied to an argument x in A yields a value as prescribed by the function body $E(x)$, i.e., where all free a in $E(a)$ have been replaced by x . By the ellipses, that is, \dots , we intend to show that the model may contain parts which map other identifiers into other mathematical values. ■

In the rest of these volumes we shall return, again and again, to semantic models of the above kind.

Syntax

Characterisation. *Syntax* is about how we can, in our case, write down specifications: rules of form, basic forms and their proper compositions. These rules for formal languages are to be of such a nature that the forms, that is, the language expressions, can be analysed, and such that, from the analysis, one can ‘construct’ (construe) the meaning. ■

Syntax is, of course, important, but its importance is secondary to semantics! We should strive for semantic clarity, then syntactic elegance. If the idea to be expressed is “muddled”, then no matter how beautiful the syntactic forms may be, humans will not easily understand them!

You have already seen some RSL syntax, for example, the scheme definition of Example 1.7. Since RSL is aimed at a rather wide spectrum of applications and at a full spectrum of development, from descriptions of actual domains, via requirements prescriptions to abstract software designs, the RSL syntax is rather “rich”. That is: has many entities. We shall try unravel these, gently, as we go along in these volumes, and only introduce the syntax that we need up to any given point in these volumes.

The syntax of class expressions, as exemplified above, thus appears to be covered by:

```

<class_expression> ::=
    class
        type
            <type_definitions>
        value
            <value_definitions>
        axiom
            <axiom_definitions>
    end

```

But since there are many more aspects to class expressions than illustrated so far, the syntax is more complicated than hinted at above.

When explaining a specification language construct we ought systematically cover its general forms and its static semantics, that is: which constraints limit the use of for example identifiers, operator symbols, keywords, delimiters, etc. and its meaning. We will, however, only give cursory explanations, leaving details to the RSL Reference Handbook [236].

1.6.3 On Specification and Programming Paradigms

We are guided by *paradigms*:

- (1) **Paradigm:** thing copied.
 - (2) **Model:** pattern, standard, rule, original, mirror;
 - (3) **Prototype:** archetype, antetype;
 - (4) **Precedent:** lead, representative, epitome
- Roget's International Thesaurus* [445].

Using paradigms we construct artifacts:

The universe ... was made exactly conformable
to its Paradigme, or universal Exemplar.
(*The Shorter Oxford English Dictionary* [350].)

These volumes are structured according to a set of specification paradigms. And these again rest on what we believe are the didactic bases of the practice and theory of software engineering.

So which are the “most basic” paradigms? Generally, we can say this: Abstraction is a specification paradigm; so is “favouring, encouraging” non-determinism in specification. Respective programming styles — functional (also referred to as applicative), logic, imperative, and parallel programming — represent a programming paradigm. Favouring a specification style that allows formally verifiable transformations of (more) abstract specifications into (more) concrete ones, and these finally into ‘executable programs’ — is a software development paradigm. There are then paradigms within paradigms: Practicing the functional specification (or the functional programming) paradigm may then be according to, for example, the *continuation* [59, 63, 315, 392, 404, 440, 471, 487, 513, 514] programming paradigm. Likewise practicing the parallel specification (or the parallel programming) paradigm may then be according to, for example, the CSP, i.e., the *communicating sequential processes*, [287, 288, 448] paradigm, and so on.

1.6.4 Descriptions, Prescriptions and Specifications

We shall, in these volumes, try strictly to use the following terms consistently and according to the following overlapping classification:

- **Description:** As a general term encompassing the below, and as a special term in connection with textual characterisations of domains.
- **Prescription:** As a specific term used primarily in connection with requirements.
- **Specification:** As a general term encompassing the above, and as a special term in connection with textual characterisations of software designs.
- **Definition:** As a general term encompassing formalisations, also of the above; and as a special term in connection with certain textual characterisations, namely and specifically, those parts that constitute proper definitions as distinguished from designations and refutable assertions.

Software Specifications, Requirements Prescriptions and Domain Descriptions

To direct a computer to perform any computation it must be so instructed. These instructions form a program. A program is a finite specification of possibly infinite sets of possibly infinite computations. So, descriptions, prescriptions and specifications form the most essential object of our endeavour: to develop software. We first explain the idea of specification, then the idea of prescription, and finally we explain the idea of description.

We *specify* computations; thus: to *design software* we *specify* how the computations should proceed: *the how* is an end goal. We *prescribe the what*, that is, the *requirements* that we expect the subsequently designed software to fulfill. And, before all that we *describe* the actual world in which these computations are to occur, that is, the (*application*) *domain*.

1.6.5 Metalanguages

We use language, say \mathcal{M} , to describe or “to talk about” other languages, say \mathcal{L} . One cannot use \mathcal{L} to describe \mathcal{L} . It leads to nonsense. \mathcal{M} is said to be a metalanguage for \mathcal{L} . To describe \mathcal{M} we need another metalanguage, or, as we could call it, a meta-metalanguage \mathcal{M}' .

