1

The Triptych Paradigm

- The **prerequisite** for studying this chapter is that you have at least some introductory level programming skills, as, for example, obtained through a year of `Java` or `C#` programming.
- The **aims** are to introduce the basic ideas of *domain engineering*, *requirements engineering* and *software design* as they relate to one another, to introduce the concept of *separation of concerns* as represented here by the concepts of *phases*, *stages* and *steps* of development, and thus to introduce the concept of the *triptych software development process model.*
- The **objective** is to make the reader a professional software engineer with respect to understanding the crucial phases, stages and steps of software development.
- The **treatment** is precise but informal.

1.1 Delineations of Software Engineering

We give two sets of characterisations of the field of software engineering. One set of characterisations is taken from the literature. The other (a singleton) set is our definition.

1.1.1 "Old" Delineations

The term "software engineering" seems to have many meanings. We shall bring in some of the characterisations that are given in previous textbooks as well as from elsewhere.

Friedrich L. Bauer [257], 1968

Software engineering is the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works efficiently on real machines.

So we are left to find out what is meant by engineering principles. These "engineering principles" cannot just be those of conventional engineering as we think that the engineering of software is radically different from other engineerings. Conventional engineering builds on the laws of physics. Software engineering builds on mathematics, notably algebra and logic.

Ian Sommerville [338], 1980–2000

> Software engineering is an engineering discipline which is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

In this definition, there are two key phrases:

> (1) *Engineering discipline:* Engineers make things work. They apply theories, methods and tools where these are appropriate but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods to support them. Engineers also recognise that they must work to organisational and financial constraints so they look for solutions within these constraints.

> (2) *All aspects of software production:* Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production.

We are getting some engineering principles unveiled, albeit of the conventional kind.

IEEE Std. 610.12–1990 [178]

The IEEE's Standard Glossary of Software Engineering Terminology:

> Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Again, a very conventional engineering characterisation.

David Lorge Parnas

> Software engineering is defined as the multi-person construction of multi-version software.

This is, of course, not all that Parnas has to say about software engineering. As much of his other musings this one is cogent.

Shari Lawrence Pfleeger [275], 2001

Pfleeger [275] (Page 2) has an indirect characterisation:

> As software engineers, we use our knowledge of computers and computing to help solve problems ... identification of problems and of when a computing solution may be appropriate, further analysis of such problems, and synthesis of solutions using method principles, techniques and tools, are ingredients of software engineering.

We are not getting much closer to our claimed difference between conventional engineerings and software engineering. Pfleeger's characterisation is OK, but insufficient.

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli [121], 2002

> Software engineering is the field of computer science that deals with the building of software systems that are so large or so complex that they are built by a team or teams of engineers.

This definition hides the real content in its reference to computer (including computing) science, i.e., the mathematical discipline upon which the software engineers work. But, as for Parnas' characterisation, the Ghezzi/Jazayeri/-Mandrioli characterisation emphasises scale.

Accreditation Board for Engineering and Technology (ABET)

ABET (www.abet.org) gives a definition of 'engineering' which some software engineering authors refer to:

> Engineering is the profession in which a knowledge of the mathematical and natural sciences gained by study, experience and practice is applied with judgment to develop ways to utilize, economically, the materials and forces of nature for the benefit of mankind.

We take exception, in parts, to the software engineering use of the ABET characterisation: First, to us, software engineering (in almost all its activities) does not rely on laws of the natural sciences (but almost exclusively only on mathematics). Second, *the materials and forces of nature* must therefore be rephrased into *mathematics.* Third, *for the benefit of mankind* is just a (as of the year 2005) politically correct utterance — so we do not include it in our consideration of what software engineering is.

Hans van Vliet [369], 2000

Pages 6–8 of Hans van Vliet's delightful work [369] details the following software engineering facets:

- *Software engineering concerns the construction of large programs.*
- *The central theme is mastering complexity.*
- *Software evolves.*
- *The efficiency with which software is developed is crucial.*
- *Regular co-operation between people is an integral part of programming-in-the-large.*
- *The software has to support its users effectively.*
- *Software engineering is a field in which members of one culture create artifacts on behalf of members of another culture.*

We quite like van Vliet's characterisation.

1.1.2 Our View: What Is Software Engineering?

Many of the above characterisations are relevant. Some characterise software engineering by how it proceeds, others by what it does. We prefer the latter style of characterisation. In order, however, to emphasise a number of new aspects of the software engineering approach that we shall be propagating in these volumes, our delineation combines both the 'how' and the 'what' styles.

We shall therefore characterise the concept of 'software engineering' as follows:

- Software engineering is the establishment and use of sound methods for the efficient construction of efficient, correct, timely and pleasing software that solves the problems such as users identify them.
- Software engineering extends the field of computing science to include also the concerns of building of software systems that are so large or so complex that they necessarily are built by a team or teams of engineers.
- Software engineering is the profession in which a knowledge of mathematics, gained by study, experience and practice, is applied, with judgment to develop ways of exploiting mathematics to (i) understand the problem domain, (ii) the problem and (iii) to develop computing systems, especially software solutions, to such problems as are conveniently solved by computing.
- Software engineering thus consists of (i) domain engineering (in order to understand the problem domain), (ii) requirements engineering (in order to understand the problem and possible frameworks for their solution) and (iii) software design (in order to actually implement desired solutions).

In the next section we shall examine these three concepts: domain engineering, requirements engineering and software design.

1.2 The Triptych of Software Engineering

Before some specific software can be designed and coded, we must understand the requirements that this software must fulfill.

Characterisation. By *design of software* we mean the process, and the documents resulting from the process, of implementing the software. That is, we mean in the narrow sense of conceiving of and expressing (i.e., specifying) these documents — in stages and steps — eventually in some executable programming language. Software design thus specifies *how* executions may proceed. ∎

Characterisation. By *requirements prescription* we mean the process, and the documents resulting from the process, of acquiring, analysing and writing down, in some language, *what* the software — to be designed — is expected to do. ∎

Before requirements can be written down, we must understand the application domain for which the software is to be developed.

Characterisation. By *domain description* we mean the process, and the documents resulting from the process, of acquiring, analysing and writing down, in some language, a model of the application domain *as it is* — void of any reference to requirements to any software for that domain, let alone references to such software. ∎

So, from descriptions of the application domain we construct prescriptions of the requirements; and from prescriptions of the requirements we design the software, i.e., construct specifications of software. Ideally speaking we would wish to proceed from describing the application domain, via prescribing the requirements, to implementing the software. Actual life sometimes forces us, and always permits us, to iterate between these three phases of software development.

1.2.1 On Universes of Discourse and Domains

Above we have used the term "application domain" without explaining what we mean by that term. We shall now explain that term as well as the more general term 'universe of discourse' and the simpler term 'domain'.

Characterisation. By a *universe of discourse* we shall understand anything that can be spoken about. ∎

Example 1.1 *Universes of Discourse:* We shall take the view here that there are basically three classes of universes of discourse:

(i) The definite, but singleton class of software engineering as an intellectual concept, i.e., software development in general and programming in particular. So, domain engineering, requirements engineering and software design could each, or as a whole, be a universe of discourse. These volumes take this intellectual concept of software engineering as its universe of discourse. As an intellectual concept the software engineering universe of discourse is not an application domain.

(ii) The indefinite class of "things" to which computing may be applied, i.e., application domains (see next).

(iii) The infinite class of anything else that does not satisfy the above characterisations. Examples are: philosophy, politics, poetry, etc.[1]   ∎

Characterisation. By an *application domain* we shall understand anything to which computing may be applied.   ∎

Example 1.2 *Application Domains:* We shall take the view that there are basically three classes of application domains:

(i) The class of applications which can be characterised as supporting the teaching or study of a subject field: educational or training software, respectively experimental software for theorem proving, or the like.

(ii) The class of applications which can be characterised as supporting the development of computing systems themselves: compilers, operating systems, database management systems, data communication systems, etc.

(iii) And the class of applications which can be characterised as not supporting the development of computing systems themselves, but that of business, or industry software.

It is basically the last class of these application domains that are of interest in this volume. We relegate to specific textbooks the treatment of special principles, techniques and tools for the development of software for application domains (i) and (ii). Classes (i) and (ii), in a sense, overlap. Class (i) is perhaps better viewed as a knowledge engineering topic, while class (ii) is conventionally seen as a systems software topic. Viewed these ways, class (iii) is then normally seen as an "end-user", i.e., "customer software" topic.   ∎

Characterisation. By a *domain* we mean an application domain.   ∎

---

[1] The reader may observe two things: Our inability to make precise that to which computing cannot be applied, and our belief that philosophy, politics, poetry, etc., belong to that class. When we claim that computing does not apply to philosophy, politics, poetry, etc., we mean that crucial philosophical thoughts, political ideas and poetic utterings cannot, in our mind, be the result of computations.

That is, the two terms "application domain" and "domain" are taken to be synonymous.

Example 1.3 *Domains:* We continue our exemplification of (application) domains — of the third class mentioned just above.

(iii.1) The applications of software within the transportation sector: Railways, airlines, shipping, public and private road transport (buses, taxis, trucks, automobiles in general), etc., individually define application domains, and together "define"[2] *transportation* as a domain.

(iii.2) The applications of software within the financial services sector: banks, insurance companies, securities trading (stock and bond exchanges, traders, brokers), portfolio and investment management, venture capital companies, etc., individually define application domains, and together "define" the *financial service industry* as a domain.

(iii.3) The applications of software within the healthcare sector: hospitals, family doctors (i.e., private, practicing physician), pharmacies, community nurses, retraining and convalescent clinics, the public health authorities, etc., individually define application domains, and together "define" *healthcare* as a domain.

(iii.4) The applications of software within the machining (metal-working) manufacturing sector: marketing, sales and orders department, design research and development, production planning department, the production "floor" with its "input" production parts inventory and its "output" products warehouse, workers, managers, etc., individually define application domains, and together "define" *machining (metal-working) manufacturing* as a domain. ∎

1.2.2 Domain Engineering

We bring in an overview of domain engineering. Part IV, Chaps. 8–16, bring in details!

General

In this section we shall give a brief characterisation of what we mean by domain engineering.

Characterisation. (I) By a *domain description* we shall understand a description of a domain, that is, something which describes observable phenomena of the domain: entities, functions over these, events and behaviours. ∎

Characterisation. (II) By *domain description* we shall also mean the process of domain capture, analysis and synthesis, and the document which results from that process. ∎

---

[2] Here our use of "define" indicates that we are not formally defining the subject term. We are merely giving a rough characterisation.

Characterisation. By *domain engineering* we mean the engineering of domain descriptions, that is, of their development: (i) from domain capture and analysis (ii) via synthesis, i.e., the domain description document itself, (iii) to its validation with stakeholders and its possible theory development.   ∎

So what does it mean to understand the application domain? To us it means that we have described it. That the description is consistent, i.e., does not give rise to contradictions, and that the description is relatively complete, i.e., does describe "all the things" needed to be described.

