Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Topic 1
The Triptych Paradigm

- The **prerequisite** for following this part of the lecture 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.

Delineations of Software Engineering
"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 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 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:

- *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.

- *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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

IEEE Std. 610.12–1990

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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Shari Lawrence Pfleeger, 2001

> 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli, 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Accreditation Board for Engineering and Technology (ABET)

> 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Hans van Vliet

- *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.*

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Our View: What Is Software Engineering?

- We shall 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

- ⋆ 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).

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- In the next part of the lecture we shall examine these three concepts:
  - ⋆ domain engineering,
  - ⋆ requirements engineering and
  - ⋆ software design.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Topic 2
The Triptych of Software Engineering
Theme of this Lecture

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

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

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- So, from descriptions of the application domain
  - ⋆ we construct prescriptions of the requirements;
- and from prescriptions of the requirements
  - ⋆ we design the software.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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

- 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 lectures 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- The indefinite class of “things” to which computing may be applied, i.e., application domains (see next).
- The infinite class of anything else that does not satisfy the above characterisations. Examples are: philosophy, politics, poetry, etc. ■

Characterisation 1.2 By an *application domain* we shall understand anything to which computing may be applied. ■

[0] 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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

- 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.

- 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.
- 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.

■

Characterisation 1.3 By a *domain* we mean an application domain. ■

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.

- 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" *transportation* as a domain.

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

- 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.

- 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.

- The applications of software within the machining (metal-working) manufacturing sector individually define application domains, and together "define" *machining (metal-working) manufacturing* as a domain.

■

Topic 3
Domain Engineering
General

- We shall give a brief characterisation of what we mean by domain engineering.

Characterisation 1.4 (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 1.5 (II) By *domain description* we shall also mean the process of domain capture, analysis and synthesis, and the document which results from that process. ■

Characterisation 1.6 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:
  - ⋆ entities,
  - ⋆ functions,
  - ⋆ events and
  - ⋆ 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.

- 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.

■

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*

The lecturer explains the above.

• Functions

- Phenomena or concepts could be
- *functions* that apply to entities and
  - ⋆ either test for some property,
  - ⋆ observe some subentity, i.e., yield a data value that is "computed" from such entities, or
  - ⋆ 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

■

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!
■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

• 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.

■

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**

• 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.

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()

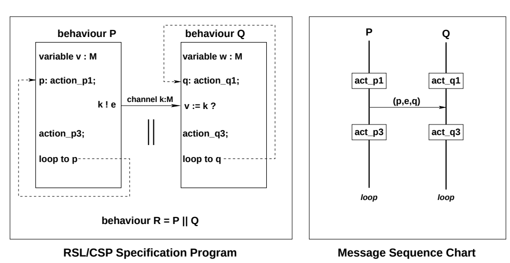

Figure 1.1: Informal process diagrams

- 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` and `RSL/CSP`. And we refer to Vol. 2, Chap. 13 for a thorough coverage of MSC.

Example 1.8 *Railway Entities, Functions, Events and Behaviours:* Our example 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.

type
  N, L, S, U
value
  is_Linear, is_Switch, is_Crossover: U → **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.

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;
  - ⋆ the departure of the train from the station ;
  - ⋆ the ride of the train down the line , including the acceleration and deceleration of the train ;
  - ⋆ the arrival of the train at the next station, its stopping at a platform;
  - ⋆ and the alighting of passengers.


```
type
  Sn, Train
value
  train_ride: Sn* → N → Train → N × 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
```

■

- 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.
- 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.