The language, say \mathcal{M} , in which we explain mathematics, i.e., the notation of mathematics and its meaning, \mathcal{N} , is thus necessarily different from \mathcal{N} . We do not describe \mathcal{M} .

1.6.6 Summary

We have briefly introduced the notions of didactics and paradigms; and of semiotics: pragmatics, semantics and syntax. We have also introduced documents: informative, descriptive and analytic, as well as (domain) descriptions, (requirements) prescriptions and (software) specifications. We have finally introduced the notions of metalanguages, and object languages.

We shall later cover these in quite some detail. Suffice it, for now, to say that the reader now knows that these are basic concepts whose reasonable understanding is indispensable when pursuing professional software engineering.

1.7 Aims and Objectives

By the ‘aims of these volumes’ we mean the topics that we will be covering or dealing with. By the ‘objectives of these volumes’ we understand that which we wish to achieve through covering certain material.

1.7.1 Aims

The Main Aims

The main aims are to teach you general software engineering principles, techniques and tools. That is (in Vol. 3): those of domain engineering, of requirements engineering and of software design. Among these we additionally single out and teach principles, techniques and tools of abstraction and modelling (in Vols. 1–2); of description (in Vol. 3); and of documentation (in Vol. 3).

Some Other Aims

Additional aims are those of providing appropriate mathematical foundations, (Vol. 1, Part II), of ensuring appropriate understanding of semiotics issues: pragmatics, semantics and syntax (Vol. 2, Part IV), and of doing all of this within an appropriate framework of models and definitions (Vol. 3, Chaps. 4 and 6).

An aim, altogether “orthogonal” to the other aims above, is to illustrate development components of software for the support of large, distributed and concurrent infrastructure subsystems and systems.

1.7.2 Objectives

The Main Objectives

The main objectives are to help ensure that you become a professional engineer within software, to thus help ensure that the software (cum computing) systems, in whose development you are involved, become trustworthy systems of highest attainable quality, and through our emphasis on exemplifying the development of software (cum computing) systems for infrastructure components to help ensure that you, with colleagues, believably can develop highly sophisticated systems.

Some Other Objectives

Other objectives are to put the broader concerns of software engineering, such as treated in these volumes, in the context of other, indispensable and more specialised computing science disciplines such as artificial intelligence and knowledge-based systems, compiler systems, concurrent, safety-critical

and real-time application systems, database management systems, distributed systems, operating systems, secure, en- and decryptable systems, and so on. Another objective is to show that formal techniques are applicable, in all phases, stages and steps of development, and to all kinds of computing systems.

1.7.3 Discussion

The usual *aims and objectives* section has been dispensed with, but with a change: usually the two concepts, aims and objectives, are “lumped” into one treatment. Here we have separated them, properly.

There is a conceptual triangle: there is the author of these volumes; there is you, the reader, who studies its contents; and there is the most important thing: the subject itself: software engineering. Aims are about which software engineering topics the author wishes to cover, i.e., to teach you. Objectives are about which effects, with respect to the discipline of software engineering, the learning of these topics is to have on you. In other words aims are about ‘what’; objectives are about ‘why’.

1.8 Bibliographical Notes

This book, all three volumes of it, is different from most other textbooks on software engineering. We shall single out the following major ways in which this book differs from the following textbooks: [423, 430, 475, 512]. First they really are short on real development examples: there are hardly any real examples of specification and design. The present book, all three volumes of it, hinges crucially on real examples of specification and design. Second, when they bring a chapter on formal methods, do so in a separate chapter “tucked away” somewhere, ad hoc. The present book emphasises the use of formal techniques in all phases, stages and steps of development. Third, they, also including [240], do not bring any material on domain engineering. It is perhaps the last thing, domain engineering, in which this book is really new.

One very nice book, [240], does show a lot of formal techniques. Ours show almost all, if not all, of these techniques, and many, many more, and puts these techniques in the context of an overall methodology. The book by Watts Humphrey [298] is a wise book on management. “Hard to beat”. The book by Hans van Vliet [512] is, in our mind, the best overall of the above-referenced books when it comes to these practical and management issues.

1.9 Exercises

Exercise 1.1. *The Sciences:* Can you define what we, in these volumes, mean by computer science, and what we mean by computing science.

Exercise 1.2. *Project Management Issues:* Can you list some of the more practical, i.e., project management issues of software engineering.

Exercise 1.3. *The Triptych of Software Engineering:* Please list the three main phases of software engineering as put forward in this volume.

Exercise 1.4. *Documentation:* Can you list the three major classes of documents (as put forward in this volume) and, within each of the classes, can you list some of the major document parts.

Exercise 1.5. *Formal Techniques and Formal Languages:* Please define what these volumes mean by formal techniques and by formal languages.

Exercise 1.6. *Method and Methodology:* What does these volumes mean by (an efficient) method, and by methodology?

Exercise 1.7. *The Very Bases:* What does this chapter hint at as the meaning of a specification?



<http://www.springer.com/978-3-540-21149-5>

Software Engineering 1

Abstraction and Modelling

Bjørner, D.

2006, XL, 714 p. 38 illus., Hardcover

ISBN: 978-3-540-21149-5