What Do We Expect from a Domain Description?

What must we expect from a domain description? We expect that it describes the application area *as it is.*

What a Domain Description Does Not Describe

To us "as it is" means that we have described it without any reference to requirements to any new computing system (i.e., software), let alone to any (implementation, etc.) of such a new computing system (i.e., software). The above was expressed in terms of *what a domain description does not contain.*

What a Domain Description Does Describe

So what does a domain description contain? To us a domain description contains: descriptions of the *phenomena* that can be observed, that can be physically sensed, in the domain, and descriptions of the *concepts*, i.e., the abstractions that these phenomena "embody".

Domain Phenomena and Concepts

What are the phenomena and concepts alluded to just above? To us these phenomena and concepts are such as: (i) entities, (ii) functions, (iii) events and (iv) behaviours. We overview these four categories of phenomena and concepts.

Entities

Entities are "things" that one can point to, things that typically become data "inside" a computer, things that are to have a *type* and a *value* (of that type).

Example 1.4 *Entities:* For a domain of harbours some typical entities are: ships, holding area(s) where ships may wait for a buoy or a quay position, buoys, quay positions and cargo storage areas. The harbour can be considered an entity composed from the above.

───── Informal Presentation: Entities: Types, Values and Observers ─────

   We shall later explain the notation now used:

type
   Harbour, Ship, HoldArea, Buoy, Quay, CSA, Position
value
   obs_Ships: Harbour → Ship-**set**
   obs_HoldAreas: Harbour → HoldArea-**set**
   obs_Buoys: Harbour → Buoy-**set**
   obs_Quays: Harbour → Quay-**set**
   obs_CSAs: Harbour → CSA-**set**
   obs_Position: Ship → Position-**set**
   obs_Position: Quay → Position-**set**
   obs_Position: Buoy → Position-**set**
   obs_Position: HoldArea → Position-**set**

From a harbour one can observe all the

- ships in the harbour,
- holding areas of the harbour,
- buoys of the harbour,
- quay positions of the harbour and all the
- container storage areas of the harbour.

Positions are associated with

- ships,
- holding areas,
- buoys and
- quays.

One may rightfully argue, as we shall later do, that ships be modelled as behaviours, not as entities. In fact, all the entities listed above could be so considered. What are we to make of that? That is, why model, as here, these phenomena as entities? The answer is, in this case, simple: if and when we model them, instead, as behaviours, then the entity types and values otherwise alluded to above will become configuration (context and state) attributes. ■

Our unfolding story of entities, functions, events and behaviours, will be treated in far more detail in Chap. 5.

   Entities are values and values are of some type. That is, a type stands for a usually infinite set of values. Some entities may be considered atomic.

They have no proper subentities. Other entities may be considered composite. One may speak of proper subentities. Entities have attributes, that is, are of types. Atomic entities may have composite types, like Cartesians of several attributes. An example is a person. A person has a name, a birth-date and a gender. These are different types, i.e., different attributes and are of constant value — usually. A person also has a current (i.e., variable) weight and height. Composite entities naturally have composite types. Some of these types describe proper subentities. Others describe how the subentities are composed into the overall, the composite entity. One can observe from an entity the values of its attributes. Thus one can observe from a composite entity the set or sequence or Cartesian components, etcetera, of its subentities.

_____ Informal Presentation: Entities: Types, Values and Observers _____

We sketch and explain the following formal text:

type
    A, B, C, ..., P, Q, R, ...
value
    a:A,

    obs_B: A → B
    obs_C: B → C
    ...
    obs_Ps: B → P-**set**
    obs_Ql: C → Q*

Type names A, B, C, ..., P, Q, R, ..., are here thought of as abstract types, i.e., sorts. The value "declaration" **values** a:A non-deterministically selects an arbitrary value from type A and names this value (a). Such value declarations correspond to the informal uttering, or writing, *let a be an entity of type* A, ... The *observer* function obs_B applies to values of type A and yields a value of type B. The *observer* function obs_C applies to values of type B and yields a value of type C. The *observer* function obs_Ps applies to values of type B and yields a set of values of type P. The *observer* function obs_Ql applies to values of type B and yields a list of values of type Q. These postulated observer functions correspond to the informal uttering, or writing, of, for example, *from an entity of type* B *one can observe a set of (possibly zero) one or more entities of type* P, ... Postulating a few or several observer functions over values of some sort does not prevent such values containing, i.e., being composed from other subentities, or having other attributes.

In Vol. 1, Sect. 5.2 we treat the concepts of phenomenological and conceptual entities, their atomicity and composition, their types and attributes and their values in detail. Vol. 1 is concerned with formal techniques for basic abstractions and models of phenomena and concepts. Vol. 2 is concerned with formal

techniques for the specification of systems of entities, functions, events and behaviours, and of languages over these. The coverage of entities, types and values of the present chapter will be greatly expanded upon in this volume in Chaps. 5–7, 10, 11, and so on.

Functions

Phenomena or concepts could be *functions* that apply to entities and (i) either test for some property, (ii) observe some subentity, i.e., yield a data value that is "computed" from such entities, or (iii) actually change the entity value — in which case we call the function an operation, or an action.

Example 1.5 *Functions:* For a domain of harbours some typical functions are:

(i) An arriving ship asks the harbour whether it can be allocated either a holding area, a buoy or a quay position.

   value: inquire: Ship×Harbour → **Bool**

(ii).1 An arriving ship which can be allocated a holding area, a buoy, or a quay position requests the position.

   value: request: Ship×Harbour → Position

(ii).2 For a ship destined for a quay position one needs to know how many containers to unload to and how many to load from the harbour:

   value: unload_load_quantities: Ship×Harbour → **Nat**×**Nat**

(iii) A ship [un]loading some cargo.

   value: [un]load: Ship×Quay → Ship×Quay

The functions just listed were, as we shall call it, roughly sketched, but given a signature. Now we give a more satisfactory narrative. Function (i) takes two arguments: a ship and a harbour. It yields whether or not the ship can be received by the harbour. Function (ii).1 takes two arguments: a ship and a harbour. It yields either a holding area, a buoy or a quay position identification. Function (ii).2 takes two arguments: a ship and a harbour. It yields a pair of natural numbers (u, ℓ) indicating that u containers are to be unloaded and ℓ containers are to be loaded. Function (iii) takes two arguments: a ship and a harbour. It yields the same type doublet, but now the ship is [less] plus some cargo, and the quay now is [plus] less that cargo. ∎

_____ Informal Presentation: Function Signatures _____

We define three *sorts*, i.e., *abstract types*, and give the *signature* of four functions:

type
(0) A, B, C
value
(1) inv_A: A → **Bool**
(2) obs_B: A → B
(3) gen_C: B → C
(4) chg_B: A × B → B

(0) A, B and C are sorts, i.e., further unspecified abstract types. (1) inv_A is a predicate: it is supposed to yield **true** for well-formed values of A, **false** otherwise. (2) obs_B is intended as an observer function: from values *a* of sort A it observes, i.e., extracts, values of type B that are somehow "contained" in *a*. (3) gen_C is intended as a generator function: from values of sort B it computes values of type C. (4) chg_B is intended as an operation (i.e., a generator function): from Cartesian values over A and B it generates values of type B intended to replace the argument of type B. One might thus write any of the below:

[5] **variable** b:B := ...; ... b:=chg_B(a,b) ...
[6] **let** b′=chg_B(a,b) **in** ...; b:=b′; ... **end**
[7] **let** b′=chg_B(a,b) **in** ... gen_C(b′) ... **end**

Example 1.6 *The A, B, Cs of Ships and Harbours:* The reader is asked to complete this example, that is, to relate the types and functions of Example 1.5 to the sorts and functions of the box above! ∎

Events

Events happen, i.e., occur. And when events occur they do so instantaneously. Events may convey information, i.e., have significance other than just occurring. We can speak of external events and of internal events. External events occur in an outside environment, "around" the part of the domain being considered — i.e., interfacing with it — and are being communicated to that part. Or external events occur within the domain being considered, and are being communicated to "somewhere" outside the part of the domain being considered. Internal events occur in one part of the domain being considered and are destined for, i.e., communicated to, another part of the domain being considered — in which case we consider those parts as belonging to different behaviours.

Example 1.7 *Events:* For a domain of harbours some typical events are: a ship arrives at a harbour; a ship declares itself ready to unload or to load; a ship and a quay engage in the events of unloading and loading; a ship declares itself ready to depart a holding area, or a buoy or a quay position. ∎

Informal Presentation: Events

In RSL we may model events in terms of RSL/CSP inputs/outputs:

type
    ShipId, ShChar, HAPos, BuoyPos, QuayPos
    MSG == mkArrive(shid:ShipId,shchar:ShChar)
            | mkHoldArea(p:Pos)
            | mkBuoy(b:BuoyPos)
            | mkQuay(q:QuayPos) | ...
    ArrDep == ready | depart
    Cargo

channel
    sh,hs:MSG
    sqr:ArrDep
    sq,qs:Cargo
value
    ship(...) ≡
        ... sh!mkArrive(si,sc) ... **let** pos = hs? ... **end** ...
        ... sqr!ready ... sq!c ... **let** c′ = qs? ... **end** ...
    harbour(...) ≡
        ... **let** mkArrive(s,c) = sh? ... hs!mkQuay(q) ... **end**
    quay(...) ≡
        ... **if** sqr?=ready **then let** c = sq? ... qs!c′ ... **end else** ... **end**

Here we have just sketched some events: the arrival of a ship, the harbour message of (as here, quay) position, the ship signalling readiness, the ship unloading some cargo (c), the ship taking aboard some cargo ($c′$), and the corresponding events in the harbour and quay behaviours.

Behaviours

Some *phenomena* (or *concepts*) are thought of as *behaviours*. They proceed, typically in time, by performing functions (actions), generating or responding

to *events* and otherwise *interacting* (i.e., *synchronising* and *communicating*) with other behaviours.

