

Opening

A brief introduction, Chap. 1, sets the stage for Part II, Chaps. 2–4. The introduction outlines what they cover and what they do not cover.

Introduction

"SLIDE 9"

In this chapter we shall overview the ‘trptych’ approach to software development. The paradigm, first proper section just below, motivates the triplet of ‘domain’, ‘requirements’ and software ‘design’ ‘phases’ covered briefly in Sect. 1.3. These phases can be pursued in a series of (usually sequentially ordered) ‘stages’ and the stages likewise in likewise ‘steps’. Work on many steps and some stages can occur in parallel. The stage and step concepts are introduced in Sect. 1.4 and covered in detail in Chaps. 2–4. The software engineering of these phases, their stages and steps are focused on constructing ‘documents’ — and the nature of these is covered in Sects. 1.5–1.8. Section 1.6 is the first major study section of this chapter. The core part of phase documents are either ‘descriptive’ (i.e., ‘indicative’, as it is), ‘prescriptive’ (i.e., ‘putative’ in the form of properties of what ones wants) or specifies a software design (i.e., are ‘imperative’). Sect. 1.9 briefly elaborates on these terms. The term ‘software’ is given a proper definition — one that most readers should find surprising — in Sect. 1.10. Section 1.11 covers the ideas behind pursuing software development both using informal and formal techniques. And Sect. 1.12 — another major study section of chapter — finally introduces the notions of entities, functions, events and behaviours.

"slide 10"

1.1 What Is a Domain ?

"SLIDE 11"

1.1.1 An Attempt at a Definition

Characterisation 1 (Domain) By a *domain* we shall understand a universe of discourse, small or large, a structure of entities, that is, of “things”, individuals, particulars some of which are designated as state components; of functions, say over entities, which when applied become possibly state-changing actions of the domain; of events, possibly involving entities, occurring in time and expressible as predicates over single or pairs of (before/after) states; and of behaviours, sets of possibly interrelated sequences of actions and events. ■

1.1.2 Examples of Domains

“SLIDE 12”

We give some examples of domains. (i) A country’s railways form a domain of the rail net with its rails, switches, signals, etc.; of the trains travelling on the net, forming the train traffic; of the potential and actual passengers, inquiring about train travels, booking tickets, actually travelling, etc.; of the railway staff: management, schedulers, train drivers, cabin tower staff, etc.; and so forth.

(ii) Banks, insurance companies, stock brokers, traders and stock exchanges, the credit card companies, etc., form the financial services industry domain.

(iii) consumers, retailers, wholesalers, producers and the supply chain form “the market” domain.

There are many domains and the above have only exemplified “human made” domains, not, for example, those of the natural sciences. We shall have more to say about this later. Essentially it is for domains like the ‘human made’ domains that this book will show you how to professionally develop the right software and where that software is right !

1.2 The Triptych Paradigm

“SLIDE 14”

*Before software can be designed one must understand its requirements.
Before requirements can be expressed one must understand the application domain.*

We assume that the reader understands the term ‘software’, but we shall, in Sect. 1.10, explain our definition of this term. By requirements we understand a document which prescribes the properties that are expected from the software (to be designed). By application domain we understand the business area of human activity and/or the technology area for which the software is to be applied. We shall, in the rest of this book, omit the prefix ‘application’ and just use the term ‘domain’.

1.3 The Triptych Phases of Software Development

“SLIDE 15”

1.3.1 The Three Phases

As a consequence of the “dogma” we view software development as ideally progressing in three *phases*: In the first phase, ‘Domain Engineering’, a model is built of the application domain. In the second phase, ‘Requirements Engineering’, a model is built of what the software should do (but not how it should that). In the third phase, ‘Software Design’, the code that is subject to executions on computers is designed.

1.3.2 Attempts at Definitions

“SLIDE 16”

Characterisation 2 (Domain Engineering) By *domain engineering* we shall understand the processes of constructing a domain model, that is, a model, a description, of the chosen domain, as it is, “out there”, in some reality, with no reference to requirements, let alone software. ■

Characterisation 3 (Requirements Engineering) By *requirements engineering* we shall understand the processes of constructing a requirements model, that is, a model, a prescription, of the chosen requirements, as we would like them to be. ■

“slide 17”

Characterisation 4 (Software Design) By *software design* we shall understand the processes of constructing software, from high level, abstract (architectural) designs, via intermediate abstraction level component and module designs, to concrete level, “executable” code. ■

Characterisation 5 (Model) By a *model* we shall understand a mathematical structure whose properties are those described, prescribed or design specified by a domain description, a requirements prescription, respectively a software design specification. ■

1.3.3 Comments on The Three Phases

“SLIDE 18”

The three phases are linked: the *requirements prescription* is “derived” from the *domain description*, and the *software design* is derived from the requirements prescription in such a way that we obtain a maximum trust in the software: that it meets customer expectations: that is, it is the right software, and that it is correct with respect to requirements: that is, the software is right.

“slide 19”

Characterisation 6 (Phase of Software Development) By a *phase of development* we shall understand a set of development stages which together accomplish one of the three major development objectives: a(n analysed, validated, verified) domain model, a(n analysed, validated, verified) requirements model, or a (verified) software design. These three “tasks”: a domain model, a requirements model, and a software design will be defined below. ■

“slide 20”

Characterisation 7 (Software Development) Collectively the three phases are included when we say ‘software development’. ■

Domain engineering is covered as follows: Chapter 2 outlines all the stages and steps of domain engineering. It does not bring examples. Instead the book provides for one large example, the ‘Model Development’ of most of Vol. II. Hence Appendices F–K provides in “excruciating” detail examples of

all the relevant aspects of domain engineering. These are then being referred to in Chap. 2.

Requirements engineering is covered as follows: Chapter 3 outlines all the stages and steps of requirements engineering. Like Chap. 2 Chap. 3 does not bring examples. Instead Appendices N–P provides in “excruciating” detail examples of all the relevant aspects of requirements engineering. These are then being referred to in Chap. 3.

Software design is not covered “in earnest” in this book. Chapter 4 overviews how one refines the requirements prescription, in stages and steps of development, into executable code. Appendix R, as a consequence, only offers some rudimentary examples.

1.4 Stages and Steps of Software Development “SLIDE 21”

We make distinctions between phases of development (i.e., the domain engineering, the requirements engineering and the software design phases), stages of development — within a phase, and steps of development — within a stage.

1.4.1 Stages of Development “SLIDE 22”

Characterisation 8 (Stage of Software Development) By a *stage of development* we mean a major set of logically strongly related development steps which together solves a clearly defined development task. ■

We shall later define the stages of the major phases, and we shall then be rather loose as to what constitutes a development step. That is, Chaps. 2–3 shall define the specific stages relevant to those phases of development.

1.4.2 Steps of Development “SLIDE 23”

Characterisation 9 (Step of Software Development) By a *step of development* we mean iterations of development within a stage such that the purpose of the iteration is to improve the precision or make the document resulting from the step reflect a more concrete description, prescription or specification. ■

1.5 Development Documents “SLIDE 24”

All we do, really, as software developers, can be seen as a long sequence of documenting, i.e., producing, writing, documents alternating with thinking and reasoning about and presenting and discussing these documents to and with other people: customers, clients and colleagues. Among the last documents to be developed in this series are those of the executable code.

In this section we shall take a look at the kind of documents that should result from the various phases, stages and steps of development, and for whose writing, i.e., as “input”, aside from other documents, we do all the thinking, reasoning, and discussing.

For any of the three phases of development, one can distinguish three classes of documents:

- Informative Documents Sect. 1.6 (Page 7)
- Modelling Documents Sect. 1.7 (Page 25)
- Analysis Documents Sect. 1.8 (Page 26)

1.6 Informative Documents

“SLIDE 26”

An informative document ‘informs’. An informative document is expressed in some national language.¹ Informative documents serve as a link between developers, clients and possible external funding agencies:

- “What is the project name ?” Item 1²
- “When is the project carried out ?” Item 1
- “Who are the project partners ?” Item 2
- “Where is the project being done ?” Item 2
- “Why is the project being pursued ?” Items 3(a)–3(b)
- “What is the project all about ?” Items 3(b)–3(g)
- “How is the project being pursued ?” Items 4–6

“slide 27”

And many other such practicalities. Legal contracts can be seen as part of the informative documents. We shall list the various kinds of informative documents that are typical for domain and for requirements engineering.

1.6.0 An Enumeration of Informative Documents

Instead of broadly informing about the aims and objectives of a development project we suggest a far more refined repertoire of information “tid-bits”. A listing of the sixteen names of these “tid-bits” hints at these:

“slide 28”

- | | |
|--|-------------|
| 1 Project Name and Date | Sect. 1.6.1 |
| 2 Project Partners (‘whom’) and Place(s) (‘where’) | Sect. 1.6.2 |
| 3 [Project: Background and Outlook] | |
| (a) Current Situation | Sect. 1.6.3 |
| (b) Needs and Ideas | Sect. 1.6.4 |
| (c) Concepts and Facilities | Sect. 1.6.5 |
| (d) Scope and Span | Sect. 1.6.6 |

¹ The fact that informative documents are informal displays a mere coincidence of two times ‘inform’.

² The item numbers refer to the enumerated listing given on Page 7.

(e) Assumptions and Dependencies	Sect. 1.6.7
(f) Implicit/Derivative Goals	Sect. 1.6.8
(g) Synopsis	Sect. 1.6.9
4 [Project Plan]	
(a) Software Development Graph	Sect. 1.6.10
(b) Resource Allocation	Sect. 1.6.11
(c) Budget Estimate	Sect. 1.6.12
(d) Standards Compliance	Sect. 1.6.13
5 Contracts and Design Briefs	Sect. 1.6.14
6 Logbook	Sect. 1.6.15

For examples of ‘information’ modelling and resulting documents we refer to Appendix E, Sect. E.1 and to Appendix M, Sect. M.1.

We shall now explain each of these kinds of informative documents.

1.6.1 Project Name and Dates

“SLIDE 30”

The first information are those of

- Project Name: the name of the endeavour;
- Project Dates: the dates of the project.