- 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′′′′′)
```

■

- We shall treat the concepts of phenomena and concepts in a later lecture.
- 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 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 in the next lecture, and treat domain engineering in detail in later lectures we shall just mention a few things.
  - ⋆ 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.
  - ⋆ 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.

  - ⋆ 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.*
  - ⋆ And we expect a domain description to be a basis for what — in other contexts than software engineering — is known as *business process reengineering.*

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 student will be well prepared to help contribute research and development on such a theory.
- But to do it properly, the student 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.

Topic 4
Requirements Engineering

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

The Machine

- We introduce the term machine.

Characterisation 1.7 By *machine* we shall understand a combination of hardware and software that is the target for, or result of, computing systems development. ■

- 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 1.8 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

. ■

- 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 1.9 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 1.10 By a *requirements prescription* we mean the process — and the document which results from the process — of requirements capture, analysis and synthesis. ■

Characterisation 1.11 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:
  - ⋆ *business process reengineering*,
  - ⋆ *domain requirements*,
  - ⋆ *interface requirements*, and
  - ⋆ *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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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.

Topic 5
Software

Characterisation 1.12 By *software* we understand

- not only *code* that may be the basis for executions by a computer,
- but also its full *development documentation*:
  - ⋆ the stages and steps of *application domain description*,
  - ⋆ the stages and steps of *requirements prescription*,
  - ⋆ 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
- a comprehensive collection of *supporting documents:*
  - ⋆ *training manuals*,
  - ⋆ *installation manuals*,
  - ⋆ *user manuals*,
  - ⋆ *maintenance manuals*, and
  - ⋆ *development* and *maintenance logbooks.*

So, software, as documentation, comprises many parts. ■

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 1.13 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 1.14 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 1.15 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. ■

Component Structure and Component Design

Characterisation 1.16 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 1.17 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),
- and new (for components since the early 1990s).

Characterisation 1.18 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. ■

Discussion 1.1 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 → tj, f1(ai) ≡ $\mathcal{C}_1$(ai)
    f2: tk → tℓ, f2(ak) ≡ $\mathcal{C}_2$(ak)
    ...
    fn: tp → tq, fn(ap) ≡ $\mathcal{C}_1$(ap)
  hide: fi, fj, ..., fk
end module

Syntax, semantics and pragmatics: Lecturer to explain. ■

Characterisation 1.19 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 1.2 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.).

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Systems, Design and Refinement

Characterisation 1.20 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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Discussion 1.3 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.

■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.21 By *software system design* we understand

- the determination,
  - ⋆ from domain requirements and from some interface requirements, of the software architecture, or
  - ⋆ 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

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- the determination of the main (system) types of values,
- the determination of the basic structuring of and facilities (i.e., functions) offered by components, and
- 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*:

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.22 By *software system refinement* we understand

- the stagewise and stepwise transformation of
  - ⋆ an abstract specification
  - ⋆ into increasingly more concretely specified modules and components

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.23 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 1.24 By *concrete specification* we mean one that uses specification cum programming constructs that prescribe efficient executions. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Components and Modules, Design and Refinement

Characterisation 1.25 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

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- 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 lecture, separation of concerns makes it easier to tackle many diverse issues.
- Hence our development needs to proceed in stages and steps of refinement.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Characterisation 1.26 By *component refinement* we shall usually understand:

- a concretisation of the usually initially abstractly defined component types,
- a concretisation of the usually initially abstractly specified initialisations of component variables, and, possibly,
- the refinement of the component modules

. ■

Characterisation 1.27 By *module refinement* we understand:

- a concretisation of the usually initially abstractly defined module types,
- a concretisation of the usually initially abstractly specified initialisations of component variables and
- a concretisation of the usually initially abstract module function definitions —
- 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 student to have a necessary background in programming
  - ⋆ we shall not cover this topic in these lectures.

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 the next lecture,
  - ⋆ and in some detail in several later lectures.
- But, these lectures will not present *anywhere near* a fully satisfactory treatment of the software design problem.

- That is neither the aim nor the objectives of these lectures.
  - ⋆ First, we have assumed some knowledge, education and training of the student.
  - ⋆ Second we have to refer to special topic texts for detailed software design principles, techniques and tools.

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 students, 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 lecture 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 lecture.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Topic 6
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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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 parts of the lecture we shall briefly review possible decompositions.
- Each represent a concern; together they represent *separation of concerns.*

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Linear, Cyclic and Parallel Development Activities

- In the next lecture parts 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 lecture 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Phases of Software Development

- We have already introduced the main three phases of software development:
  - ⋆ domain development,
  - ⋆ requirements development, and
  - ⋆ 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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 a later lecture.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- 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 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.28 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 1.29 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, prescribing) 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 1.30 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 ($k, k', \ldots, k''$). ■

Example 1.10 *Stage Kinds:*

- Examples of domain stage kinds are:
  - ⋆ (d_1) business 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 reengineering,