Formal Presentation: Behaviour

We sketch and explain the following specification text:

type M
channel t_p,t_q : **Bool**, k : M

value
   P() ≡
   p:**while** t_p? **do**
     action_p1;
     k ! v
     action_p3
   end

Q() ≡
variable w:M;
q:**while** t_q? **do**
    action_q1;
    v := k ? **in**
    action_q3
end

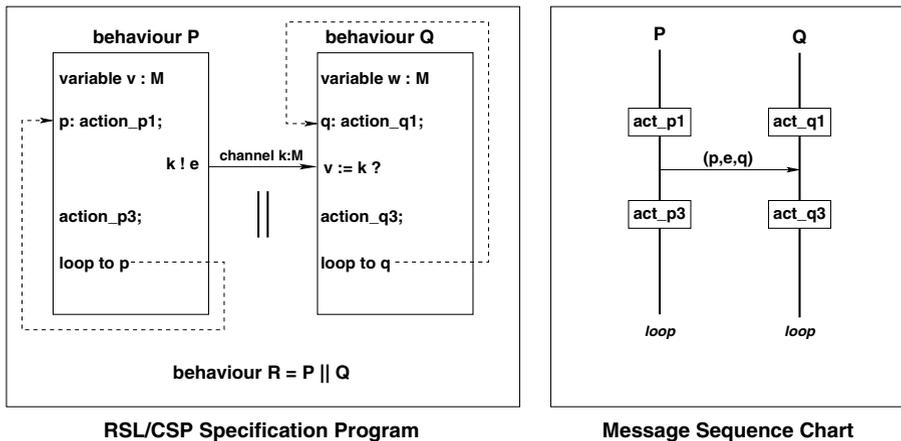R() ≡ P() || Q()

Fig. 1.1. Informal process diagrams

Formal Presentation: Behaviour

We explain the specification text to the left in Fig. 1.1: **type** M is the type of the messages to be sent from **behaviour** P to **behaviour** Q. **Channel**s t_p, t_q are oracles. They determine cyclic behaviour of processes P and Q. **Channel** k is the medium by means of which messages are to be conveyed between behaviours. **Behaviour**s P and Q — in this simple example — take no arguments, hence (). And they are both cyclic: after having honoured their only three **action**s they **loop** back to their first action. The

only really interesting pair is the pair of **input/output** actions. Output
k ! e prescribes that the value of expression e be communicated on chan-
nel k before the computation proceeds to the next action. Input v := k ?
prescribes that a computation inspects channel k to see whether there is
an available message. Once a message has been received it is assigned to
the variable v. Two possibilities now exist: either the process named by P
awaits that the message it has put on channel k is consumed by something
— here the process named by Q. Or the process named by P does not wait
for the message it has put on channel k to be consumed. In the former case
we say that the channel is a 0-capacity buffer; messages are *not persistent*.
In the latter case we say that the channel provides for an infinite-capacity
buffer, then messages are *persistent*. When sketching, i.e., presenting pro-
grams, like the above, the programmer cum specifier must state whether
the channel provides for nonpersistence or persistence. In RSL/CSP channels
are nonpersistent. If channels are nonpersistent, i.e., are 0-capacity buffers,
then we say that the fact that the computation based on behaviour P does
not proceed to its next action before the computation based on behaviour
Q has consumed the message (sent by Q) constitutes a *synchronisation*. In
either case, the message being transferred constitutes a *communication*. We
shall usually use the term behaviour in favour of process. However, when
a behaviour (or rather, a set of behaviours) is implemented by (i.e., exists
inside) the computer, we shall also call it a *process*.

To the right in Fig. 1.1 we have shown an MSC (message sequence chart).
The *loop* annotation is strictly speaking outside the proper MSC syntax.

We refer to Vol. 1, Chap. 21 for a thorough coverage of CSP [168,301,311]
and RSL/CSP. And we refer to Vol. 2, Chap. 13 for a thorough coverage of
MSC [182–184].

Example 1.8 *Railway Entities, Functions, Events and Behaviours:* Our ex-
ample derives from railways. Example railway entities are: (i) the railway net
(N), (ii) its lines (L), (iii) its stations (S), (iv) the units (U) of the net into
which it can be decomposed (linear, switches, crossovers, etc.), etc.

—————— Formal Presentation: Railway Entities ——————

type
   N, L, S, U
value
   is_Linear, is_Switch, is_Crossover: U $\to$ **Bool**

An example railway function is: (v) the issuance of a ticket in return for the
monies it costs. The function issue takes monies (Mo), from-station (Sn), to-
station (Sn), date (Da), train number (Tn) and the state of all train reservations

(TnRes) as arguments and delivers a ticket (Ticket) and an updated state of all train reservations as results.

─────────── Formal Presentation: Railway Functions ───────────

type
    Mo, Sn, Da, Tn, TnRes, Ticket
value
    issue: Mo × Sn × Sn × Da × Tn → TnRes → TnRes × Ticket

(vi) An example railway behaviour is: passengers getting on a train, at a station platform; followed by the departure of the train from the station platform; the ride of the train down the line towards the next station, including the acceleration and deceleration of the train along the line; the arrival of the train at the next station, and subsequently its stopping at a platform; and the alighting of passengers at that platform.

─────────── Formal Presentation: Railway Behaviours ───────────

type
    Sn, Train
value
    train_ride: $Sn^* \to N \to Train \to N \times Train$
    train_ride(snl)(net)(trn) ≡
        if len snl ≤ 1
            then
                (net,trn)
            else
                let (net′,trn′) = get_on_train(**hd** snl)(net)(trn);
                let (net″,trn″) = train_dept(**hd** snl,**hd tl** snl)(net′)(trn′);
                let (net‴,trn‴) = ride(**hd** snl,**hd tl** snl)(net″)(trn″);
                let (net⁗,trn⁗) = arriv_and_stop(**hd tl** snl)(net‴)(trn‴);
                let (net⁗′,trn⁗′) = get_off_train(**hd tl** snl)(net⁗)(trn⁗);
                train_ride(**tl** snl)(net⁗′)(train⁗′)
        end end end end end end

We explain the formula above: Sn^* denotes lists of station names. We assume that the net also registers passengers at stations. train_ride is based on a list of two or more station names, a railway net (with lines and stations) and a train. If the list of station names is of length less than two, the train ride is finished. getting_on_the_train is assumed to accept passengers from the first station of the station list, whereby both the net and the train states change. departure_of_train models the train ride inside the station named first in the station name list. Again, the departure, from the platform to the

beginning of the line going to the next station, changes both the net and the train states. ride models the line part of the journey between neighbouring stations, including acceleration, etc. arrival_and_stop models the train ride inside the station named second in the station name list. Again, the arrival, from the end of the line into this next station to its platform, changes both the net and the train states. getting_off_the_train is assumed to model the alightening of passengers at the second station of the station list, whereby both the net and the train states change. Finally, the train_ride resumes as from this station onwards.

∎

------- Formal Presentation: Some Comments -------

The above behaviour was expressed purely functionally, with references only to simple mathematical functions. That is, these functions are all to be thought of as executing instantaneously. So what is their temporal behaviour, one may very well ask? It is the set of sequences of actions and events denoted by the function definitions. In Vol. 1, Chap. 21, Sect. 21.2.3 we explain what we mean by a trace semantics of behavioural specifications. Temporality is exhibited by orderings of these actions and events. One may, however, read the above formula as if each function took some not-further-specified time to execute, i.e., to be applied. Thus you may trick yourself into believing that the formulas prescribe a timed behaviour.

Example 1.9 *Railway Functions:* We continue Example 1.8.

------- Formal Presentation: Railway Functions and Behaviours -------

Next we give a set of definitions of functions. These evolve around channels and function definitions with synchronisation and communication between functions. We may then claim that this formalisation more properly describes a behaviour. We have tried to make the two formalisations, the above and the below, as similar as possible.

type
    P, SIdx, Tn, Σ, Train
    mTn == mkTn(t_n:Tn)
    mPs == mkPs(p_s:P-**set**)
channel
    { c[s]:(mTn|mPs) | s:SIdx }
value
    obs_Tn: Train → Tn

    passengers: Tn → Σ → Σ × P-**set**
    passengers: Train → SIdx → P-**set**

passengers: Train → P-**set**

station(s)(σ) ≡
    let tn_or_ps = c[s] ? **in**
    case tn_or_ps **of**
        mkTn(tn) →
            let (σ',ps′) = passengers(tn)(σ) **in**
            c[s] ! mkPs(ps′);
            station(s)(σ') **end**,
        mkPs(ps) →
            station(s)(merge(ps)(σ))
    end end

merge: P-**set** → Σ → Σ

train(sl)(τ) ≡
    if len snl ≤ 1
        then
            skip /∗ **assert:** passengers(τ) = {} ∗/
        else
            let s = **hd** sl **in**
            c[s] ! mkTn(obs_Tn(τ));
            let mkPs(ps) = c[s] ? **in**
            let τ' = seat(τ)(ps) **in**
            let τ'' = leave(τ')(s) **in**
            let τ''' = ride(τ'')(s,**hd tl** sl) **in**
            let τ'''' = arrive(τ''')(**hd tl** sl) **in**
            let (τ''''',ps′) = passengers(τ'''')(s);
            c[s] ! mkPs(ps′);
            train(τ''''')(**tl** sl)
    end end end end end end end end
    assert: tn = obs_Tn(τ) = ... = obs_Tn(τ''''')

seat: Simple function
leave: Behaviour − communicates with station rail net
ride: Behaviour − communicates with line rail net
arrive: Behaviour − communicates with station rail net

We shall treat the concepts of phenomena and concepts in Chapter 5. Suffice it for now to justify the above remarks as follows. Entities typically are manifest;

that is, they exist in time and space. Functions can be conceived through their effects, but cannot, in and by themselves, be observed. *Nobody*[3] *has ever seen the number which we may represent by any of the numerals* 7, vii, seven, IIIIIII, III, *etc.* Even more so for behaviours: we may observe a progression of changing entities, effects of function applications and events; but we cannot "see" the behaviour, only conceive of it! The same is true for events.

Further Expectations from Domain Descriptions

What else must we expect from a domain description? Although we shall review domain engineering in Section 1.3.3, and treat domain engineering in detail in Chaps. 8–16, we shall just mention a few things.

(i) We expect a domain description to be readable and understandable by all stakeholders of the domain, i.e., by all those people who "populate" the domain.

(ii) We expect a domain description to be the basis for learning about the domain, that is, for education about and training in the domain — say, for such people as are being hired into a job in the domain, or for such people that need services offered by the domain.

(iii) We expect a domain description to be the basis for constructing a major part of the requirements, namely that part which we shall call the *domain requirements*.

(iv) And we expect a domain description to be a basis for what — in other contexts than software engineering — is known as *business process reengineering*. We shall cover issues of business process reengineering in Chap. 11's Sect. 11.2.1.

Domain Descriptions as Bases for Domain Theories

Physicists have spent the last 400 years studying nature. Traditional engineering disciplines, such as civil engineering, mechanical engineering, chemical engineering, electrical engineering, and electronics engineering, all build on various theories of physics and chemistry. The engineering artifacts, that such engineers build, embody, so-to-speak, fragments of these theories.

For the class of application domains that was characterised as being end-user, public administration and institution oriented, as well as business and industry oriented, for that class of human-made universes of discourse we cannot refer to any such similar theories!

Isn't it about time that we develop theories, such as physicists have done, for respective application domains? This author thinks so.

With the principles and techniques of domain engineering the reader will be well prepared to help contribute research and development on such a theory.

---

[3] Although the analogy should, more properly, be to a function, it is here to another mathematical thing, namely a number.

But to do it properly, the reader needs to learn additional principles, formal techniques and related tools.

More on Domain Engineering

We have briefly previewed some domain concepts, there are many more. For domain engineers to know how to proceed, what to do, and how to do it, and do it professionally, with assurance, it is important that they know what domain engineering entails. In particular they must know what should be, and not be, in a domain description document, that is, its parts and structure. We shall cover these and other domain engineering concepts a little more in Section 1.3.3, and in detail in Chaps. 8–16.