For examples of ‘project name and dates’ modelling and resulting documents we refer to Appendix E, Sect. E.1.1, Page 224 and to Appendix M, Sect. M.1.1, Page 365.

1.6.2 Project Partners and Places

“SLIDE 31”

The second information is that of

- Project Partners: who carries out the project.
Full partner (collaborator) details are (eventually) to be given:
 - ★ Client(s): full names, addresses, and possibly names of contact persons, etc., of the people and/or companies and/or institutions who and which have ‘ordered’ the project and who and which shall receive its resulting documents.
 - ★ Developer(s): full names, addresses, and possibly names of contact persons, etc., of the people and/or companies and/or institutions who and which are primarily developing the deliverables of the project and who and which shall receive its main funding.
 - ★ Project Consultant(s): full names, addresses, and possibly names of possible consultants, i.e., companies and/or individuals outside “the circle” of clients and developers.
 - ★ Project Funding Agencies: full names, addresses, possibly names of contact persons, etc., of the people and/or agencies who and which are possibly [co-]funding the project.

- ★ **Project Audience:** full names, addresses, and possibly names of contact persons, etc., of the people and/or agencies who and which are possibly (also) interested in the project.
- **Project Places:** where is the project carried out ? Full addresses: visiting and postal mailing addresses and electronic addresses.

For examples of ‘project partners and places’ modelling and resulting documents we refer to Appendix E, Sect. E.1.2, Page 224 and to Appendix M, Sect. M.1.2, Page 366.

1.6.3 Current Situation

“SLIDE 34”

Usually a domain engineering project is started for some reason. Either the developer or the client, or both, have only scant knowledge of the domain, or, when they have it is not written down but is “inside” the heads of some or most of their (i.e., developer or client) staff. Similarly a requirements engineering project is started for some reason. A common reason is that of the current situation on the client side. Either no IT is used but there is a need for some IT, or current IT is outdated, or new demands are made by owners, management or employees in general at the client, demands that “translate” into altered or new IT; or customers of the client may have similar expectations — of better e-service etc., from the client, i.e., their provider. For a software design project

“slide 35”

“slide 36”

The ‘Current Situation’ document must outline this in succinct terms: say half to a full page.

For examples of ‘current situation’ modelling and resulting documents we refer to Appendix E, Sect. E.1.3, Page 225 and to Appendix M, Sect. M.1.4, Page 366.

1.6.4 Needs and Ideas

“SLIDE 37”

Needs

Usually the current situation is paraphrased, i.e., accentuated, by expressions of specific ‘needs’ for a domain description, or for a requirements prescription, or for a completed software design, i.e., for software.

The need for a domain description could either be that it should form the basis for an orderly process of requirements development, or the basis for teaching and learning courses, say for new staff of the enterprise (of the domain), or both.

“slide 38”

The need for a requirements prescription could either be that it should form the basis for an orderly process of requirements development, or the basis for a tender, i.e., an offer to develop some software, or both.

“slide 39”

Usually can express needs while at the same time indicate how one might foresee an expressed need being possibly fulfilled, i.e., achieved.

“slide 33”

Dines Bjorner: 8th DRAFT: October 14, 2008

A need for a software design may be that it must be based on an existing requirements prescription.

A need for a requirements prescription may be that it must be based on an existing domain description.

A need for a domain description may be that it must be just informal, another need may be that it be both informal and formal.

Ideas³

One thing are the ‘needs’. Another thing are the ‘ideas’. If there are needs but no ideas, or if there is no need but ideas, then “forget it”: no reason to embark on a development !

By *ideas* we mean that there are some substantial concepts that, when properly deployed, can lead to a believable development, whether of a domain description, of a requirements prescription, or of a software design.

By *domain ideas* we mean such concepts “upon” or “around” which one can build, one can model, a domain description. Examples will be given in Sect. 2.3 on page 55

By *requirements ideas* we mean such concepts “upon” or “around” which one can build, one can model, a requirements prescription. Examples will be given in Sect. 3.3 on page 113

By *software design ideas* we mean such concepts “upon” or “around” which one can build, one can model, a software design. Examples will be given in Sect. 4.3 on page 130

• • •

For examples of ‘needs and ideas’ modelling and resulting documents we refer to Appendix E, Sect. E.1.4, Page 226 and to Appendix M, Sect. M.1.5, Page 366.

1.6.5 Concepts and Facilities

“SLIDE 42”

The pragmatics of the ‘concepts and facilities’ section is to — ever so briefly — inform all parties to the contract of which are the most important ideas of the subject domain of the contract. A facility is a physical phenomenon (here embodied, for example, in the form of software tools) while a concept is a mental construction (covering, usually some physical phenomena or concepts of these).

In the context of informing only about a domain description development project the concepts and facilities are intended, in the document section of that name, to be the most pertinent concepts and facilities on which the domain description should focus.

³ “SLIDE 40”

In the context of informing only about a requirements prescription development project the concepts and facilities are intended, in the document section of that name, to be the most pertinent concepts and facilities of the requirements prescription: which are the novel ideas which the requirements should be based. For examples of ‘concepts and facilities’ modelling and resulting documents we refer to Appendix E, Sect. E.1.5, Page 227 and to Appendix M, Sect. M.1.6, Page 366.

1.6.6 Scope and Span

“SLIDE 45”

Characterisation 10 (Scope) By *scope* — in the context of informative software development documentation — we shall understand an outline of the broader setting of the problem, i.e., the universe of discourse at hand. . ■

Characterisation 11 (Span) By a *span* — in the context of informative software development documentation — we shall understand an outline of the more specific area and the nature of the problem that need be tackled. ■

“slide 46”

Let us examine a few generic cases of scope/span determination.

(i) “Pure” domain engineering scope and span: By ‘“pure” domain engineering’ we mean a project aimed at just producing a domain model. In such a case the scope should typically be chosen as wide as possible, while the span is a proper, but not too “small” subset of the scope.

“slide 47”

(ii) Domain and requirements engineering scope and span: By ‘domain and requirements engineering’ we mean a project first aimed at producing a domain model and then, from it, “derive” a requirements model. In such a case the scope should typically be chosen to be comfortably wider than the scope of the requirements part of the project.

“slide 48”

(iii) Requirements engineering and software design scope and span: By ‘requirements engineering and software design’ we mean a project first aimed at producing a requirements model and then, from it, “derive” a software design. In such a case the scope and span part of the requirements part of the project should be equal. Software design projects have their scope and span being set by the requirements part of the project.

For examples of ‘scope and span’ modelling and resulting documents we refer to Appendix E, Sect. E.1.6, Page 228 and to Appendix M, Sect. M.1.7, Page 367.

1.6.7 Assumptions and Dependencies

“SLIDE 49”

There are two kinds of assumptions and dependencies. One kind has to do with sources of knowledge. For domain development there needs to be the sources from which the domain engineer can learn about and develop the domain description. We assume and depend on that. For requirements development there needs to be a domain description as well as people from whom

the requirements engineer can elicit the requirements and thus develop the requirements prescription. We assume and depend on that. And for software design there needs to be a requirements prescription. We assume and depend on that. The other kind has to do with delineation of the domain.

Usually a domain description (one upon which we base our (domain) requirements) leaves out what we might call the “fringes” of the domain, i.e., the environment of that domain. To also describe those parts might simply “be too much”! That environment is simply judged too large, too unwieldy, to describe.

Yet, sooner or later, that environment will show up in the requirements prescription, if it is not already in the domain description. The requirements prescription eventually, thus, comes to depend — maybe not exactly crucially, but anyway — on events originating in the environment, or the ability of the computing system to dispose of some output to that environment.

In the ‘assumptions and dependencies’ project document the project responsible must clearly express these assumptions and dependencies.

For examples of ‘assumptions and dependencies’ modelling and resulting documents we refer to Appendix E, Sect. E.1.7, Page 229 and to Appendix M, Sect. M.1.8, Page 367.

1.6.8 Implicit/Derivative Goals

“SLIDE 52”

Usually computing systems provide, or offer, a large number of entities, functionalities, events and behaviours, and it is those requirements we prescribe. But those entities, functionalities, events and behaviours really do not themselves reveal why they are or were prescribed. Usually their prescription serves “ulterior” goals which cannot be quantified in a way that indicates what the prescribed computing system should offer.

Typical meta-goals are such as: (i) “*Deployment of the computing system should result in greater profits for the company.*” (ii) “*Deployment of the computing system should result in the company attaining a larger market share for its products.*” (iii) “*Deployment of the computing system should result in fewer worker accidents.*” (iv) “*Deployment of the computing system should result in more satisfied customers (and staff).*”

Other kinds of meta-goals are: (v) “The existence of a domain description will have led or should lead to better understanding of the domain, hence to improved performance of domain staff trained in the domain based on such domain descriptions.” (vi) “The existence of a requirements prescription will have led or should lead to more appropriately targeted software.”

In the ‘implicit/derivative goals’ project document the project responsible must clearly express these implicit/derivative goals.

For examples of ‘implicit/derivative goals’ modelling and resulting documents we refer to Appendix E, Sect. E.1.8, Page 230 and to Appendix M, Sect. M.1.9, Page 368.

“slide 50”

“slide 51”

“slide 53”

“slide 54”

“slide 55”

1.6.9 Synopsis

“SLIDE 56”

The four sub-groups of informative document parts: current situation, needs and ideas, scope and span, and concepts and facilities, form an introductory “whole” that now need be “solidified”. They need to be brought together in a more coherent fashion — in what we shall call the synopsis document

“slide 57”

Characterisation 12 (Synopsis) By a *synopsis*⁴ — in the context of informative software development documentation — we shall understand the same as a resumé, a summary, that is, a comprehensive view, that is, an extract of a combination of current situation, needs and ideas, concepts, and scope and span documentation informing about a universe of discourse for which some development work is desired, for example: (i) the construction of a domain description, (ii) or the construction of a requirements prescription based on an existing domain description, or both; (iii) or the construction of a software design based on existing requirements prescription; (iv) or both (requirements and software design), (v) or all (domain, requirements and software design); (vi) or the first two (domain and requirements). ■

“slide 58”

For examples of ‘synopsis’ modelling and resulting documents we refer to Appendix E, Sect. E.1.9, Page 231 and to Appendix M, Sect. M.1.10, Page 368.