  - ⋆ (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 1.4 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 lecture, 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 lecture, 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 1.31 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 1.32 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

. ■

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*.

- ⋄ These include: modelling *business process* facets of a domain,
- ⋄ modelling *intrinsics* facets of a domain,
- ⋄ modelling possible *support technology* facets of a domain,
- ⋄ modelling possible *management and organisation* facets of a domain,
- ⋄ modelling possible *rules and regulations* facets of a domain,
- ⋄ modelling possible *script* facets of a domain, and
- ⋄ modelling possible *human behaviour* facets of a domain.

- We shall briefly characterise these stages shortly, and
- we cover them in detail in a much later lecture.

Characterisation 1.33 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, an enterprise, a public administration, an infrastructure component,
- are given — each from possibly a number of stakeholder perspectives

. ■

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 1.34 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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.35 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 by human power, as operated, from afar by mechanical wires, as operated electromechanically, or as operated electronically and electromechanically (say, in interlocking mode). ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.36 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- Domain management
  - ⋆ ensures that these decisions are passed on to the ("lower") levels of management, and to "floor" staff,
  - ⋆ makes sure that such orders, as they were, are indeed carried out,
  - ⋆ handles undesirable deviations in the carrying out of these orders cum decisions,
  - ⋆ and "backstops" complaints from lower management levels and from floor staff

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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.). ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.37 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- Hence we mean the "lines of command":
  - ⋆ who does what and
  - ⋆ who reports to whom, both
    - ◇ administratively, and
    - ◇ functionally

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.38 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

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.39 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

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.40 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

. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

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 lecture,
- but refer to much later lecture for more details.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Topic 7
Requirements Development

- From an earlier lecture we repeat some of the below characterisations.

Characterisation 1.41 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". ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

Characterisation 1.42 By *requirements prescription* we mean the process — and the document which results from the process — of requirements capture, analysis and synthesis. ■

Characterisation 1.43 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. ■

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Stages of Requirements Development

- We see four different kinds of requirements:
  - ⋆ *business process reengineering*,
  - ⋆ *domain requirements*,
  - ⋆ *interface requirements*, and
  - ⋆ *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 1.44 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 a later lecture:
  - ⋆ introduction of some new and removal of some old *support technologies*,
  - ⋆ introduction of some new and removal of some old *management and organisation structures*,
  - ⋆ introduction of some new and removal of some old *rules and regulations*,
  - ⋆ 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
  - ⋆ related *scripts*.

• Domain Requirements

Characterisation 1.45 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 a later lecture:
  - ⋆ 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 1.46 By *interface requirements* we understand

- such requirements, to software, which are expressed
- in terms of domain phenomena shared between the environment and the machine

. ■

Department of Computer Science and Engineering, Institute of Informatics and Mathematical Modelling, Technical University of Denmark, Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- We consider five kinds of interface requirements which will be covered in a later lecture:
  - ⋆ *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 1.47 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 a later lecture:
  - ⋆ *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 lecture,
- but refer to a much later lecture for more details.

Topic 8
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 a much, much later lecture.

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 these lectures, assumed that the student 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 student has studied, or will study, such topics as:
  - ⋆ Dijkstra's *Discipline of Programming*,
  - ⋆ Gries' *Science of Programming*,
  - ⋆ Reynolds' *Craft of Programming*,
  - ⋆ Hehner's *Logic and Practical Theory of Programming*,
  - ⋆ Jones' *Systematic Software Development*,
  - ⋆ Morgan's *Refinement Calculus* or
  - ⋆ Back and von Wright's (earlier) *Refinement Calculus*.
- In a (much) later lectures we shall, however, briefly illustrate notions of software design refinement.

Topic 9
Discussion: Phases, Stages and Steps
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.

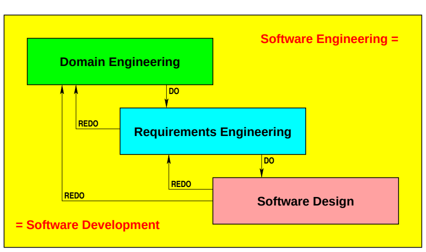

Figure 1.2: A diagramming of the iterative triptych phase development

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 developers 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

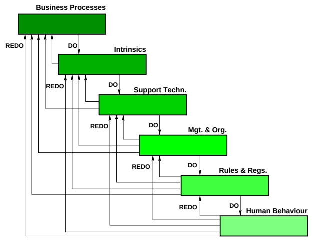

Figure 1.3: A diagramming of iteration of domain stages

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

- Typically
  - ⋆ the domain requirements,
  - ⋆ the interface requirements and
  - ⋆ the machine 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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

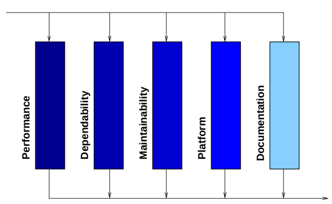

Figure 1.4: A diagramming of stage concurrency of machine requirements

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Topic 10
The Triptych Process Model — A First View
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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Characterisation 1.48 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 lecture we shall thus very briefly summarise the basic ideas of the software development process model that these lectures are based upon.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

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.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

- In the next parts of the lecture we shall summarise the triptych process model.
- Throughout it is useful to keep in mind our remarks the previous lecture on iterative and concurrent phases, stages and steps of development.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark
DTU

Topic 11
Conclusion to Chapter 1

It is time to complete this long introductory survey chapter.

Summary

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

- *Definitions of software engineering:* First we brought in “old” and “new” definitions and characterisations of “what is software engineering”.
- *The triptych of software engineering:* Then we surveyed the three key phases of our unique approach to software development: domain engineering, requirements engineering and software design.

Department of Computer Science and Engineering
Institute of Informatics and Mathematical Modelling
Technical University of Denmark
Richard Petersens Plads, DK-2800 Kgs.Lyngby, Denmark

- *Phases, stages and steps of software development:* 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:* 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 lectures.