1.2.3 Requirements Engineering

In this section we shall give a brief characterisation of what we mean by requirements engineering.

The Machine

We introduce the term machine.

Characterisation. By *machine* we shall understand a combination of hardware and software that is the target for, or result of, computing systems development. ∎

Discussion. Although the title of these volumes is *Software Engineering* we cannot avoid also dealing with the engineering of the hardware aspects of the computing system for which a domain description is first established, for which requirements are then developed, and for which subsequent software design is finally sought — or completed. That is, although the main development actions may have to do with software, there will necessarily be a hardware design component in that development. The software "resides" on, or "in" some hardware (computer); the software thus relies on that computer and its peripherals having certain minimal properties, etc. So the requirements shall stipulate properties, not only of the software, but possibly also of the hardware. ∎

The objective of writing down requirements is to prescribe desired properties of a machine: the software and the hardware on which the software resides.

The Machine Environment

We introduce the concept of the environment of a machine.

Characterisation. By *machine environment* we shall understand the rest of the world. More specifically, we mean those parts of the world which interface to the machine: its users, whether humans or technology.   ∎

Discussion. The concept of machine environment is a fuzzy one. Ideally the machine environment includes all the stakeholders of the machine, that is, of the new services (functions and facilities) offered by the machine, as well as all the nonhuman interfaces to the machine: monitored and controlled phenomena of the world "surrounding" the machine. But, to predict, in advance of establishing the requirements to the software to be designed, and in advance of the actual installation of that software, which are to be "all" these stakeholders and "all" those affected phenomena, is an art; it is not easy.   ∎

So the objective of writing down requirements is also to delineate, to decide upon and distinguish between what is to "belong" to the machine, and what is to "belong" to the environment.

General

Characterisation. By *requirements* we mean a document which prescribes desired properties of a machine: *what* the machine shall (must, not should) offer of functions and behaviours, and what entities it shall maintain.   ∎

Characterisation. By a *requirements prescription* we mean the process — and the document which results from the process — of requirements capture, analysis and synthesis.   ∎

Characterisation. By *requirements engineering* we understand the engineering, that is, we understand the development of requirements prescriptions: from requirements prescription via the analysis of the requirements document itself, its validation with stakeholders and its possible theory development. ∎

Different Kinds of Requirements

We see four different kinds of requirements: (i) *business process reengineering*, (ii) *domain requirements*, (iii) *interface requirements*, and (iv) *machine requirements*.
    Conventionally the following terms are in circulation: *systems requirements*, which approximately covers our overall requirements, *user requirements*, which approximately covers our domain and interface requirements, *functional requirements*, which approximately covers our domain and interface requirements and *non-functional requirements:* approximately covers our machine requirements.

More on Requirements Engineering

We have briefly previewed some requirements concepts. There are many more. For the requirements engineer to know how to proceed, what to do, and how to do it, and do it professionally, with assurance, it is important that that engineer knows what requirements engineering entails, in particular: what should be, and not be, in a requirements prescription document: its parts and structure.

We shall cover these and other requirements engineering concepts a little more in Section 1.3.4, and in detail in Chaps. 17–24.

1.2.4 Software

Characterisation. By *software* we understand (i) not only *code* that may be the basis for executions by a computer, (ii) but also its full *development documentation*: (ii.1) the stages and steps of *application domain description*, (ii.2) the stages and steps of *requirements prescription*, (ii.3) and the stages and steps of *software design* prior to code, with all of the above including all *validation* and *verification* (including *test*) *documents*. In addition, as part of our wider concept of software, we also include (iii) a comprehensive collection of *supporting documents:* (iii.1) *training manuals*, (iii.2) *installation manuals*, (iii.3) *user manuals*, (iii.4) *maintenance manuals*, and (iii.5, iii.6) *development* and *maintenance logbooks*. So, software, as documentation, comprises many parts.　■

1.2.5 Software Design

From an understanding of what software syntactically, i.e., as documents, "is", we can go on to characterise pragmatic and semantic aspects related to software.

Characterisation. By *software design* we understand the process, as well as all the documents resulting from the process, of turning requirements into executable code (and appropriate hardware).　■

We make a distinction between two kinds of abstract software specifications, and hence their designs: the software architecture, and the component structure. After a brief presentation of these we shall comment on their nature.

Software Architecture and Software Architecture Design

Characterisation. By a *software architecture* we mean a first specification of software, after requirements, that indicates *how* the software is to handle the given requirements in terms of components and their interconnection — though without detailing, i.e., designing these components.　■

Characterisation. By a *software architecture design* we mean the development process of going from existing requirements and possibly some already designed components to the software architecture — producing all appropriate architecture documentation. ∎

The term *component design* is used here in a perhaps confusing sense: In architecture design we may certainly identify components, delineating their "boundaries"[4], but leaving their "internals" undefined.[5]

Component Structure and Component Design

Characterisation. By a *component structure* we mean a second kind of specification of software — after requirements and software architecture — one which indicates *how* the software is to implement individual components and modules. ∎

Characterisation. By *component design* (I) we mean the development process of going from existing requirements and a software architecture design to the detailed component modularisation — producing all appropriate component and module documentation. ∎

Software Architecture Versus Component + Module Structure

We are not saying that one must first design the software architecture, and thereafter the component plus module structure. We are presently leaving their order of development — and one of the two, or even a "mix" of them — unexplained!

Modules, Components and Systems

The principle of grouping programming text into modules and collections of modules into components is both old (for modules since the late 1960s, e.g., Simula 67 [30]), and new (for components since the early 1990s).

Characterisation. By a *module specification* we shall understand a syntactic construct, i.e., a structure of program text, which, as a unit of program text, defines what we shall otherwise also call an *abstract data type:* namely a collection of data values and a collection of functions (i.e., operations) over these. ∎

---

[4] When reading this book in the formal version: Defining their types only as sorts and defining their functions only by giving signatures.

[5] When reading this book in the formal version: That is, their concrete type counterparts, respectively the function definition bodies, undefined. Thus architecture design may "design" part of the component structure.

Discussion. So, by an *abstract data type*, i.e., a *module* we mean a set of data values and a set of procedures (routines) that apply to such data values and yield such data values. Typically *module specifications* are of the following schematic form:

module m:
   types
      t1 = te1, t2 = te2, ..., tt = tet;
   variables
      v1 **type** ta := ea, v2 **type** tb := eb, ..., vv **type** tc := ec
   functions
      f1: ti $\rightarrow$ tj, f1(ai) $\equiv \mathcal{C}_1$(ai)
      f2: tk $\rightarrow$ tℓ, f2(ak) $\equiv \mathcal{C}_2$(ak)
      ...
      fn: tp $\rightarrow$ tq, fn(ap) $\equiv \mathcal{C}_1$(ap)
   hide: fi, fj, ..., fk
end module

(1) *Syntactically* the idea of the above is roughly as follows: m is the name of the module. The module defines t types: $t1, t2, \ldots, tt$, local to that module. A type definition has a left-hand side name ti, and a right-hand side type expression tei. These right-hand side type expressions could be such things as `integer`, `real`, `Boolean`, `character`, `record` structures over types, `vector` structures over a type, and so on. The module also declares v variables: $v1, v2, \ldots, vv$, local to that module. A variable declaration consists of three parts: the variable name, the type of the data values that the variable is allowed to contain and an initialising expression e (something). The module then defines n functions (procedures, routines, operations, methods, or whichever name you wish to call them). Each function definition has two parts: a *function signature,* and a *function definition body.* The function signature defines the name of the function, f, and the type of argument and result values — those types, t, t', that are mentioned on either side of the function space symbol $\rightarrow$. The function definition body consists of three parts: the function name followed by a *formal parameter list*, a *function definition symbol*, say, as here, $\equiv$, and the *function body* — here schematised in the form of the abstract clause $\mathcal{C}$(a). This clause can stand for an expression, or a list of statements, or whatever your favourite programming language allows. The module finally lists those function names which are local, i.e., which cannot be referred to by program texts outside the defining module.

(2) The above described, to some — albeit incomplete — extent, the syntax of a typical kind of module. We shall explain the *semantics*, i.e., the meaning, of a module, by just saying: You are assumed to know the meaning of type definitions, of variable declarations, and of function (procedure, method, etc.) definitions. So what's new? The new "thing" is the "encapsulation", the structuring, the "putting together" of these program text structures in a

module structure — one beginning with the keyword **module** and one ending with the keyword **end module.** The meaning of this encapsulation is — again roughly speaking — as it changes "slightly" from actual programming language to actual programming language — that all types, all variables, all function definition bodies, and some function names and signatures are hidden, that is: Their definition cannot be known by program texts outside the defining module. Thus one can substitute one module text by another as long as the function signatures that were visible from outside remain the same in the new, the replacement module. This typically means that the signature types of visible functions are not locally defined types.

(3) The *pragmatics* of a module — perhaps its most important distinguishing feature — can be summarised as follows: The programmer has decided, for whatever reasons, to "lump together" some data structures, in the form of typed variable declarations, and some functions over these, to form an abstract data type, while preserving the right to replace the implementation details of this abstract data type with any other that is believed to yield the same abstract data type. Two things are involved here. First, we have the *hiding* of *implementation details*, that is, (i) the local types, the local variables, the auxiliary functions (those which are hidden) and the bodies of all function definitions. (ii) Second, we have the decomposition of larger program texts into a collection, a usually unordered set, of module definitions. We shall have more to say about this in later sections and chapters. ∎

Characterisation. By a *component specification* we shall usually understand a set of type definitions, a set of component local variable declarations, together defining a component local state, and a set of modules. ∎

The above is just a rough, generic characterisation of components.

Discussion. The idea is that a *component specification* to some surrounding text offers functions of some of its modules. The surrounding text may consist of modules, what we call *initial modules* of a *software system*. ∎

We may suggest a syntax for components:

> **component**
>   **types:** $\mathcal{T}_{i_1}, \mathcal{T}_{i_2}, \ldots, \mathcal{T}_{i_t}$
>   **variables:** $\mathcal{V}_{j_1}, \mathcal{V}_{j_2}, \ldots, \mathcal{V}_{j_v}$
>   **modules:** $\mathcal{M}_{k_1}, \mathcal{M}_{k_1}, \ldots, \mathcal{M}_{k_m}$
>   **hide:** $\mathcal{H}_{\ell_1}, \mathcal{H}_{\ell_1}, \ldots, \mathcal{H}_{\ell_h}$
> **end component**

$\mathcal{T}_i$ suggests some form of type definitions, $\mathcal{V}_j$ suggests some form of variable declarations, $\mathcal{M}_k$ suggests some form of module specifications, and $\mathcal{H}_\ell$ suggests some form of export or hiding of module visible functions (etc.).