“slide 59”

“slide 60”

1.6.10 Software Development Graphs

“SLIDE 61”

Development projects need be managed. This is true also for single person projects. Management of domain engineering projects must take into account that these are normally research projects: little is objectively known about the domain before it is properly described; hence one must be prepared for “unforeseen” resource usage. Software development graphs are a means of capturing, either beforehand, during, or after the project how that project is to be done, is being done, or was done, respectively !

“slide 62”

Graphs⁵

Characterisation 13 (Software Development Graph) By a *software development graph* we shall *syntactically* understand a labelled graph whose distinctly labelled *nodes* (*vertexes*) designate development activities (phases, stages or steps), and whose distinctly labelled, directed *edges* (*arcs*) designate *precedence relations* between (node designated) activities.

“slide 64”

Semantically a software development graph designate a set of project behaviour designators. A *project behaviour designator* is a sequence of *phase*, *stage* or *step state designators* and *state transition designators*.

⁴ Synopsis: Greek, comprehensive view, from *synopsis*: to be going to see together.

⁵ “SLIDE 63”

A *phase, stage or step state designator* is a node label such that the node is that of a phase part, or a stage part, or a step part of a software development graph.

A *state transition designator* is an edge label such that the edge is that of an edge of a development graph. ■

A Conceptual Software Development Graph⁶

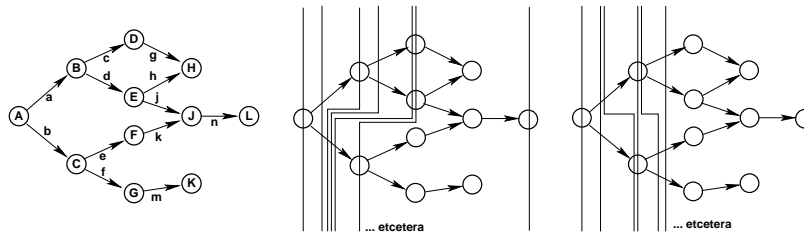


Fig. 1.1. A software development graph (left) and two (incomplete) project behaviour designers (center and right)

The center graph of Fig. 1.1 portrays the following incompletely listed project behaviour designer:

$$\langle \{A\}, \{a,b\}, \{B,b\}, \{c,d,b\}, \{D,E,b\}, \{D,E,C\}, \dots, \{L\} \rangle$$

The “abstracted” software development graph of Fig. 1.1 denotes a very large number of project behaviours, that is, a very large number of project behaviour designers, and, for each of these, depending on the states of phase, stage or step, as represented, for example, by the states of the documents related to each of the nodes, a very large number of (dynamic) behaviours.

Who Sets Up the Graphs ?⁷

Management is responsible for setting up an appropriate software development graph (for each project). A software development graph shows how management intends to pursue the project: which phases, stages and steps to conduct, that is, to which depth of adherence to the triptych principles management wishes to achieve its aims.

Chapters 2 and 3 will illustrate “abstracted”, i.e., generic, software development graph reflecting phases, stages and steps. Sections 2.14 and 3.14 summarize these (Figs. 2.3 on page 103 and 3.1 on page 124).

⁶ “SLIDE 65”

⁷ “SLIDE 67”

How Do Software Development Graphs Come About ?⁸

For a given, specific project, its software development graph comes about in any number of ways. (i) Either the project is a “repeat” project, that is, is developing a kind of software which has been developed before. In that case one simply uses the software development graph used in those earlier projects. But since there probably are some small, or perhaps not even that small, difference between the current project and the previous ones, the currently chosen software development graph may be modified. Thus every software development graph will be recorded for possible re-use in future. It becomes part of the “corporate assets” of the software house.

“slide 69”

(ii) Or the project is a “research” project, that is, is developing a new kind of software which has not been developed before. In that case one starts with the process diagram most appropriate for the project.

If it is a domain engineering project then one starts with the domain engineering process graph of Fig. 2.3 on page 103 as the software development graph; modifies this graph to suit the specific domain at hand, all the while recalling that development of domain descriptions are really research rather than engineering tasks, hence accepting that the software development graph need be modified along the way: clear resource estimates of time and effort cannot be assured.

“slide 70”

If it is a requirements engineering project then one starts with the domain engineering process graph of Fig. 3.1 on page 124 as the software development graph; and modifies this graph to suit the specific requirements at hand. One must always be prepared to modify the software development graph along the way.

For examples of ‘concrete software development graph’ modelling and resulting documents we refer to Appendix E, Sect. E.1.10, Page 233 and to Appendix M, Sect. M.1.11, Page 368.

1.6.11 Resource Allocation

“SLIDE 71”

Characterisation 14 (Software Development Graph Attribute) An *attributed software development graph* is a software development graph whose nodes and edges have been assigned development attributes. ■

“slide 72”

Usually *node development attributes* include whether the node is a domain, a requirements or a software design development node; whether the node is a phase, stage, or step node; of what specific kind the node — when not just a phase node — is: any one of the stages of the three triptych phases⁹;

⁸ “SLIDE 68”

⁹ The phases of domain and requirements modelling and analysis will first be “revealed” in Chaps. 2 and 3 — the only stage of the information document development is just that stage.

any one of the 16 kinds of information document development steps enumerated on Page 7; or any one of the many stages or steps of the domain and requirements modelling and analysis to be “revealed” in Chaps. 2 and 3.

Given an attributed software development graph and given experience from projects “similar” to the one described by the graph one can now estimate resources to be allocated to each task, that is, to the carrying out the actions implied by each of its nodes. These resource estimates are of the following kinds: number and qualifications of project staff; when, i.e., during which periods each individual, but not yet named staff, is to be available for the action denoted by the box being attributed; tools (office space, equipment (incl. IT equipment), software — by allocated staff members — to be available for that action; ‘begin’ and ‘end time’; etcetera.

These estimates can be affixed to the nodes (boxes); thus augmenting its set of attributes.

For examples of ‘resource allocation’ modelling and resulting documents we refer to Appendix E, Sect. E.1.11, Page 233 and to Appendix M, Sect. M.1.12, Page 368.

1.6.12 Budget (and Other) Estimates

“SLIDE 75”

From the augmented (i.e., extended attributed) software development graph one can now derive a number of estimates:

- (i) a budget estimate, per phase and stage, and thus for the entire (software development graph [SDG] designated) project;
- (ii) a time estimate, per phase and stage, and thus for the entire (SDG designated) project;
- (iii) a staff estimate, per phase and stage, and thus for the entire (SDG designated) project (here it must be analysed which activities can occur in parallel) and usually in the form of a histogram;
- (iv) an equipment estimate, per phase and stage, and thus for the entire (SDG designated) project;
- etcetera.

For examples of ‘budget estimate’ modelling and resulting documents we refer to Appendix E, Sect. E.1.12, Page 234 and to Appendix M, Sect. M.1.13, Page 368.

1.6.13 Standards Compliance

“SLIDE 76”

A distinction is made between development standards and documentation standards.

“slide 73”

“slide 74”

Development Standards

Usually development occurs in the context of following some development standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [102]) has established a number of standards for the development of a various kinds of software. Other national and international organisations, including the International Organization for Standardization (ISO [104]) and the International Telecommunication Union (ITU [108]), have established similar standards.

Documentation Standards¹⁰

Usually documentation occurs in the context of following some documentation standards (one or more). The Institute of Electrical and Electronics Engineers (IEEE [102]) has established a number of standards also for the documentation of a various kinds of software. Other national and international organisations, including the International Organization for Standardization (ISO [104]) and the International Telecommunications Union (ITU [108]), have also established similar standards.

Standards Versus Recommendations¹¹

Some standards are binding, some are recommendations. Reference to specific standards and recommendations can be written into project contracts with the meaning that the project must comply with these standards and recommendations. Some standards mandate or recommend the use — and hence the documentation style — of certain development practices. Other standards mandate or recommend the use of specific spelling forms, mnemonics, abbreviations, etc.

Specific Standards¹²

There are very many standards for software development and for its documentation. Some standards come and go. Others are quite stable. A study of more specialised standards reveals the following acronyms: MIL-STD-498, DOD-STD-2167A, RTCA/DO-178B, JSP188 and DEF STAN 05-91. The reader is invited to search for these on the Internet. It therefore makes little sense for us to list other than a few clusters of seemingly more stable and trustworthy standards.

“slide 80”

- *International Organization for Standardization (ISO):* <http://www.iso.ch/>

¹⁰ “SLIDE 77”

¹¹ “SLIDE 78”

¹² “SLIDE 79”

- ★ ISO 9001: Quality Systems Model for quality assurance in design, development, production, installation and servicing
- ★ ISO 9000-3: Guidelines for the application of ISO 9001 to the development, supply and maintenance of software
- ★ ISO 12207: Software Life Cycle Processes <http://www.12207.com/>
- IEEE Standards: <http://standards.ieee.org/>
 - ★ IEEE Std 610.12-1990, Standard Glossary of Software Engineering Terminology
This standard contains definitions for more than 1000 terms, establishing the basic vocabulary of software engineering.
 - ★ IEEE Std 1233-1996, Guide for Developing System Requirements Specifications
This standard provides guidance for the development of a set of requirements that, when realized, will satisfy an expressed need.
 - ★ IEEE Std 1058.101987, Standard for Software Project Management Plans
This standard specifies the format and contents of software project management plans.
 - ★ IEEE Std 1074.1-1995, Guide for Developing Software Life Cycle Processes
This guide provides approaches to the implementation of IEEE Std 1074. (This standard defines the set of activities that constitute the mandatory processes for the development and maintenance of software.)
 - ★ IEEE Std 730.1-1995, Guide for Software Quality Assurance Plans
The purpose of this guide is to identify approaches to good Software Quality Assurance practices in support of IEEE Std 730. (The standard establishes a required format and a set of minimum contents for Software Quality Assurance Plans. The description of each of the required elements is sparse and thus provides a template for the development of further standards, each expanding on a specific section of this document.)
 - ★ IEEE Std 1008-1987 (reaffirmed 1993), Standard for Software Unit Testing
The standard describes a testing process composed of a hierarchy of phases, activities, and tasks. Further, it defines a minimum set of tasks for each activity.
 - ★ IEEE Std 1063-1987 (reaffirmed 1993), Standard for Software User Documentation
This standard provides minimum requirements for the structure and information content of user documentation.
 - ★ IEEE Std 1219-1992, Standard for Software Maintenance
This standard defines a software maintenance process.
- Software Engineering Institute (SEI): <http://www.sei.cmu.edu>
 - ★ Software Process Improvement Models and Standards, including SEI's various Capability Maturity Models
- UK Ministry of Defence Standards <http://www.dstan.mod.uk/>

"slide 81"

"slide 82"

"slide 83"

"slide 84"

- * 00-55: Requirements for Safety Related Software in Defence Equipment
<http://www.dstan.mod.uk/data/00/055/02000200.pdf>
- * 00-56: Safety Management Requirements for Defence Systems
<http://www.dstan.mod.uk/data/00/056/01000300.pdf>

So, please, use the Internet for latest on standards relevant to your project.

For examples of ‘standards compliance’ modelling and resulting documents we refer to Appendix E, Sect. E.1.13, Page 235 and to Appendix M, Sect. M.1.14, Page 368.

1.6.14 Contracts and Design Briefs

“SLIDE 85”

Contracts

The current situation, needs and ideas, concepts and facilities, scope and span and synopsis document parts set the stage for, and are a necessary background for contractual documents. Usually one contract document is sufficient for small projects. And usually several related contract documents are needed for larger projects.

Characterisation 15 (Contract) By a *contract* — in the context of informative software development documentation — we shall understand a separate, clearly identifiable document (i) which is legally binding in a court of law, (ii) which identifies parties to the contract, (iii) which describes what is being contracted for, possibly mutual deliveries, by dates, by contents, by quality, etc., (iv) which details the specific development principles, techniques, tools and standards to be used and followed, (v) which defines price and payment conditions for the deliverables, (vi) and which outlines what is going to happen if delivery of any one deliverable is not made on time, or does not have the desired contents, or does not have the desired quality, etc. ■

“slide 86”

Items (iii–iv) constitute the main part of a *design brief*. (See below.)

“slide 87”

For national and for international contracts predefined forms which make more precise what the contracts must contain are usually available. We will not bring in an example. Such an example would have to reflect the almost ‘formal’ status of ‘legal binding’, and would thus have to be extensive and very carefully worded, hence rather long. Instead we refer to national and international contract forms.

The software development field is undergoing dramatic improvements. Clients are entitled to have legally guaranteed quality standards (incl. correctness verification). Hence contracts will have to refer to (i) the broader domain and give specific references to named domain stakeholders, if the development of a domain description is (to be) contracted; or (ii) existing domain descriptions and give specific references to named stakeholders, if the development of a requirements prescription is (to be) contracted; or (iii) existing requirements prescriptions and give specific references to named stakeholders, if the development of software is (to be) contracted.

“slide 88”

Therefore contracts should name “the methods” by means of which the deliveries will be developed — as we have indicated in item (iv) of the characterisation.

Contract Details¹³

- 1 **Overview:** Contracts between an organization and a software vendor should clearly describe the rights and responsibilities of the parties to the contract. The contracts should be in writing with sufficient detail to provide assurances for performance, source code accessibility, software and data security, and other important issues. Before management signs the contracts, it should submit them for legal counsel review.

“SLIDE 90”

Organizations may encounter situations where software vendors cannot or will not agree to the terms an organization requests. Under these circumstances, organizations should determine if they are willing to accept or able to mitigate the risks of acquiring the software without the requested terms. If not, consideration of alternative vendors and software may be appropriate.

- 2 **General Issues of Licensing:**

Software is usually licensed, not purchased; and under licensing agreements, organizations obtain no ownership rights, even if the organization paid to have the software developed. In general, for domain descriptions and requirements prescriptions, a license should clearly define permitted users and sites.

- 3 **Copyright:**

Proprietary as well as open-source software are protected by copyright laws. If need be then clients and vendors must make sure that also their domain descriptions and requirements prescriptions are protected by being proprietary.

- 4 **Domain, Requirements and Software Development Specifications:**

Contracts for the development of custom domain descriptions, requirements prescriptions, and software design must be very specific about “SLIDE 94” the scope and span of domain descriptions and requirements prescriptions, that requirements prescriptions build on accepted domain descriptions, that requirements prescriptions are feasible and satisfiable, and that software designs build on accepted requirements prescriptions.

- 5 **Performance Standards:**

This issue relates to requirements and software. When the requirements prescriptions are claimed feasible and satisfiable, then there must be software that satisfies the requirements. These requirements also include performance requirements, part of the machine requirements to be covered in Chap. 3.

¹³ “SLIDE 89”

- 6 Documentation, Modification, Updates and Conversion:** “slide 96”
 A licensing or development agreement should require vendors to deliver appropriate documentation. This should include all kinds of documentation — such as defined later. A license or separate maintenance agreement should address the availability and cost of document updates and modifications.
- 7 Bankruptcy:** “slide 97”
 In addition to escrow agreements, organizations should consider the need for other clauses in licensing agreements to protect against the risk of a vendor bankruptcy. For mission-critical software, organizations should consult with their legal counsel on how best to deal with the Bankruptcy laws, which typically gives a bankrupt vendor discretion to determine which of its executory contracts it will continue to perform and which it will reject. Proper structuring of the contract can help an organization protect its interests if a vendor becomes insolvent.
- 8 Regulatory Requirements:** “slide 98”
 Domain descriptions, requirements prescriptions and software designs must individually often have to comply with national (state and federal), regional (NAFTA, EU, etc.), and/or international (ICAO, IMO, etc.) regulatory agency requirements. These compliance requirements must be clearly stated in the contract.
- 9 Payments:** “slide 99”
 Software development contracts normally call for partial payments at specified milestones, with final payment due after completion of acceptance tests. Organizations should structure payment schedules so developers have incentives to complete the project quickly and properly. Properly defined milestones can break development projects into deliverable segments so an organization can monitor the developer’s progress and identify potential problems.
 Contracts should detail all features and functions the delivered software will provide. If a vendor fails to meet any of its express requirements, organizations should retain the right to reject the tendered product and to withhold payment until the vendor meets all requirements.
- 10 Representations and Warranties:** “slide 100”
 Organizations should seek an express *representation and warranty* — this is a statement by which one party gives certain assurances to the other, and on which the other party may rely — in the document deliverables, that the licensed documentation whether a domain description a requirements prescriptions, or a software design (incl. code) does not infringe upon the intellectual property rights of any third parties.
- 11 Dispute Resolution:** “slide 101”
 Organizations should consider including dispute resolution provisions in contracts and licensing agreements. Such provisions enhance an organization’s ability to resolve problems expeditiously and may provide for continued software development during a dispute resolution period.

"slide 102"

12 Agreement Modifications:

Organizations should ensure software licenses clearly state that vendors cannot modify agreements without written signatures from both parties. This clause helps ensure there are no inadvertent modifications through less formal mechanisms some states may permit.

"slide 103"

13 Vendor Liability Limitations:

Some vendors may propose contracts that contain clauses limiting their liability. They may add provisions that disclaim all express or implied warranties or that limit monetary damages to the value of the product itself, specific liquidated damages, etc.. Generally, courts uphold these contractual limitations on liability in commercial settings unless they are unconscionable. Therefore, if organizations are considering contracts, they should consider whether the proposed damage limitation bears an adequate relationship to the amount of loss the financial organization might reasonably experience as a result of the vendor's failure to perform its obligations. Broad exculpatory clauses that limit a vendor's liability are a dangerous practice that could adversely affect the soundness of an organization because organizations could be injured and have no recourse.

"slide 104"

14 IT Security:

We interpret this contract aspect only in the light of software. There is an ISO recommendation of IT Security: INTERNATIONAL ISO/IEC STANDARD 17799 Reference number ISO/IEC 17799:2005(E), ISO/IEC 2005, ISO/IEC 17799:2005(E), Information technology, Security techniques: Code of practice for information security management, ISO copyright office, Case postale 56, CH-1211 Geneva 20, Switzerland. E-mail copyright@iso.org, Web www.iso.org. Published in Switzerland. Second edition, 2005-06-15. We advice clients and developers to carefully adhere to that ISO recommendation.

"slide 105"

15 Subcontracting and Multiple Vendor Relationships:

Some software vendors may contract third parties to develop software for their clients. To provide accountability, it may be beneficial for organizations to designate a primary contracting vendor. Organizations should include a provision specifying that the primary contracting vendor is responsible for the software regardless of which entity designed or developed the software. Organizations should also consider imposing notification and approval requirements regarding changes in vendor's significant subcontractors.

"slide 106"

16 Restrictions and Adverse Comments:

Some software licenses include a provision prohibiting licensees from disclosing adverse information about the performance of the software to any third party. Such provisions could inhibit an organization's participation in user groups, which provide useful shared experience regarding software packages. Accordingly, organizations should resist these types of provisions.

Design Briefs¹⁴

Characterisation 16 (Design Brief) By a *design brief* we understand a clearly delineated subset text of the contract. To recall (from the characterisation): This text (item (iii)) describes what is being contracted for possibly mutual deliveries, by dates, by contents, by quality, etc., and ((iv)) it details the specific development principles, techniques and tools; that is, the design brief directs the developers, the providers of what the contract primarily designates, as to what, how and when to develop what is being contracted. ■

For examples of ‘contract and design brief’ modelling and resulting documents we refer to Appendix E, Sect. E.1.14, Page 236 and to Appendix M, Sect. M.1.15, Page 368.

1.6.15 Logbook

“SLIDE 108”

Characterisation 17 (Logbook) By a *logbook* we understand a record, a set of notes, which as correctly as is humanly feasible, lists the development, release, installation, use, maintenance, etc., history of a project. ■

A logbook serves as a necessary reference in innumerable, usually unforeseeable instances of development.