Systems, Design and Refinement

Characterisation. By a *software system specification* we shall understand a set of what we shall call *initial* modules together with a set of components — and such that functions of the set of initial modules together invoke functions of modules in the set of components. Systems are what we are developing. ■

Discussion. The idea is that a *system* is a completely self-contained "item" of software, and that it is composed from components and the *core*, that is, the initial modules. The idea is also that a most abstract level system may be the same as a software architecture, or a component plus initial module structure. ■

Characterisation. By *software system design* we understand (i) the determination, (i.1) from domain requirements and from some interface requirements, of the software architecture, or (i.2) from machine requirements and from other interface requirements, of the component structuring plus initial modules. Since software architecture design also entails determination of component structuring plus initial modules, we get, more generally, that software system design, in its first stage, i.e., where only the domain description and the requirements prescription exists, entails (ii.1) the determination of the main (system) types of values, (ii.2) the determination of the basic structuring of and facilities (i.e., functions) offered by components, and (ii.3) the determination of such initial modules as are necessary to get the system executing once it is committed. ■

Usually a first stage of systems design is expressed abstractly, i.e., in a form not suited as a prescription for execution. Hence we need stages and steps of what is called *refinement*:

Characterisation. By *software system refinement* we understand (i) the stagewise and stepwise transformation of (i.1) an abstract specification (i.2) into increasingly more concretely specified modules and components. ■

Characterisation. By *abstract specification* we mean one that indicates how requirements are to be implemented, but does it by using specification cum programming constructs that are not necessarily efficiently executable. ■

Characterisation. By *concrete specification* we mean one that uses specification cum programming constructs that prescribe efficient executions. ■

Components and Modules, Design and Refinement

Characterisation. By *component design* (II) we shall additionally understand the determination of which facilities, that is, which functions (defined

locally "within" the component), and which types (defined globally, i.e., "outside"), the component shall offer. We shall also, by component design roughly speaking mean, the decomposition of the component into modules, and hence, the functions offered by these modules.   ∎

One cannot expect a first attempt at component design to succeed in finalising all aspects of an efficient implementation. As will be argued in the next section, separation of concerns makes it easier to tackle many diverse issues. Hence our development needs to proceed in stages and steps of refinement.

Characterisation. By *component refinement* we shall usually understand: (i) a concretisation of the usually initially abstractly defined component types, (ii) a concretisation of the usually initially abstractly specified initialisations of component variables, and, possibly, (iii) the refinement of the component modules.   ∎

Characterisation. By *module refinement* we understand: (i) a concretisation of the usually initially abstractly defined module types, (ii) a concretisation of the usually initially abstractly specified initialisations of component variables and (iii) a concretisation of the usually initially abstract module function definitions — (iv) with the latter often entailing the introduction of additional auxiliary (i.e., hidden) function definitions.   ∎

Code Design

Finally, we reach the development stage where such program specifications are constructed that can be the basis for efficient execution. We call this kind of program specification 'code'. Since we shall assume the reader to have a necessary background in programming we shall not cover this topic in these volumes.

More on Software Design

We have briefly previewed some software concepts. There are many more. We shall cover these and other software design concepts a little more in Section 1.3.1, and in some detail in Chaps. 25–30. But, these volumes will not present *anywhere near* a fully satisfactory treatment of the software design problem. That is neither the aim nor the objectives of these volumes. First, we have assumed some knowledge, education and training of the reader. Second we have to refer to special topic texts for detailed software design principles, techniques and tools.

1.2.6 Discussion

We have introduced the three main phases of software development:

- Domain engineering in which we describe "what there is"
- Requirements engineering in which we prescribe "what there shall be!"
- Software design in which we specify "how it will be!"

We have indicated, assuming some programming maturity of the readers, some software design structuring — such as revolving around components and modules (with locality and hiding of names), types and variables (abstract and concrete), and program statements and expressions — summarised as clauses ($\mathcal{C}$).

We have not — so far — suggested similar structuring mechanisms for domain descriptions, or for requirements prescriptions. The software design cum programming language structuring constructs of components and modules, of types and variables, etc., aid the developer in knowing "what to do next!" by providing documentation "standards". In the next section we shall preview such textual structuring (decomposition, composition) mechanisms for domain descriptions and requirements prescriptions.

Also, we have intimated, rather loosely, notions of abstract versus concrete software design specifications, and hence we have intimated the entailed notion of refinement. We have not mentioned such stagewise and stepwise mechanisms, for domain descriptions and requirements prescriptions in general, other than for the business process reengineering, and domain, interface and machine requirements stages. Such development stage and step principles are mentioned already in the next section.

1.3 Phases, Stages and Steps of Development

Three Terms

The terms phase, stage, and step, are just that: terms. They are meant to designate basically the same idea: the decomposition of something occurring in time into adjacent, repeated or concurrent intervals. The "something" here is the development of software. The adjacent, concurrent or overlapping intervals are logically and otherwise distinguishable development activities.

A Principle of "Separation of Concerns"

The main reason for decomposing the *software development process* into clearly distinguishable development activities is to tackle separate development issues at separate times, hopefully scheduling these in adjacent, concurrent or overlapping intervals in a fruitful, beneficial way.

In the next several numbered sections we shall briefly review possible decompositions. Each represent a concern; together they represent *separation of concerns*.

Linear, Cyclic and Parallel Development Activities

In the next sections we shall present a view of the software development process as proceeding in strict linear order. Given human nature, such is rarely the case. At the end of this section we shall therefore present two additional views on the software development process: one in which iterations, backwards and forwards, are discussed; and one in which the concurrent tackling of logically separate stages or steps is discussed. By a repeated, or cyclic, interval we mean two intervals, occurring in non-overlapping time periods, in which basically the same item of work is done, i.e., repeated, for example, because a first iteration was not good enough. By concurrent or overlapping intervals we mean two (or more) intervals, in which clearly unrelated work items can be done, independently of one another, hence in parallel.

1.3.1 Phases of Software Development

We have already introduced the main three phases of software development: (i) domain development, (ii) requirements development, and (iii) software design. We have earlier argued for their distinctness, i.e., their focus on truly separate concerns, but we have also emphasised their desirable order, namely as listed.

1.3.2 Stages and Steps of Development

In order to capture a notion of development stage it is important to first capture a notion of the complete documentation — as here — of a phase of development. The documentation for a phase of development is complete if all there is to be documented — at a certain level of abstraction — has been so documented! The idea of "all there is to be documented" is explained in Chap. 2.

It is thus important to also capture a notion of abstract versus concrete documentation — as here — of a phase of development. The phase documentation can be more or less abstract, i.e., more or less concrete. The phase documentation is abstract if primarily properties have been described. The phase documentation is concrete if primarily a model in terms of either a computable program, or a model in terms of such discrete mathematical notions as sets, Cartesians, lists, maps, etc., has been presented. The above distinction allows one to speak about grades of abstractness, versus grades of concreteness.

The distinction between stages and steps is basically a pragmatic distinction. That is, there is no "hard" theoretical basis for making that distinction, but there are good, sensible, practical reasons for doing so.

Characterisation. By a *development stage* we shall understand a set of development activities which either starts from nothing and results in a complete phase documentation, or which starts from a complete phase documentation

of stage kind, and results in a complete phase documentation of another stage
kind.   ■

Characterisation. By a *stage kind* we shall loosely understand a way of
characterising a set of development documents as being comprehensive (i.e.,
relatively complete) and, at the same time, as specifying (describing, prescrib-
ing) a set of properties of what is being specified in such a way that other such
sets of documents can be said to describe the same stage kind, or a different
stage kind.   ■

Characterisation. Thus a *stage kind* imposes an equivalence relation on
a set of sets of related documents: some sets, $s_k, s'_k, \ldots, s''_k$ as belonging to
the same kind (k), other sets, $s_k, s'_{k'}, \ldots, s''_{k''}$ as belonging to different kinds
$(k, k', \ldots, k'')$.   ■

Discussion. The notion of *stage kinds* is deliberately vague. As for philo-
sophical notions, it therefore needs to be discussed and exemplified. Here we
shall discuss that notion. The basic problem is really that, in actual devel-
opment practice, we need operate with a spectrum of "phases", "stages" and
"steps". That is, the simple tripartite decomposition into phases (domain,
requirements and software design) may be OK, whereas the likewise simple
quadruple decomposition into, for example, business process reengineering
requirements, domain requirements, interface requirements and machine re-
quirements stages may, in several development cases, not be entirely satisfac-
tory. The borders between these stages are not that sharp. Human ingenuity
allows us to break molds, and to discover new principles and techniques. So
why do we then suggest the phases, stages and steps that we do indeed name
and describe? We do this so that the reader can be looking vigilant for ad-
ditional stage and step concepts. In the best case the reader will discover
additional, usefully nameable stage concepts. In the worst case the reader
may finally decide that our stage and step conjecture is all wrong, and must
be refuted. Such, sometimes, happens — as is amply illustrated in works by
Imre Lakatos [208] and Sir Karl Popper [277–279].   ■

The examples given next presuppose that you have read the previous ma-
terial carefully. We are basically referring to concepts that were just briefly
mentioned, i.e., to concepts that will only be further mentioned below, to be
finally "disposed of" in later chapters. Some examples, and the discussion text
following, refer to concepts that are introduced only a few pages further on!

Example 1.10 *Stage Kinds:* Examples of domain stage kinds are: (d_1) busi-
ness processes, (d_2) intrinsics, (d_3) support technologies, (d_4) management
and organisation, (d_5) rules and regulations and (d_6) human behaviour.

Examples of requirements stage kinds are: (r_1) business process reengineer-
ing, (r_2) domain requirements, (r_3) interface requirements and (r_4) machine
requirements.

Examples of software design stage kinds are: (s_1) software architecture, (s_2) component design (in which the entire component structuring of a software architecture is decided), (s_3) module design (in which all modules of all components are designed) and (s_4) code.   ■

Discussion. One may properly argue whether the following are not also *stage kinds* rather than steps: domain requirements (r_{2_1}) projection, (r_{2_2}) determination, (r_{2_3}) instantiation, (r_{2_4}) extension and (r_{2_5}) fitting. It really is just a matter of convenience, hence pragmatics.   ■

To properly describe what we shall wish to call a step of development it seems necessary to further elaborate on two concepts. First the concept of a *module of description*. We already covered a version of this notion in an earlier section, where it was "tied" to the concept of program specification (text). We now enlarge upon the module concept and speak of similar, contained domain description and requirements prescription parts. Second we refer to the concept of *refinement*. We also covered a version of this notion in an earlier section, where it was likewise "tied" to the concept of relation between pairs of program specifications (texts). We now enlarge upon the refinement concept and speak of similar refinements of domain description modules as well as requirements prescription modules.

Characterisation. By a *development step* we mean a refinement of a description module, from a more abstract to a more concrete description.   ■

It may now be necessary to improve upon the characterisation of the concept of stage, so as to make the distinction between stages and steps more practical.