“slide 109”

Example 1 (Logbook) An “abstracted” ... (dot, dot, dot) example is:

2 Jan. 1991: Initial meeting between partners *ℰc*.
 ...
 31 May 1993: Acceptance of domain model *ℰc*.
 ...
 24 October 1994: Acceptance of requirements model *ℰc*.
 ...
 3 June 1996: Acceptance of software delivery *ℰc*.
 ...

The *ℰc*. signify reports, and the ... signify other logbook entries. . ■

1.6.16 Discussion of Informative Documentation

“SLIDE 110”

General

We have identified some useful components of informative document parts. There may be other such informative parts. It all may depend on the universe of discourse, i.e., the problem at hand. We thus encourage the software developer to carefully reflect on which are the necessary and sufficient informative document parts.

There is usually a separate set of informative documents to be worked out for each phase of development: (i) the domain phase, (ii) the requirements phase, and (iii) the software design phase.

“slide 111”

The current situation, needs, ideas, concepts, scope, span, synopsis and contract document parts differ in content between these phases. Usually the informative document parts, although crucially important, need not require excessive resources to develop, but their development must still be very careful!

In general, the informative document parts are concerned with the socio-economic, even geopolitical, and hence pragmatic context of the projects about which they inform. As such they are “fluid”, i.e., less precise, in what they aim at and what their objectives are. The next two documentation kinds are, in that respect, much more precise, and much more focused.

Methodological Consequences: Principle, Techniques and Tools¹⁵

Principle 1 (Information Document Construction) When first contemplating a new software development project, make sure — as the very first thing — to establish a proper complement of (all) informative documents. Throughout the entire development and after — during the entire lifetime of the result, whether a domain model, or a requirements model, or a software system — maintain this set of informative documents. ■

“slide 113”

Principle 2 (Information Documents) The informative documents must be authoritative, definitive and interesting to read. ■

“slide 114”

Technique 1 (Information Document Construction) First establish a document embodying the fullest possible table of contents, whether for just a domain development, or a requirements development, or a software design project, or for a combination of these. Then fill in respective document parts, “little by little”, just a few sentences, using terse, precise, i.e., concise language, while avoiding descriptions (prescriptions and specifications) and analyses. Throughout maintain clear monitoring and control of all versions of these documents. ■

“slide 115”

“slide 116”

Tool 1 (Information Document Construction) A text processing system, preferably L^AT_EX, but MS Word will do, with good cross-referencing facilities, even between separately ‘compilable’ documents, provides a ‘minimum’ tool of documentation. Add to this a reasonably capable version monitoring and control system (such as CVS [53]) and you have a workable system. ■

The subject of document version monitoring and control will not be dealt with in this volume.

“slide 117”

“slide 118”

¹⁴ “SLIDE 107”

¹⁵ “SLIDE 112”

1.7 Modelling Documents

“SLIDE 119”

Documents which describe, prescribe or specify something, such document are intended to model those things. They, the document, are not those things, just conceptualisations, i.e., models of them. In this book we shall only seriously cover the modelling of domains and of requirements.

1.7.1 Domain Modelling Documents¹⁶

“SLIDE 120”

Chapter 2 covers domain engineering in general and Sect. 2.9 covers domain modelling in particular.

Domain descriptions are documents. They are usually rather substantial. They usually include the following kinds of documents:

- | | |
|---|-----------|
| 1 stakeholder identification and liaison records, | Sect. 2.4 |
| 2 acquisition sketches, | Sect. 2.5 |
| 3 business process rough sketches, | Sect. 2.7 |
| 4 terminologies, | Sect. 2.8 |
| 5 and domain models proper. | Sect. 2.9 |

“slide 121”

Chapter 2 will cover the domain engineering phase with its

- | | |
|--|------------|
| • (i) stakeholder identification, | Sect. 2.4 |
| • (ii) domain acquisition, | Sect. 2.5 |
| • (iii) domain analysis and concept formation, | Sect. 2.6 |
| • (iv) business process rough sketching, | Sect. 2.7 |
| • (v) terminology, | Sect. 2.8 |
| • (vi) domain modelling, | Sect. 2.9 |
| • (vii) domain model verification, | Sect. 2.10 |
| • (viii) domain model validation, | Sect. 2.11 |
| • and (ix) domain theory formation | Sect. 2.13 |

stages. Documents emerge from each of these stages.

Documents 1, 2, 3, 4 and 5 correspond to (i), (ii), (iv), (v) and (vi). The other activities are analytic.

1.7.2 Requirements Modelling Documents¹⁷

“SLIDE 122”

Chapter 3 covers requirements engineering in general and Sect. 3.9 covers requirements modelling in particular.

Requirements prescriptions are documents. They are usually rather substantial. They usually include the following kinds of documents:

¹⁶ By ‘Domain Modelling Documents’ we mean the same as by ‘Domain Description Documents’.

¹⁷ By ‘Requirements Modelling Documents’ we mean the same as by ‘Requirements Prescription Documents’.

- 1 stakeholder identification and liaison records,
- 2 acquisition sketches,
- 3 business process re-engineering rough sketches,
- 4 terminologies, and
- 5 requirements models proper.

“slide 123”

Chapter 3 will cover the requirements engineering phase with its

- (i) stakeholder identification, Sect. 3.4
- (ii) requirements acquisition, Sect. 3.5
- (iii) requirements analysis and concept formation, Sect. 3.6
- (iv) business process re-engineering rough sketching, Sect. 3.7
- (v) terminology, Sect. 3.8
- (vi) requirements modelling, Sect. 3.9
- (vii) requirements model verification, Sect. 3.10
- (viii) requirements model validation, Sect. 3.11
- (ix) requirements feasibility and satisfiability analysis, Sect. 3.12
- and (x) requirements theory formation. Sect. 3.13

stages. Documents emerge from each of these stages.

Correspondence between Items 1–5 and Items (i–x) are as for corresponding domain stages and documents.

1.8 Analysis Documents

“SLIDE 124”

1.8.1 Verification, Model Checks and Tests

Characterisation 18 (Analysis) By analysis we mean a process which results in a document and which analyses another document: a domain description, a requirements prescription, or a software design, and where the analysis is either a verification (in the sense of formally proving a property), or a model check (in the sense of writing another, mechanically analysable, document which “models” the former and checks whether it possesses a given property), or a formal (or even informal) test (in the sense of subjecting the former document to a form of “execution” to observe whether that execution yields a given result). ■

1.8.2 Concept Formation

“SLIDE 125”

Yet there is also another form of analysis. One that results in the analysing engineer forming a concept.

Characterisation 19 (Concept Formation) By concept formation we mean an analysis process in which the analysing engineer from analysed phenomena or analysed concrete concepts form a concept, respectively a “more” abstract, i.e., less concrete concept. ■

1.8.3 Domain Analysis Documents “SLIDE 126”

Chapter 2 covers domain engineering in general and Sects. 2.6 and 2.10–2.13 covers domain analysis in particular.

Stages (iii, vii, viii, ix) listed in Sect. 1.7.1 are analytic. They result in the following kinds of documents:

- | | |
|---|----------------|
| 1 domain analysis (and concept formation) | see Sect. 2.6 |
| 2 domain model verification, | see Sect. 2.10 |
| 3 domain model validation, | see Sect. 2.11 |
| 4 and domain theory formation. | see Sect. 2.13 |

1.8.4 Requirements Analysis Documents “SLIDE 127”

Chapter 3 covers domain engineering in general and Sects. 3.6 and 3.10–3.13 covers requirements analysis in particular.

Stages (iii, vii, viii, ix, x) listed in Sect. 1.7.2 are analytic. They result in the following kinds of documents:

- | | |
|--|------------|
| 1 requirements analysis (and concept formation), | Sect. 3.6 |
| 2 requirements model verification, | Sect. 3.10 |
| 3 requirements model validation, | Sect. 3.11 |
| 4 requirements feasibility and satisfiability, | Sect. 3.12 |
| 5 and requirements theory formation. | Sect. 3.13 |

1.9 Descriptions, Prescriptions, Specifications “SLIDE 128”**1.9.1 Characterisations**

We have, so far, used the terms descriptions, prescriptions and specifications — and we shall continue to use these terms — with the following meanings.

(A) *Descriptions* are of “what there is”, that is, descriptions are, in this book, of domains, “as they are”;

(B) *Prescriptions* are of “what we would like there to be”, that is, prescriptions are, in this book, of requirements to software; and

(C) *Specifications* are of “how it is going to be”, that is, specifications are, in this book, of software.

1.9.2 Reiteration of Differences “SLIDE 129”

Descriptions are intended to state objective facts, i.e., are *indicative*. Prescriptions are intended to state commonly supposed and assumed to exist facts, i.e., are *putative* which we here take to be the same as *optative*: expressive of wish or desire. Specifications are intended to be expressive of a command, not to be avoided or evaded, i.e., are *imperative*.

“slide 130”

Descriptions are intended to state objective facts, i.e., are *indicative*. Prescriptions are intended to state commonly supposed and assumed to exist facts, i.e., are *putative* which we here take to be the same as *optative*: expressive of wish or desire. Specifications are intended to be expressive of a command, not to be avoided or evaded, i.e., are *imperative*.

- (i) Software shall satisfy requirements.
- (ii) Requirements defines properties of software.
- (iii) Requirements must be commensurate with “their domain”; that is, requirements must satisfy all the properties of the domain insofar as these have not been re-engineered.
- (iv) Requirements prescriptions denote requirements models.
- (v) Requirements models are not the software, only abstractions of software.
- (vi) Requirements models are computable adaptations of subsets of domain models.
- (vii) Domains satisfy a number of laws.
- (viii) Domain laws should be expressed by or derivable from domain descriptions.
- (ix) Domain descriptions denote domain models.
- (x) Domain models are not the domain, only abstractions of domains.

1.9.3 Rôle of Domain Descriptions

“SLIDE 132”

Domain descriptions for common computing system (colloquially: IT) applications relate to requirements prescriptions and software specifications (incl. code) as physics relate to classical engineering artifacts: (a) electricity, plasma physics, etc., relate to electronics; (b) mechanics, aerodynamics, etc., relate to aeronautical engineering; (c) nuclear physics, thermodynamics, etc., relate to nuclear engineering; etcetera.

Domain engineering relate to IT applications as follows: (d) transport domains to software (engineering) for road, rail, shipping and air traffic applications; (e) financial service industry domains to software (engineering) for banking, stock trading; portfolio management, insurance, credit card, etc., applications; (f) market trading (“the market”) domains to software (engineering) for consumer, retailer, wholesaler, supply chain, etc., applications (aka “e-business”); etcetera.

The Sciences of Human and Natural Domains¹⁸

The ‘Human Domains’

The domains for which most software systems are at play are — what we shall call — the human domains of financial service industries banks, insurance companies, stock

¹⁸ “SLIDE 134”

“slide 131”

“slide 133”

(etc.) trading brokers, traders, exchanges, etcetera; transportation industries roads, rails, shipping and air traffic; “the market” of consumers, retailers, wholesalers, product originators, and their distribution chains; etcetera,

*The Natural Sciences*¹⁹

In contrast the natural sciences includes physics: classical mechanics: statics, kinematics, dynamics, continuum mechanics: solid mechanics and fluid mechanics, mechanics of liquids and gases: hydrostatics, hydrodynamics, pneumatics, aerodynamics, and other fields; electromagnetism, relativity, thermodynamics and statistical mechanics, quantum mechanics, etcetera

“SLIDE 136”

The above listing is of disciplines within the natural sciences. It is not to be confused with a listing of research areas such as: condensed matter physics, atomic, molecular, and optical physics, high energy/particle physics, astrophysics and physical cosmology, etc.

*Research Areas of the Human Domains*²⁰

To establish a domain description for an area within the human domain — for which there was no prior domain description — is a research undertaking — just as it is for establishing a domain description for an area within the domain of natural sciences. There are thus as many²¹ human domain research areas as there are reasonably clearly separable such areas within the human domain.

Rôle of Domain Descriptions — Summarised²²

That then is the rôle of domain descriptions to gain understanding, through research, and, independently, to obtain the right software: software that meet client expectations.

1.9.4 Rôle of Requirements Prescriptions

“SLIDE 139”

A main rôle of a requirements prescription is to prescribe “the machine” !

The Machine

Characterisation 20 (Machine) By ‘the machine’ we shall mean a combination of hardware and software. ■

¹⁹ “SLIDE 135”

²⁰ “SLIDE 137”

²¹ and we think: exciting research areas

²² “SLIDE 138”

Machine Properties

The purpose of developing a requirements prescription is to prescribe properties of a machine.

1.9.5 Rough Sketches

“SLIDE 140”

Characterisation 21 (Rough Sketch) By a *rough sketch* we mean an informal text which does not claim to be consistent or complete, and which attempts, perhaps in an unstructured manner, to outline a phenomenon or a concept. ■

Rough sketches are useful “starters” towards narratives, and are used in acquired domain or requirements knowledge, and in outlining business processes and re-engineered such.

We refer to the rough sketch example of Sect. F.2 on page 249.

1.9.6 Narratives

“SLIDE 141”

Characterisation 22 (Narrative) By a *narrative* we mean an informal text which is structured, which is claimed consistent and relative complete, and which informally defines a phenomenon or a concept. ■

Narratives will be our main “work horse”, our chief means, at communicating domain descriptions and requirements prescriptions to all stakeholders.

We refer to the narrative example of Page 249 of Sect. F.3.1 on page 249.

Characterisation 23 (Annotation) By an *annotation* we mean an informal text which is structured so as to follow, usually line-by-line a formal (mathematical) text which it aims at explaining to a lay reader not familiar with the mathematical formulas. ■

We usually mandate that all formulas be annotated. But we do not mandate a specific “formal” way of structuring the annotations.

We refer to the annotations example of Page 250 of Sect. F.3.1 on page 249 (which annotates the formalisation of Page 250 of Sect. F.3.1 on page 249).

1.10 Software

“SLIDE 145”

1.10.1 What is Software ?

Characterisation 24 (Software) By software we understand: a set of documents: the domain development (incl. verification and validation) documents, the requirements development (incl. verification and validation) documents, and the software design development (incl. verification) documents. ■

1.10.2 Software is Documents !

“SLIDE 146”

Domain Documents

The domain development documents include the informative documents and the documents which record stakeholder identification and relations, domain acquisition, domain analysis and concept formation, rough sketches of the business (i.e., domain) processes, terminologies, domain description, domain verification (incl. model check and test), domain validation and domain theory formation.

Requirements Documents²³

The requirements development documents include the informative documents and the documents which record stakeholder identification and relations, requirements acquisition, requirements analysis and concept formation, rough sketches of the re-engineered business (i.e., new, revised domain) processes, terminologies, requirements description, requirements verification (incl. model check and test), requirements validation and requirements theory formation.

Software Design Documents²⁴

And the software design development documents include the informative documents, the documents which record architectural designs (“how derived from requirements”) and verifications (incl. model checks and tests), component designs and verifications (incl. model checks and tests), module designs and verifications (incl. model checks and tests), code designs and verifications (incl. model checks and tests), and the actual executable code documents.

Sections 2.15, 3.15 and 4.13 shall detail the above documents.

Software System Documents²⁵

Characterisation 25 (Software System) By a *software[-based] system* we shall understand a set of software system documents (see below) as well as the hardware, the IT equipment for which the software is oriented: computers, their peripherals, data communication equipments, etcetera. ■

The *software system documents* include: the actual executable code documents, as well as ancillary documents: demonstration (i.e., demo) manuals, training manuals, installation manuals, user manuals, maintenance manuals, and development and maintenance logbooks.

“slide 150”

²³ “SLIDE 147”

²⁴ “SLIDE 148”

²⁵ “SLIDE 149”

1.11 Informal and Formal Software Development “SLIDE 151”

In this book we shall advocate a combination of informal and formal development. And in this section we shall use the term specification (specify) to also cover description (describe) and prescription (prescribe), etc.

1.11.1 Characterisations

Informal Development

Characterisation 26 (Informal Development) By *informal development* we understand, in this book, a software development which does not use formal techniques, see below; instead it may use UML and an executable programming language. ■

Formal Development²⁶

Characterisation 27 (Formal Development) By *formal development* we mean, in this book, a software development which uses one or more formal techniques, see below, and it may then use these in a spectrum from systematically via rigorously to formally. ■

For characterisations of systematically, rigorously and formally we refer to characterisations below.

Formal Software Development

Characterisation 28 (Formal Software Development Technique) By a *formal development technique* we mean, in this book, a software development in which specifications are expressed in a formal language, that is, a language with a formal syntax so that all specifications can be judged well-formed or not; a formal semantics so that all well-formed specifications have a precise meaning; and a (relatively complete) proof system such that one may be able to reason over properties of specifications or steps of formally specified developments from a more abstract to a more concrete step. Additionally a formal technique may be a calculus which allows developers to calculate, to refine “next”, formally specified development steps from a preceding, formally specified step. ■

Formal techniques are usually supported by software tools that check for syntactic and helps check for semantic correctness.

Examples of formal techniques, sometimes referred to as formal methods, are Alloy [109], ASM (Abstract State Machines) [167], B and event-B [2, 46], DC (Duration Calculus) [203], MSC and LSC (Message and Live Sequence

²⁶ “SLIDE 152”

Charts) [90, 105, 106, 107], Petri Nets [155, 111, 164, 163, 165], Statecharts [86, 87, 89, 91, 88], RAISE (Rigorous Approach to Industrial Software Engineering) [27, 28, 29, 75, 77, 76], TLA+ (Temporal Logic of Actions) [118, 119, 137], VDM (Vienna Development Method) [38, 39, 72] and Z [180, 181, 201, 96]. The EATCS²⁷ Monograph [37] arose from [167, 46, 63, 145, 76, 137, 96] and covers ASM, B and event-B, CafeOBJ, CASL, DC, RAISE, TLA+, VDM and Z.

This book will, in Vol. II, primarily feature the RAISE approach and thus use its Specification Language RSL. For a more comprehensive introduction to formal techniques we refer to [27, 28, 29].

Systematic (Formal) Development !²⁸

Characterisation 29 (Systematic (Formal) Development) By a *systematic use of a formal technique* we mean, in this book, a software development which which formally specifies whenever something is specified, but which does not (at least only at most in a minor of cases) reason formally over steps of development. ■

Rigorous (Formal) Development !²⁹

Characterisation 30 (Rigorous (Formal) Development) By a *rigorous use of formal techniques* we mean, in this book, a software development which which formally specifies whenever something is specified, and which formally express (some, if not all) properties that ought be expressed, but which does not (at least only at most in a minor number of cases) reason formally over steps of development, that is, verify these to hold, either by theorem proving, or by model checking, or by formally based tests. ■

Formal (Formal) Development !³⁰

Characterisation 31 (Formal (Formal) Development) By *formal use of a formal techniques* we mean, in this book, a software development which which formally specifies whenever something is specified, which formally expresses (most, if not all) properties that ought be expressed, and which formally verifies these to hold, either by theorem proving, or by model checking, or by formally based tests. ■

²⁷ EATCS: European Association for Theoretical Computer Science

²⁸ "SLIDE 154"

²⁹ "SLIDE 155"

³⁰ "SLIDE 156"

1.11.2 Recommendations

“SLIDE 157”

This book advocates that software development be pursued according to the triptych paradigm, and that the phases, stages and steps, as outlined in Chaps. 2–4, be pursued in a combination of both informal and formal descriptions, prescriptions and specifications, in a systematic to rigorous fashion.

1.12 Entities, Functions, Events and Behaviours

“SLIDE 160”

So what is it that we describe, prescribe and specify, informally or formally? The answer is: entities, operations, events and behaviours. We shall, in this section, survey these concepts of domains, requirements and software designs. In the domain we observe phenomena. “SLIDE 161” From usually repeated such observations we form (immediate, abstract) concepts. We may then “lift” such immediate abstract concepts to more general abstract concepts. Phenomena are manifest. They can be observed by human senses (seen, heard, felt, smelled or tasted) or by physical measuring instruments (mass, length, time, electric current, thermodynamic temperature, amount of substance, luminous intensity). Concepts are defined.