Characterisation. By a *development stage* we mean a set of development activities such that some (one or more) activities have created new, externally conceivable (i.e., observable) properties of what is being described, whereas some (zero, one or more) other activities have refined previous properties.   ■

1.3.3 Domain Development

Stages of Domain Development

Within domain development we can distinguish the following major stages — on which work can beneficially be pursued basically in the order now listed: identification and classification of *domain stakeholders*, and identification and modelling, relative to identified domain stakeholder classes, of a number of *domain facets*. (i) These include: modelling *business process* facets of a domain, (ii) modelling *intrinsics* facets of a domain, (iii) modelling possible *support technology* facets of a domain, (iv) modelling possible *management and organisation* facets of a domain, (v) modelling possible *rules and regulations*

facets of a domain, (vi) modelling possible *script* facets of a domain, and (vii) modelling possible *human behaviour* facets of a domain. We shall briefly characterise these stages shortly, and we cover them in detail in Chap. 11.

Characterisation. By a *business process* domain we shall understand one or more behavioural descriptions in which strategic, tactical and operational sequences of transactions of a business,[6] an enterprise,[7] a public administration,[8] an infrastructure component,[9] are given — each from possibly a number of stakeholder perspectives[10]. ∎

The business process facet overlaps with the next facets. So be it!

Example 1.11 *Railway Business Processes:* A simple business process is that of a passenger inquiring, with a travel agent, about train travel possibilities; being offered some alternatives; settling for one; reserving appropriate tickets; paying and collecting these; starting the travel (i.e., train journey); being ticketed and finishing the journey. ∎

Characterisation. By domain *intrinsics* we shall understand those phenomena and concepts of a domain which are basic to any of the other facets (listed below), with such a domain intrinsics initially covering at least one specific, hence named, stakeholder view. ∎

Example 1.12 *Rail Intrinsics:* Examples of rail intrinsics are: the rail net, the lines, and the stations — as seen from the passenger perspective — and the above plus the rail units (whether linear [including curved], points [switches], crossover, etc.), connectors (that allow units to be put together), etc. — as seen from the rail net signalling staff; and so on. ∎

Characterisation. By domain *support technology* we shall understand ways and means of implementing certain observed phenomena. ∎

Example 1.13 *Railway Support Technologies:* A rail unit switch can be implemented in either of a number of support technologies: as operated purely

---

[6] By a business we mean such things as a retail store, a wholesaler, a hotel, a restaurant, etc.

[7] By an enterprise we mean such things as a manufacturing plant, like a distribution company, like a logistics firm, etc.

[8] By a public administration we mean such things as taxes and excises, social services etc.

[9] By an infrastructure component we mean such things as a nation's healthcare system (whether public and/or private), a rail infrastructure owner, a railway, etc.

[10] The above examples, i.e., footnotes 6–9, overlap, and are only suggestive.

by human power, as operated, from afar by mechanical wires, as operated electromechanically, or as operated electronically and electromechanically (say, in interlocking mode). ∎

Characterisation. By domain *management* we shall understand such people who determine, formulate and thus set standards (rules and regulations, see next) concerning strategic, tactical, and operational decisions. Domain management (i) ensures that these decisions are passed on to the ("lower") levels of management, and to "floor" staff, (ii) makes sure that such orders, as they were, are indeed carried out, (iii) handles undesirable deviations in the carrying out of these orders cum decisions, and (iv) "backstops" complaints from lower management levels and from floor staff. ∎

Example 1.14 *Railway Management:* An aspect of train operator management is that some functions, being of strategic nature, are considered on a yearly basis (whether to offer new train services). Other functions, being of a tactical nature, are considered more regularly, although not daily (whether prices should be lowered or raised due to lower, or higher costs, or due to competition or lack thereof). Yet other functions being of an operational nature, and are considered, and decided upon, "from hour to hour" (rescheduling trains due to delays, etc.). ∎

Characterisation. By domain *organisation* we shall understand the structuring of management and nonmanagement staff levels, and the allocation of strategic, tactical and operational concerns to within management and nonmanagement staff levels. Hence we mean the "lines of command": who does what and who reports to whom, both administratively, and functionally. ∎

Example 1.15 *Railway Organisation:* Example 1.14 considered management functions. The number and specialised nature of these usually warrants corresponding organisational structures: executive management entrusted with strategic issues, mid-level management with tactical issues and "floor" (or operational) management with operational issues. ∎

Characterisation. By a domain *rule* we shall understand some text which prescribes how people or equipment are expected to behave when dispatching their duty, respectively when performing their functions. ∎

Example 1.16 *Railway Rules:* In China: At railway stations, no two (or more) trains are allowed to enter and/or leave, including basically move around, simultaneously. In fact, train arrivals and departures must be scheduled to occur with at least 2-minute intervals.

Elsewhere: A line between neighbouring stations is usually segmented into blocks with the rule that at most one train may occupy any one block, or even, in cases, with at least one "empty" (i.e., no train block) between two trains. ∎

Characterisation. By a domain *regulation* we shall understand some text which prescribes what remedial actions are to be taken when it is decided that a rule has not been followed according to its intention. ∎

Example 1.17 *Railway Regulations:* Regulations may thus prescribe properties that must hold when rescheduling trains, for instance, negotiating with neighbouring stations, etc. Or regulations may prescribe punitive staff actions when a train driver disobeys a train signal. ∎

Characterisation. By domain *human behaviour* we shall understand any of a quality spectrum of humans carrying out assigned work: From careful, diligent and accurate, via sloppy despatch and delinquent work, to outright criminal pursuit. ∎

Example 1.18 *Railway Staff Behaviour:* A railway ticket collector may check and double-check that all passengers have been duly ticketed, or may fail to do so, or may deliberately skip checking a whole carriage, etc. ∎

Steps of Domain Development

Steps of domain development are now to be seen as such activities which do not materially, i.e., in substance, change the properties of what is being described, but which refine, from more abstract to more concrete, their way of description. Other than the above, we shall not cover the issue of domain development steps in the present chapter, but refer to Chaps. 8–16 for more details.

1.3.4 Requirements Development

From Sect. 1.2.3 we repeat some of the below characterisations.

Characterisation. By *requirements* we shall understand a document which prescribes the desired properties of a machine: *what* the machine shall (must, not should) offer of functions and behaviours, and what entities it shall "maintain". ∎

Characterisation. By *requirements prescription* we mean the process — and the document which results from the process — of requirements capture, analysis and synthesis. ∎

Characterisation. By *requirements engineering* we understand the development of requirements prescriptions: from requirements prescription via the analysis of the requirements document itself, its validation with stakeholders and its possible theory development.   ∎

Stages of Requirements Development

We see four different kinds of requirements: (i) *business process reengineering*, (ii) *domain requirements*, (iii) *interface requirements*, and (iv) *machine requirements*. Conventionally the following terms are in circulation:

- *functional requirements* which approximately cover our domain requirements;
- *user requirements* which approximately cover our interface requirements;
- *non-functional requirements* which approximately cover some of our machine requirements; and
- *system requirements* which approximately cover some other of our machine requirements.

Business Process Reengineering Requirements

Characterisation. By *business process reengineering requirements* we understand such requirements which express assumptions about the future, usually changed, business process behaviour of the environment of the machine as brought about by the introduction of computing.   ∎

We suggest five domain-to-business process reengineering operations — which will be covered in Sect. 19.3: (i) introduction of some new and removal of some old *support technologies*, (ii) introduction of some new and removal of some old *management and organisation structures*, (iii) introduction of some new and removal of some old *rules and regulations*, (iv) introduction of some new and removal of some old work practices (relating to *human behaviours*), and related *access rights* (i.e., password authentication, authorisation), and (v) related *scripts*.

Domain Requirements

Characterisation. By *domain requirements* we understand such requirements, to software, which are expressed solely in terms of domain phenomena and concepts.   ∎

We suggest five domain to requirements operations that will be covered in Sect. 19.4: domain projection, domain determination, domain instantiation, domain extension and domain fitting.

Business Process Reengineering and Domain Requirements

So in setting out, initially, acquiring (eliciting, "extracting") requirements, the requirements engineer naturally starts "in" or "with" the domain. That is, asks questions, of or to the stakeholders, that eventually should lead to the formulation of business process reengineering and domain requirements.

Interface Requirements

Characterisation. By *interface requirements* we understand such requirements, to software, which are expressed in terms of domain phenomena shared between the environment and the machine. ∎

We consider five kinds of interface requirements which will be covered in Sect. 19.5: *shared data initialisation requirements, shared data refreshment requirements, man-machine dialogue requirements, man-machine physiological interface requirements,* and *machine-machine dialogue requirements.*

Machine Requirements

Characterisation. By *machine requirements* we understand those requirements of software that are expressed primarily in terms of concepts of the machine. ∎

We shall, in particular, consider the following kinds of machine requirements — to be covered in Sect. 19.6: *performance requirements, dependability requirements, maintenance requirements, platform requirements* and *documentation requirements.*

Steps of Requirements Development

Steps of requirements development are now to be seen as such activities which do not materially, i.e., in substance, change the properties of what is being prescribed, but which refine, from more abstract to more concrete, their way of prescription. Other than the above, we shall not cover the issue of requirements development steps in the present chapter, but refer to Chaps. 17–24 for more details.

1.3.5 Computing Systems Design

Given a comprehensive set of requirements, including, notably, machine requirements, one is then ready to tackle, systematically, the issue of implementing these requirements. Usually these requirements not only, as their main implication, direct us to design software, but also in many instances imply hardware design. In other words: computing systems design derives from requirements.

Stages and Steps of Hardware Design

Performance, dependability and platform requirements typically imply a need for rather direct considerations of hardware — whether computers, computer peripherals, or sensory and actuator technologies. By stages and steps of hardware design we thus mean such which determine the overall composition of hardware: information technology units, buses, etc., and their interfaces, and the specific design of information technology units and buses, and so on.

Sometimes trade-off decisions have to be made as to whether a required function or behaviour is to be implemented in hardware or in software. These are called *codesign* decisions. Other than just mentioning these facts here, we shall not cover the subject till Chap. 25.

Stages of Software Design

We have briefly mentioned the problem before: sometimes a set of requirements and a set of (domain) assumptions on the stability of the environment and the execution platform allow us to first develop a high-level, i.e., abstract, software design from domain (and possibly some interface) requirements. At other times these assumptions are such (i.e., imply instability, such) that we must first, defensively, develop a less high-level, i.e., a less abstract, software design from machine requirements.

In the former case we say that we are first designing a *software architecture:* something that very directly reflects what the user most directly expects. In the latter case we say that we are first designing a *software component and module structure:* something that very directly reflects what some machine requirements imply. The boundary between the two design choices is not sharp.

It is possible to identify other software design stages. Some may involve "conversion" from informal (or formal, abstract specification) language specifications to the identification and (hence) reuse of existing, "ready-made" and/or instantiatable (i.e., parameterisable) "off-the-shelf" (OTS) modules and components. Others involve conversion from informal (or formal, abstract specification) language specifications to formal (say, programming) language specifications, without the use of OTS software. This stage includes the final coding stage. Also here the boundaries are usually fuzzy.