We shall analyse phenomena and concepts according to the following simple, but workable classification: *entities*, *functions* (over entities), *events* (involving changes in entities, possibly as caused by function invocations, i.e., *actions*, and/or possibly causing such), and *behaviours* as (possibly sets of) sequences of actions (i.e., function invocations) and events.

1.12.1 Entities

“SLIDE 163”

Characterisation 32 (Entity) By an *entity* we mean something we can point to, i.e., something manifest, or a concept abstracted from, such a phenomenon or concept thereof. ■

Entities are either atomic or composite. The decision as to which entities are considered atomic or composite is a decision solely taken by the describer.

Atomic Entities³¹

Characterisation 33 (Atomic Entity) By an *atomic entity* we intuitively understand an entity which “cannot be taken apart” (into other, the sub-entities) and which possess one or more attributes. ■

³¹ “SLIDE 164”

Attributes — Types and Values.³²

With any entity we can associate one or more attributes.

Characterisation 34 (Attribute) By an *attribute* we understand a pair of a **type** and a **value**. ■

“SLIDE 166”

Example 2 (Atomic Entities)

Entity: Person		Entity: Bank Account	
Type	Value	Type	Value
Name	Dines Bjørner	number	212 023 361 918
Weight	118 pounds	balance	1,678,123 Yen
Height	179 cm	interest rate	1.5 %
Gender	male	credit limit	400,000 Yen

“Removing” an attribute from an entity destroys its “entity-hood”.

Composite Entities³³

Characterisation 35 (Composite Entity) By a *composite entity* we intuitively understand an entity (i) which “can be taken apart” into sub-entities, (ii) where the composition of these is described by its *mereology*, and (iii) which, apart from the attributes of the sub-entities, further possess one or more attributes. ■

Sub-entities are entities.

Mereology³⁴

Characterisation 36 (Mereology) By *mereology* we understand a theory of part-hood relations. That is, of the relations of part to whole and the relations of part to part within a whole. ■

The term mereology seems to have been first used in the sense we are using it by the Polish mathematical logician Stanisław Lesniewski [130, 141, 183, 184, 190].

³² “SLIDE 165”

³³ “SLIDE 167”

³⁴ “SLIDE 168”

Composite Entities — Continued³⁵

Example 3 (Transport Net, A Narrative)

Entity: Transport Net		
Subentities: Segments Junctions		
Mereology: “set” of one or more $s(\text{egment})$ s and “set” of two or more $j(\text{unction})$ s such that each $s(\text{egment})$ is delimited by two $j(\text{unction})$ s and such that each $j(\text{unction})$ connects one or more $s(\text{egments})$		
Attributes		
	Types:	Values:
	Multimodal	Rail, Roads
	Transport Net of	Denmark
	Year Surveyed	2006

“slide 170”

“slide 171”

To put the above example of a composite entity in context we give an example of both an informal narrative and a corresponding formal specification:

Example 4 (Transport Net, A Formalisation) A transport net consists of one or more segments and two or more junctions. With segments [junctions] we can associate the following attributes: segment [junction] identifiers, the identifiers of the two junctions to which segments are connected [the identifiers of the one or more segments connected to the junction], the mode of a segment [the modes of the segments connected to the junction]

“slide 172”

type

N, S, J, Si, Ji, M

value

$$\begin{aligned} \text{obs_Ss}: N \rightarrow \mathbf{S\text{-set}}, & \quad \text{obs_Js}: N \rightarrow \mathbf{J\text{-set}} \\ \text{obs_Si}: S \rightarrow \text{Si}, & \quad \text{obs_Ji}: J \rightarrow \text{Ji} \\ \text{obs_Jis}: S \rightarrow \mathbf{Ji\text{-set}}, & \quad \text{obs_Sis}: J \rightarrow \mathbf{Si\text{-set}} \\ \text{obs_M}: S \rightarrow M, & \quad \text{obs_Ms}: J \rightarrow \mathbf{M\text{-set}} \end{aligned}$$
axiom

$$\begin{aligned} \forall n:N \cdot \mathbf{card} \text{ obs_Ss}(n) \geq 1 \wedge \mathbf{card} \text{ obs_Js}(n) \geq 2 \\ \forall n:N \cdot \mathbf{card} \text{ obs_Ss}(n) \equiv \mathbf{card} \{ \text{obs_Si}(s) \mid s:S \cdot s \in \text{obs_Ss}(n) \} \\ \forall n:N \cdot \mathbf{card} \text{ obs_Js}(n) \equiv \mathbf{card} \{ \text{obs_Ji}(c) \mid j:J \cdot j \in \text{obs_Js}(n) \} \dots \end{aligned}$$
type

Nm, Co, Ye

value

$$\text{obs_Nm}: N \rightarrow \text{Nm}, \text{obs_Co}: N \rightarrow \text{Co}, \text{obs_Ye}: N \rightarrow \text{Ye}$$

Si, Ji, M, Nm, Co, Ye are not entities. They are names of attribute types and, as such, designate attribute values. N is composite, S and J are considered atomic .

³⁵ “SLIDE 169”

States³⁶

Characterisation 37 (State) By a domain *state* we shall understand a collection of domain entities chosen by the domain engineer. ■

The pragmatics of the notion of state is that states are recurrent arguments to functions and are changed by function invocations.

Formal Modelling of Entities³⁷

How do we model entities ? The answer is: by selecting a name for the desired “set”, that is, type of entities; by defining that type to be either an abstract type, i.e., a sort,

type
A

or a concrete type, i.e., with defined, concrete values.

type
A = Type_Expression

Values of the type are then expressed as:

value
a:A

“slide 175”

As our main support example unfolds in Vol. II we shall illustrate sorts with their observer functions and concrete types over either basic types (Booleans, integers, natural numbers, reals, etc., or over composite types (sets, Cartesians, records, lists, maps, functions). Appendix Sect. S.1 (Pages 405–408) gives a terse introduction to the type system of our main formal specification language RSL.

1.12.2 Functions

“SLIDE 176”

Characterisation 38 (Function) By a *function* we shall understand something which when *applied* to what we shall call *arguments* (i.e., entities) *yield* some entities called the *result* of the function (application). ■

Actions

Characterisation 39 (Action) By an *action* we shall understand the same thing as applying a state-changing function to its arguments (including the state). ■

³⁶ “SLIDE 173”

³⁷ “SLIDE 174”

Functions — Resumed³⁸

The observer functions of the formal example above are not the kind of functions we are (later) seeking to identify in domains and requirements. These observer functions are mere technicalities: needed, due to the way in which we formalise — and are deployed in order to express sub-entities, mereologies and attributes.

Function Signatures³⁹

Characterisation 40 (Function Signature) By a *function signature* we mean the *name and type* of a function.

type

A, B, ..., C, X, Y, ..., Z

value

$f: A \times B \times \dots \times C \rightarrow X \times Y \times \dots \times Z$

The last line above expresses a schematic function signature. ■

Function Descriptions⁴⁰

Characterisation 41 (Function Description) By a function description we mean a function signature and something which describes the relationship between function arguments (the a:A's, b:B's, ..., c:C's and the x:X's, y:Y's, ..., z:Z's). ■

Example 5 (Well Formed Routes)

$P = J_i \times S_i \times J_i$ /* path: triple of identifiers */

$R' = P^*$ /* route: sequence of connected paths */

$R = \{ | r:R' \bullet wf_R(r) | \}$ /* subtype of R' : those r 's satisfying $wf_R(r)$ */

value

$wf_R: R' \rightarrow \mathbf{Bool}$

$wf_R(r) \equiv$

$\forall i:\mathbf{Nat} \bullet \{i, i+1\} \subseteq \mathbf{inds} \ r \Rightarrow \mathbf{let} \ (,,j_i')=r(i), (j_i'',,,)=r(i+1) \ \mathbf{in} \ j_i'=j_i'' \ \mathbf{end}$

The last line above describes the route wellformedness predicate. The meaning of the “(,,” and “,,)” is that the omitted path components “play no rôle” ■

³⁸ “SLIDE 177”

³⁹ “SLIDE 178”

⁴⁰ “SLIDE 179”

1.12.3 Events

"SLIDE 181"

Characterisation 42 (Event)

- An event can be characterised by
 - ★ a predicate, p and
 - ★ a pair of ("before") and ("after") of pairs of
 - states and
 - times:
 - $p((t_b, \sigma_b), (t_a, \sigma_a))$.
 - ★ Usually the time interval $t_a - t_b$
 - ★ is of the order $t_a \simeq \text{next}(t_b)$

Sometimes the event times coincide, $t_b = t_a$, in which case we say that the event is instantaneous. The states may then be equal $\sigma_b = \sigma_a$ or distinct !

We call such predicates as p for event predicates.

By an *event* we shall thus, to paraphrase, understand an instantaneous change of state not directly brought about by some explicitly willed action in the domain, but either by "external" forces. or implicitly as a non-intended result of an explicitly willed action.

Events may or may not lead to the initiation of explicitly issued operations.

Example 6 (Events) A 'withdraw' from a positive balance bank account action may leave a negative balance bank account. A bank branch office may have to temporarily stop actions, i.e., close, due to a bank robbery. ■

Internal events: The first example above illustrates an internal event. It was caused by an action in the domain, but was not explicitly the main intention of the "withdraw" function.

External events: The second example above illustrates an external event. We assume that we have not explicitly modelled bank robberies!

1.12.4 Behaviours

"SLIDE 184"

Simple Behaviours

Characterisation 43 (Simple Behaviour) By a *simple behaviour*

- we understand a sequence, q , of zero, one or more
 - ★ actions
 - ★ and/or events
 - ★ $q_1, q_2, \dots, q_i, q_{i+1}, \dots, q_n$
- such that the state
 - ★ resulting from one such action, q_i ,
 - ★ or in which some event, q_i , occurs,
- becomes the state in which the next action or event, q_{i+1} ,

"slide 182"

"slide 183"

- ★ if it is an action, is effected,
- ★ or, if it is an event, is the event state

“slide 185”

Example 7 (Simple Behaviours) The opening of a bank account, followed by zero, one or more deposits into that bank account, and/or withdrawals from the bank account in question, ending with a closing of the bank account. Any prefix of such a sequence is also a simple behaviour. Any sequence in which one or more events are interspersed is also a simple behaviour. ■

General Behaviours⁴¹

A *behaviour* is either a *simple behaviour*, or is a *concurrent behaviour*, or, if the latter, can be either a *communicating behaviour* or not (i.e., just a concurrent behaviour).

Concurrent Behaviours⁴²

Characterisation 44 (Concurrent Behaviour) By a concurrent behaviour we shall understand a set of behaviours (simple or otherwise). ■

“SLIDE 188”

Example 8 (Concurrent Behaviours) A set of simple behaviours, that may result from two or more distinct bank clients, each operating of their own, distinct, that is, non-shared accounts, forms a concurrent behaviour. ■

Communicating Behaviours⁴³

Characterisation 45 (Communicating Behaviour) By a *communicating behaviour* we shall understand a set of two or more behaviours where otherwise distinct elements (i.e., behaviours) share events. ■

“SLIDE 190”

Sometimes we do not model the behaviour from which external events are incident (i.e., “arrive”) or to which events emanate (i.e., “depart”). But such an environment is nevertheless a behaviour.

“SLIDE 191”

Example 9 (Communicating Behaviours) Consider a bank. To model that two or more clients can share the same bank account one could model the bank account as one behaviour and each client as a distinct behaviour. Let us assume that only one client can open an account and that only one client can

⁴¹ “SLIDE 186”

⁴² “SLIDE 187”

⁴³ “SLIDE 189”

close an account. Let us further assume that sharing is brought about by one client, say the one who opened the account, identifying the sharing clients. Now, in order to make sure that at most one client accesses the shared account at one one time (in any one “smallest” transaction interval) one may model “client access to account” as a pair of events such that during the interval between the first (begin transaction) and the second (end transaction) event no other client can share events with the bank account behaviour. Now the set of behaviours of the bank account and one or more of the client behaviours is an example of a communicating behavior. ■

Formal Modelling of Behaviours⁴⁴

Communicating behaviours, the only really interesting behaviours, can be modelled in a great variety of ways: from set-oriented models in B [2, 47], RSL [75, 77, 27, 28, 29, 74, 35], VDM [38, 39, 72, 71], or Z [180, 181, 201, 96, 95], to models using for example CSP [98, 173, 175, 99], (as for example “embedded” in RSL [75].), or, to diagram models using, for example, Petri nets [111, 155, 164, 163, 165], message [105, 106, 107] or live sequence charts [57, 90, 113], or state-charts [86, 87, 89, 91, 88].

1.12.5 Discussion

“SLIDE 193”

The main aim of Sect. 1.12 is to ensure that we have a clear understanding of the modelling concepts of entities, functions, events and behaviours. To “reduce” the modelling of phenomena and concepts to these four kinds of phenomena and concepts is, of course, debatable. Our point is that it works, that further classification, as is done in for example John F. Sowa’s [179], is not necessary, or rather, is replaced by how we model attributes of, for example, entities, and how we model facets (Sect. 2.9.1, next chapter).

1.12.6 Functions, Events and Behaviours as Entities

Review of Entities

In the example of Chap. F we identify the following as being entities: (i) nets (Item 6 on page 250), (ii) links (Item 5 on page 249), (iii) hubs (Item 5), (iv) insert commands (Item 18 on page 255), (v) remove commands (Item 19 on page 255), (vi) time (Item 31 on page 264), (vii) time intervals (Item 35 on page 265), (viii) vehicles (Item 42 on page 265), (ix) positions (Item 43 on page 265) and (x) traffic (Item 44 on page 266).

It may surprise some that we designate the insert and remove commands as entities. They are certainly of conceptual nature, but can be given manifest

⁴⁴ “SLIDE 192”

representations in the form of documents (that, for example order the building of a link and its eventual inclusion in the net).

It may surprise some that we designate time and time intervals as entities. They are certainly of conceptual and very abstract nature, but so is our choice.

It may surprise some that we designate positions as entities. They are certainly manifest: one can point to a position.

And it may finally surprise some that we designate traffics as entities. It is certainly manifest, and can be recorded, say by video-recording the traffic. So that is also our choice.

Functions as Entities

TO BE WRITTEN

Events as Entities

TO BE WRITTEN

Behaviours as Entities

TO BE WRITTEN

1.13 Domain vs. Operational Research Models “SLIDE 194”

1.13.1 Operational Research (OR)

Since World War II, as a result of research and application of what became known as OR models (OR for Operational Research), these have won a significant position also within the transportation infrastructure. But domain models are not OR models. OR models usually use classical applied mathematics: calculus ([partial] differential equations), statistics, probability theory, graph theory, combinatorics, signal analysis, theory of flows in networks, etcetera where domain engineering use formal specification languages emphasising applied mathematical logic and modern algebra.

1.13.2 Reasons for Operational Research Analysis “SLIDE 195”

OR models are established, that is, OR analysis is performed, for the following reasons: to solve a particular problem, usually a resource allocation and/or scheduling problem, but also, less often, the problem is one of taking advice: should an investment be made, should one form of resource be “converted” into another form, etc. Once solved the solver and the client knows how to best allocate and/or schedule the investigated resource or whether to perform

a certain kind of investment, etc. OR models typically do not themselves lead to software derived from the OR model, but sometimes results of OR analysis become constants in or parameters for otherwise independently developed software.

1.13.3 Domain Models

“SLIDE 196”

Domain models are usually established (i) to understand an area of a domain much wider than that analysable by current OR techniques, and sometimes (ii) for purposes of “deriving” appropriate requirements, and (iii) for implementing the right software. It has then turned out that in order to achieve items (i–iii) above one has to use the kind of mathematics shown in this book.

1.13.4 Domain and OR Models

“SLIDE 197”

But domain and OR modelling are not really that separated — as it may appear from the above. Oftentimes software (as well as hardware) design decisions must (or ought to) be based on OR analysis. The two kinds of modelling must still be pursued. But it is desirable that their scientists and engineers, i.e., that their practitioners, collaborate. Today they do not collaborate. Today only the domain engineers are aware of the existence of OR engineers.

1.13.5 Domain versus Mathematical Modelling

“SLIDE 198”

We could widen our examination of domain modelling versus OR modelling to domain modelling versus mathematical modelling, where the latter extends well beyond OR modelling to the modelling of physical and human made domains in its widest sense — such as also practiced by physicists, biologists, etc.

For OR modelling as well as mathematical modelling we can say that domain modelling currently lacks the formal techniques offered by the former.

But we are digressing !

1.14 Summary

“SLIDE 199”

The exercises of this chapter, see next, reveal the essence of this chapter: (i) the ‘trptych paradigm’ (Sect. 1.2); (ii) the ‘trptych phases of software engineering’ (Sect. 1.3); (iii) the ‘stages’ and ‘steps’ of software development (Sect. 1.4); (iv) the three classes of development documents (Sect. 1.5); (v) the detailed nature of 16 kinds of ‘informative documents’ (Sect. 1.6); (vi) the concepts of ‘modelling documents’ (Sect. 1.7); (vii) the concepts of ‘analysis documents’ (Sect. 1.8); (viii) the concepts of ‘descriptions, prescriptions’ and ‘specifications’ (Sect. 1.9); (ix) the concept of ‘software’ (Sect. 1.10); (x) the

“slide 200”

concepts (Sect. 1.11) of ‘informal development’, ‘formal development’, ‘informal and formal development’, ‘formal software development technique’, ‘systematic development’, ‘rigorous development’ and ‘formal development’; (xi) the concepts of ‘entities’, ‘functions’, ‘events’ and ‘behaviours’ (Sect. 1.12); and (xii) the concepts of ‘operational research’ versus those of ‘domain models’ (Sect. 1.13).

1.15 Exercises

Exercise 1. The Triptych Paradigm: Rehearse the text of the triptych dogma so that you can express it, “by heart”, precisely.

Solution 1 Vol. II, Page 431, suggests a way of answering this exercise.

Exercise 2. The Triptych Phases of Software Development: Rehearse the text of the triptych phases of software development so that you can express it, “by heart”, precisely.

Solution 2 Vol. II, Page 431, suggests a way of answering this exercise.

Exercise 3. Phases, Stages and Steps of Software Development: Explain the concepts of software development phase, stage and step.

Solution 3 Vol. II, Page 431, suggests a way of answering this exercise.

Exercise 4. Development Documents:

Enumerate the three kinds of development documents outlined in this chapter, that is, express them, “by heart”, precisely.

Solution 4 Vol. II, Page 432, suggests a way of answering this exercise.

Exercise 5. Enumeration of Informative Documents:

Enumerate, as best as you can, the 16 kinds of informative documents outlined in this chapter — express them, “by heart”, as precisely as you can.

Solution 5 Vol. II, Page 432, suggests a way of answering this exercise.

Exercise 6. Descriptions, Prescriptions, Specifications:

In which contexts are the terms descriptions, prescriptions and specifications used in this book, such as proclaimed in this chapter.

Solution 6 Vol. II, Page 432, suggests a way of answering this exercise.

Exercise 7. Software:

Explain what the term ‘software’ covers, such as characterised in this chapter. That is: list as you can best remember, the names of the documents that together “make up” software.

Solution 7 Vol. II, Page 432, suggests a way of answering this exercise.

Exercise 8. Informal and Formal Software Development:

Explain, as precisely as possible, the terms ‘informal development’, ‘formal development’, ‘informal and formal development’, ‘formal software development technique’, ‘systematic development’, ‘rigorous development’ and ‘formal development’ such as characterised in this chapter.

Solution 8 Vol. II, Page 434, suggests a way of answering this exercise.

Exercise 9. Entities and States, Functions and Actions, Events and Behaviours:

Please define, as close to the characterisations of this chapter the notions of entities, states, functions, actions, events and behaviours.

Solution 9 Vol. II, Page 434, suggests a way of answering this exercise.

Exercise 10. Mereology, Atomic and Composite Entities:

Please define, as close to the characterisations of this chapter the notion of entities: atomic and composite. Focus, in particular, on the issue of the attributes of composite entities, their sub-entities and their mereology.

Solution 10 Vol. II, Page 435, suggests a way of answering this exercise.

“slide 201”