Steps of Software Design

We have, for this book, assumed that the reader already has some knowledge of programming, i.e., of software design, albeit at a perhaps rather concrete, i.e., coding, level. In line with this assumption we shall not treat the important concept of software refinement. Instead we shall assume that the reader has studied, or will study, such textbooks as: Dijkstra's *Discipline of Programming* [86], Gries' *Science of Programming* [130], Reynolds' *Craft of Programming* [296], Hehner's *Logic and Practical Theory of Programming* [158,159], Jones'

Systematic Software Development [197,198], Morgan's *Refinement Calculus* [249] or Back and von Wright's (earlier) *Refinement Calculus* [19]. In Chap. 29 we shall, however, briefly illustrate notions of software design refinement.

1.3.6 Discussion: Phases, Stages and Steps

The *phase, stage* and *step* concepts, i.e., the concept of *separation of concerns*, are — pragmatically as well as semantically — important. Hence we shall further clarify these concepts.

Iterations of Phases, Stages and Steps

Ideally it would be nice if a software development could proceed linearly, from domain development, via requirements development, to software design. But reality seldom permits linear thinking and development. Instead one often encounters, in software developments which span the three phases, that they iterate: forwards and backwards between temporally neighbouring, even further temporally spaced phases. Figure 1.2 attempts to illustrate this iteration.
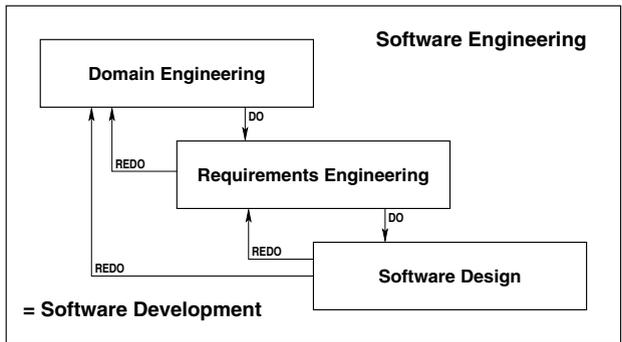

Fig. 1.2. A diagramming of the iterative triptych phase development

The forward, i.e., the temporally linear progression, is in Fig. 1.2 shown by arrows labelled DO. The backward, i.e., the temporally iterative regression, is shown by arrows labelled REDO. Thus iteration is afforded by traversing, in one's development, a sequence of one or more DO and one or more REDO labelled arrows. We shall later explain, more carefully, what it may mean to do iterative and evolutionary development. Similar remarks can be made concerning iterative stagewise and iterative stepwise developments.

Concurrency of Phases, Stages and Steps

Usually phases are developed, one at a time. First the domain phase is developed, then the requirements phase, and finally the software design phase. For

stages of a phase and steps of different stages one may sometimes be able to
carry out their development concurrently, that is, by different teams of devel-
opers at the same, or at least in partially overlapping time intervals. Stage of
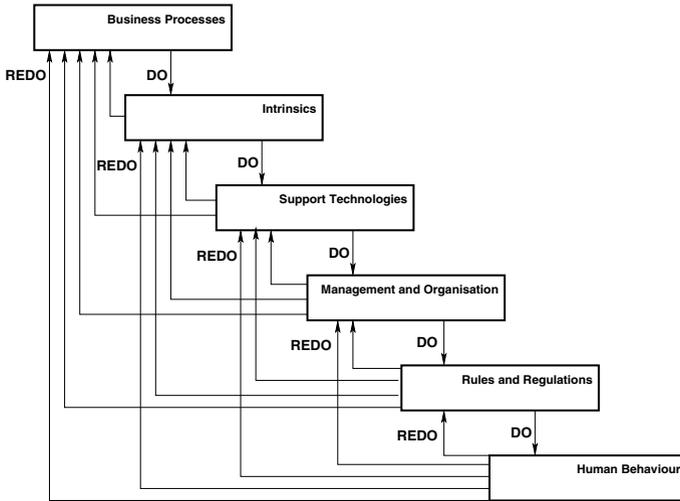domain modelling usually follow the sequential order shown in Fig. 1.3.

Fig. 1.3. A diagramming of iteration of domain stages

Typically the domain requirements, the interface requirements and the ma-
chine requirements stages can be developed independently, i.e., concurrently.
Independent development is also appropriate for the individual "steps" within
machine requirements: performance, dependability, maintainability, platform,
and documentation requirements steps. Figure 1.4 illustrates the possibly in-
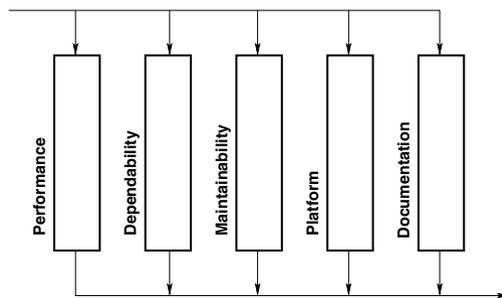dependent development of machine requirements stages.

Fig. 1.4. A diagramming of stage concurrency of machine requirements

The diagrams shown in this section lead us into the subject of process models.

1.4 The Triptych Process Model — A First View

The term process model has had, and continues to have, some currency and is somewhat fashionable among practitioners and researchers of software engineering. We shall therefore very briefly respond to that notion.

1.4.1 The Concept of a Process Model

In software development many teams of many people each may have to collaborate over long periods of time and over geographically widely distributed locations. It is therefore of utmost importance that clear guidelines, principles, techniques and tools are established and are agreed upon by all teams and people involved. The concept of a *software development process model* and its enunciation serves this role.

Characterisation. By a *software development process model* we shall understand a set of guidelines for how to start, conduct and end a software development project, a set of principles and techniques for decomposing these parts (start, conduct and end) into smaller, more manageable parts, and a set of principles, techniques and tools for what to do in, and how to do, these smaller parts. ∎

In this section we shall thus very briefly summarise the basic ideas of the software development process model that these volumes are based upon.

1.4.2 The Triptych Process Model

Figure 1.2 highlighted what we shall refer to as the triptych phase process model. Figure 1.3 diagrammed an iterative process model for part of domain development. Figure 1.4 illustrates a concurrent process model for part of requirements development.

In the next sections we shall summarise the triptych process model. Throughout it is useful to keep in mind our remarks (Sect. 1.3.6) on iterative and concurrent phases, stages and steps of development.

1.5 Conclusion to Chapter 1

It is time to complete this long introductory survey chapter.

1.5.1 Summary

We have introduced crucial aspects of our approach to software development.

- *Definitions of software engineering:* First, in Sect. 1.1 we brought in "old" and "new" definitions and characterisations of "what is software engineering".
- *The triptych of software engineering:* Then, in Sect. 1.2 we surveyed the three key phases of our unique approach to software development: domain engineering, requirements engineering and software design.
- *Phases, stages and steps of software development:* In Sect. 1.3 we reviewed these three phases and further suggested stages and steps of development within these phases and stages. We invite the reader to recapitulate the stages.
- *Software development process models:* And in Sect. 1.4.1 we very briefly broached the topic of process models, in particular the one brought forward by this book. This process model will be enlarged upon in subsequent chapters, notably Chap. 16, Chaps. 24, 30 and 31.

1.5.2 What Will Be Covered Later?

Naturally, this chapter has only provided a glimpse of things to come.

- *Domain Engineering:* Part IV will cover domain engineering in "excruciating" detail.
- *Requirements Engineering:* Part V will cover requirements engineering in "painstaking" detail.
- *Software Design:* And Part VI will cover software design in a somewhat more superficial manner!

1.6 Bibliographical Notes

Section 1.1 referred to several leading textbooks on software engineering. Those and others are by the following authors:

- Ian Sommerville [338]
- Roger S. Pressman [284]
- Shari Lawrence Pfleeger [275]
- Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli [121]
- Watts S. Humphrey [175]
- Hans van Vliet [369]

If you do not have access to Vols. 1 and 2 of this series on Software Engineering, then we recommend the Ghezzi, Jazayeri and Mandrioli textbook [121]. In particular, Software Engineering, Vol. 2 makes our otherwise recommended access to [121] unnecessary.

To complement the present three volumes we strongly recommend Hans van Vliet's fine work [369]. Also, Humphrey's work, [175], is a good supplement to the present three volumes.

1.7 Exercises

1.7.1 On a Series of Software Developments

In this volume we will relate to a number of complete software developments. Some will be covered, here and there, in the body of chapters in this volume, or have already been partially touched upon in previous volumes (railways, etc.). Other aspects of this claimed complete development will be covered in the exercise sections of most chapters.

Common to this coverage is a number of rough sketches of specific application domains. Expressed briefly and phrased as problem questions to be solved these are:

1. *What is administrative forms processing?*
2. *What is an airport?*
3. *What is air traffic?*
4. *What is a container harbour?*
5. *What is a document system?*
6. *What is a financial services system?*
7. *What is freight logistics?*
8. *What is a hospital?*
9. *What is a manufacturing company?*
10. *What is the market?*
11. *What is a metropolitan area[11] tourism industry?*
12. *What is a railway system?*
13. *What is a university?*
14. *What is public administration?*
15. *What is a ministry of finance?*

Next we briefly give very rough sketches of each of these individual domains.

1. **What is administrative forms processing?**

Typically enterprises base part of their day-to-day operations (especially administration) on a small set of forms. These include *employment forms:* application, employment offer, offer acceptance or rejection, work exercise form, form(s) for reporting sick leave, leave with, or without pay, termination or notification forms, and so on; and *procurement forms:* product or service inquiry, product or service offers, requisition, receipt form, inspection (acceptance or

---

[11] Such cities as Singapore, Macau, Hong Kong, London, New York, Tokyo, Paris, etc. can be said to be 'Metropolitan Areas'.

rejection) form, payment form, etcetera. Each form basically contains pre-formatted fields, to be filled in, partially or fully. Each such partially filled in form may undergo several rounds of filling in and possibly, where needed, approvals (i.e., signatures).

2. **What is an airport?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of people (passengers), material (fuel, catering, luggage), aircraft, information (passenger, luggage, catering, fuel, servicing, etc., information), and control in an airport.

3. **What is air traffic?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the movements (start-up, take-off, flight, preparation for landing, possible holding (in holding areas), touch-down and taxiing) of aircraft — under the monitoring and control by ground, terminal, area and continental air traffic control towers.

4. **What is a container harbour?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of ships and cargo, into and out from a container harbour: ships arriving at a container harbour, ships having, possibly, to anchor for container quay place, ships unloading and loading containers, ships being detained for customs, illegal cargo, or lack of sea-worthiness reasons in a harbour, ships cleaning their fuel tanks in a harbour, and ships leaving harbour.

5. **What is a document system?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with documents: their creation as originals, at a certain time and location, their placement with (allocation to) people or file cabinets, their copying (whereby unique, distinct copies are made, with no two copies of the same document being the same due to their necessarily being copied at different times), their editing (whereby the document which is being edited — whether an original, a copy, or a version — becomes a version of the document it was "edited from"), their movement (i.e., transfer from persons or file cabinets to (other) persons or (other) file cabinets, all necessarily having different locations — or their movement because the person with whom a document is associated is carrying that document "around"), or their shredding.

6. **What is a financial services system?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with people, customers, using banks, insurance companies, stockbrokers and portfolio managers. Thus also the entities, functions, etc., of these phenomena need be described. Of special interest are transfers of securities instruments between banks, insurance companies, stockbrokers, the (assumed one) stock exchange, and portfolio managers.

7. **What is freight logistics?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with (1) people (senders) inquiring with logistics firms about and actually sending or receiving freight transports; (2) with logistics firms arranging such transportation with trucking companies, with freight train operators, with ship owners, and with air cargo companies — as well as logistics firms interacting with trucking and freight train depots, harbours and airports; with (3) trucks, trains, ships and aircraft unloading and loading freight at depots, harbours and airports, etc. A central concept is that of a *way bill* (or a *bill of lading*), which directs freight from its point of origin via intermediate hubs (depots, harbours, airports), to its final destination.

8. **What is a hospital?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of patients, visitors and healthcare workers, of materials (beds, medicine, etc.), information (patient medical records with update information on clinical tests, X-rays, ECGs, MR scans, CT scans, etc.) and control in a hospital. Thus patient treatment, as a process, and its interaction with other hospital processes needs to be narrated.

9. **What is a manufacturing company?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the flow of orders into and deliveries from a manufacturing company, as well as the flow of materials (parts), equipment (trucks, conveyor belts, etc.), information (sales orders, production orders, etc.), and control among and within the various departments of a manufacturing enterprise: marketing, sales and service, design, production floor (machines [lathes, saws, mills, planers, etc.] and their in and out trays, delivery trucks, etc.), parts and products warehouses, etc.

10. **What is the market?**

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with customers inquiring about, ordering, taking delivery, returning (rejecting), accepting and paying

for, merchandise — with, from, and to retailers, who again perform similar actions with wholesalers, who again perform similar actions with producers, and where distribution companies may be involved in deliveries from producers to wholesalers to retailers to consumers.

11. What is a metropolitan area tourism industry?

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the inquiry, arrival, flow and departure of people (tourists, conference-goers, business people) about, to, within and from a metropolitan area: between airports and hotels, and between hotels, restaurants, shops, museums, theatres, parks, and historic sights. Inquiry about and reservations of hotel rooms, restaurants (tables), theatres (tickets), the inquiry and buying of transport cards, what to buy, planning of shopping (itinerary), etc. — all are part of what a visitor to a metropolitan area undergoes, including possible visits to the dentist, medical doctor or hospital emergency room.

12. What is a railway system?

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the rail net (lines and stations), timetables, train traffic, passengers inquiring, buying tickets, cancelling or using tickets, etc. The lines and stations consists of rail units, signals, etc. Thus railway system personnel despatch and reschedule, maintain (clean, repair, etc.) trains, and personnel are rostered (i.e., assigned to train duties), and so on.

13. What is a university?

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with students, lecturers and administrators, students' and lecturers' courses cum classes, course descriptions, lecture plans, lecture rooms (exercise and occupancy), examinations, etc. Included is students applying for admission to a university and registering for courses; of lecturers preparing courses, posting information, lecture notes, etc., and actually giving lectures. You are to "add", i.e., join to the previous, all those "other things" that you associate with being a university.

14. What is public administration?

The basis for this group of exercises is the creation and administration of such laws which concern the daily life of citizens and whose effects they daily, weekly, monthly, yearly or just occasionally "suffer" or benefit from — as the case may be. For example, such laws as govern social welfare, healthcare benefits, taxes and excise, building codes and land zoning (regulations), and so on. Creation of these laws takes place in parliament. Ministries prepare drafts of

laws to be put before parliament. Parliamentary committees discusses these laws and may recommend changes or adoption by the parliament. Parliament discuss these laws and eventually adopts the laws with which this group of exercises are concerned. Ministries formulate and ministers sign rules and regulations on how civil servants are to administrate these laws. Public administration divisions may further amend these rules and regulations with respect to particular interpretation. And so on.

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with the creation, all the way through parliament, ministries, public administration divisions down to the individual offices with which the citizens interact. Citizens may wish to know about the history of the law: what motives there were for creating the law in the first place, what discussions took place in parliamentary subcommittees, in parliament (i.e., how it was enacted), what rules and regulations were formulated by ministerial offices and which administrative procedures were formulated, and the practice of these by public administration divisions, and so on. Finally, and, as concerns the intention of the law, citizens need to interact with the law by requesting, for example, benefits from it, by submitting, for example, reports, and so forth.

15. What is a ministry of finance?

Domain descriptions of this topic shall identify and describe the entities, functions upon, and events and behaviours in connection with a ministry of finance's taxation and budget departments and the treasury.

In contrast to the above 14 domain outlines — where we relied on your own prior, albeit only superficial, knowledge of those domains — we shall elucidate the "workings" of the ministry of finance somewhat more.

A ministry of finance's perception of the nation in which it serves is that it is hierarchically organised: the state (s), the (nonoverlapping) provinces (p_i), the (nonoverlapping) districts (within provinces, d_{i_j}), and the (nonoverlapping) communes (cities, townships, villages, etc., $c_{i_{j_k}}$) within provinces — such that all provinces "make up" the state $(\{p_1, p_2, \ldots, p_i, \ldots, p_p\} = s)$, all districts of a province "make up" that province $(\{d_{i_1}, d_{i_2}, \ldots, d_{i_\iota}, \ldots, d_{i_d}\} = p_i)$, and all communes of a district "make up" the district $(\{c_{i_{\iota_1}}, c_{i_{\iota_2}}, \ldots, c_{i_{\iota_i}}, \ldots, c_{i_{\iota_c}}\} = d_{i_\iota})$.

Now the main functions of a ministry of finance wrt. the taxation and budget departments and the treasury are as follows:

(1) Annually an order is issued — by the ministry of finance taxation department — whereby the corresponding taxation departments of each province (of the state), each district within each province (of the state), and each commune within each district (etc., etc.) are to assemble, gather, obtain, by census or otherwise, statistical data, that is "the assessment data". These data represent "best guesses" of the basis for tax revenue (such as personal income, sales (for sales tax purposes), fees (for services rendered by province, district or commune authorities), etc.). From the communes this kind of data is

communicated (perhaps in simplified, summary form) to the district of that commune, and likewise from district to province, and to state. These communications must take place before certain dates $(D_{a_{c \to d}}, D_{a_{d \to p}}, D_{a_{p \to s}})$.

(2) More or less simultaneously an order is issued — by the budget department of the ministry of finance — whereby each ministry $(m_\mu$, incl. the ministry of finance, $m_f)$ is to set up a budget, B_{m_μ}, for next year's activities (i.e., expenditures) E_{m_μ}. The ministry of finance sets an initial ceiling I_{m_μ} (of so many millions of, say, dollars) for respective ministries' expected incomes. The various ministries contribute their (possibly negotiated) budgets for next year to the ministry of finance by a certain date $D_{\to m}$. A twist to this budgeting process may occur if the ministry of finance judges, well before $D_{\to m}$, but after $D_{a_{p \to s}}$, that the assessment data warrants either a downward (pessimistic), or an upward (optimistic), adjustment of the income I_{m_μ}. The submitted budget B_{m_μ} must balance within the possibly adjusted set income ceiling I_{m_μ}. The various ministries also have "shadow" budget departments in each province, district and commune.

(3) The parliament then negotiates and eventually, in time for the next year, passes the national budget, B_s, as assembled from all ministries' individual budgets B_{m_μ}.

(4) The budget B_s is subdivided into province, district and commune expenditures.

(5) Finally, the next fiscal year arrives, and the ministry of finance taxation department requests the taxation departments (of provinces, districts and communes) to regularly gather all relevant taxes and regularly send appropriate proportions of these taxes to the corresponding commune, district, province and state treasuries. Thus some proportion of a commune tax revenue goes to that commune's treasury, and the rest to the district treasury. As districts, independently of communes, also gather taxes, their income derives from these taxes and from the communes, and its outlay goes locally, to the district treasury and the treasury of its province, and so on.

1.7.2 Introductory Remarks

Three remarks are in order:

- *Selected Topic:* When in the following we mention a selected topic, we are referring to any one of the 15 problems listed in Sect. 1.7.1.
- *Informal/Formal:* When using the present volume in an informal course on software engineering, answers to the below questions need only be given informally, i.e., in precise natural language text (e.g., English). When using the present volume in a formal course you are to present both a precise natural language text and formulas to support that text.
- *Incremental/Evolutionary Solutions:* For each of the exercises — and, in principle, you need solve all of them — you may try your mind and hand on it. But since it is rather early in this volume and since, in particular,

much material on how to really solve these exercises is given in almost all chapters that follow you are expected to review your solution to the below exercises, one review, ideally, for each of the many coming chapters!

1.7.3 The Exercises

Exercises in subsequent chapters will repeat the first nine exercises, but in more elaborate forms.

Exercise 1.1 Domain Entities: For the fixed topic, selected by you, list some dozen or so domain entities. Give suitably short names for their types and describe these, whether simple or composite, and, if composite, describe their composition.

Exercise 1.2 Domain Functions: For the fixed topic, selected by you, list some half dozen or so domain functions: Give suitably short names to these functions, and describe their signatures, that is, which arguments they "take", and what results in the "yield".

Exercise 1.3 Domain Events: For the fixed topic, selected by you, list some half dozen or so domain events. Give suitably short names to these events, and describe them briefly.

Exercise 1.4 Domain Behaviours: For the fixed topic, selected by you, list, say, three behaviours. Give suitably short names to these behaviours, and describe them briefly.

   Hint: Think of behaviours as processes, i.e., of a behaviour as "one" process. Then describe that behaviour as it may also interact, or communicate, thus exemplifying events, with other behaviours.

Exercise 1.5 Domain Requirements: For the fixed topic, selected by you, list, say three or four, domain requirements. Describe them briefly and informally.

Exercise 1.6 Interface Requirements: For the fixed topic, selected by you, list two or three interface requirements. Describe them briefly and informally.

Exercise 1.7 Machine Requirements: For the fixed topic, selected by you, list one machine requirements for each of the "standard" areas: performance, dependability, maintenance, platform, and documentation. Describe them briefly and informally.

Exercise 1.8 Software Architecture Design: For the fixed topic, selected by you, attempt, admittedly rather prematurely, to sketch a software architecture — say in terms of (briefly specified) boxes and (briefly specified) arrows, where boxes denote single-thread processes and arrows denote interactions (messages) between processes. Do this only informally.

Exercise 1.9 Software Component Design: This exercise is a continuation of Exercise 1.8. For the architecture sketch given, by you, in answer to Exercise 1.8, single out one or two "boxes" and specify their data structures and functions. Do this only informally.

Exercise 1.10 The Triptych Process Model: Without referring back to Sect. 1.4, try write down, for yourself, what the essence is of the triptych process model. That is, phases and stages. Name these. Try to sketch some process model diagrams.