(-1. )

**Start of Lecture 1: SUMMARY & INTRODUCTION**

(-1. )

# FROM DOMAINS TO REQUIREMENTS

**April 16–30, 2010 Lectures, TUWien**

**Dines Bjørner**
**Fredsvej 11, DK-2840 Holte, Denmark**
**bjorner@gmail.com, www.imm.dtu.dk/˜db**

## 0. Abstract

- We shall present core aspects of the Triptych approach to software engineering.
- The benefits from deploying this approach are that we both achieve *the right software* and *software that is right*[Boehm 1981]).
- *The right software* is software that meets all of the customers' expectations and only those.
- *Software that is right* is software that is correct with respect to specific requirements prescriptions.

(0. Abstract )

- Experience has shown that using also the formal techniques part of the Triptych approach has lead to projects that are on time and at initially estimated costs.
- To achieve **the right software** we "prefix" the phase of requirements engineering with a phase of domain engineering – and these lecture slides will present core aspects of domain engineering.
- To achieve **software that is right** we do two things:
  - (i) "derive" requirements prescriptions from domain descriptions and software design from requirements prescriptions – and this these lecture slides will present core aspects of a somewhat different approach to requirements engineering, and
  - (ii) formulate descriptions and prescriptions both informally, in precise, say English narratives, and formally. The latter is not shown in these lecture slides.

(0. **Abstract** )

- The "somewhat" different approach to requirements engineering, however, and as we shall see, fits reasonably "smoothly" with current requirements engineering approaches[van Lamsweerde].
- Precursors of the 'triptych' approach was used in DDC's 44 man-year Ada Compiler development project [Bjørner and Oest].
  - That project was on time and at cost,
  - and time and cost were significantly below those of other commercial Ada compiler developments .

(0. **Abstract** )

- The 'triptych' approach has been in partial use since the early 1990s,
  - including at the United Nations University's International Institute for Software Technology (`www.iist.unu.edu`).
  - Young software engineers, while being tutored by UNU-IIST's science & engineering staff,
    - ∗ domain engineered,
    - ∗ requirements engineered
    - ∗ and software designed (incl. implemented)[2002, LNCS 2757]
      - · trustworthy software systems
      - · that have met customer expectations –
      - · with what seems be substantially fewer man-power resources than usually experienced and within planned time limits.

(0. **Abstract** )

- Domain engineering, in the sense of these lecture slides,
  - is offered as a means to help secure that software engineers deliver *the right software* –
  - where formalisation of relevant stages and steps of software development helps secure that *the software is right*.
- In these lecture slides we shall present the essence of a software development *triptych*:
  - from domains
  - via requirements
  - to software design.

(0. **Abstract** )

- We emphasize the two first phases: *domain engineering* and *requirements engineering*.
  - We show the pragmatic stages of the construction of *domain descriptions*: the facets of
  - *intrinsics*,
  - *support technologies*,
  - *rules & regulations*,
  - *script (licenses and contracts)*,
  - *management & organisation*, and
  - *human behaviour*.

(0. **Abstract** )

- And we show how to construct main facets of *requirements prescriptions*:
  - *domain requirements* and
  - *interface requirements*.
- In this respect we focus in particular on the *domain requirements* development stages of
  - *projection*,
  - *instantiation*,
  - *determination* and
  - *extension*.

## Lecture Notes for T U Wien, April 2010

- The present version of this document is intended as the "written" support for my April 2010 lectures at the Technical University of Vienna. Austria.
  - The **www.imm.dtu.dk/~db/wien** web page gives details.
  - From there you can see that Sects. 1–5 covers 5 lectures
  - and that Appendix A covers 8 lectures.

(0. **Abstract** )

- To examples of sections 2–4 we have "added" formalisations.
- These formalisations are in the `RAISE` specification languages `RSL`.
- And I have additionally added an extensive appendix,
  - An RSL Primer[1],
- so that students can also learn `RSL`, the specification language for a **r**igorous **a**pproach to **i**ndustrial **s**oftware **e**ngineering, `RAISE`.
- The primer contains many examples which expands on the examples of sections 2–4.

[1]a small introductory book on a subject

(0. **Abstract** )

## "Formalisation–Parametrised" Examples and Primer

- The formalisations of the examples of sections 2–4 could as well be expressed in one of the other prominent formal specifications languages current at this time (April 7, 2010), for example:
  - `Alloy`,
  - `Event B`,
  - `VDM-SL` or
  - `Z`.
- It could be interesting
  - if this little book could entice
  - my Alloy, Event B, VDM-SL and Z colleagues
  - to "rewrite/reformulate" the formal parts of all examples
  - into their main tool of formal expression (besides mathematics).
- I would be very willing to engage in such a project
  - having the aim of making my and their notes
  - Internet-based and thus publically available.

## On Studying the Examples

- In order to learn to **write** poems one must **read** poetry.
- In order to learn th **write** formal specifications one must **read** formal specifications
- We have ourselves found
  - that even if students attend pedagogically and didactically exciting and sound lectures
  - they must still, in the quiet of their study room, without listening to Ipod (or the like),
  - carefully study the examples we are presenting.
- And we are presenting many examples, 49 in all !
  - To begin with little explanation is given of the formulas.
  - Instead we rely on the student's ability to relate the numbered formulas to the numbered annotation textst.
  - As from Appendix  we present a schematic syntax and informal semantics of the spexification language, RSL, used in these lectures.
- Students are well adviced in studying all examples.

## On Course Lectures Based on these Slides

## 1. Introduction
### 1.1. Some Observations

- Current software development,
  - when it is pursued in a state-of-the-art,
  - but still a conventional manner,
  - starts with requirements engineering and
  - proceeds to software design.
- Current software development practices
  - appears to be focused on processes
  - (viz.: "best practices': tools and techniques').

- An aeronautics engineer to be hired by `Airbus` to their design team for a next generation aircraft must be pretty well versed in applied mathematics and in aerodynamics.
- A radio communications engineer to be hired by `Ericsson` to their design team for a next generation mobile telephony antennas must be likewise well versed in applied mathematics and in the physics of electromagnetic wave propagation in matter.
- And so forth.

- Software engineers hired for the development of software
  - for hospitals,
  - or for railways,
- know little, if anything, about
  - health care,
  - respectively rail transportation (scheduling, rostering, signalling, etc.).
- The `Ericsson` radio communications engineer can be expected to understand Maxwell's Equations, and to base the design of antenna characteristics on the transformation and instantiation of these equations.

- It is therefore quite reasonable to expect the domain-specific software engineer to understand proper, including formal descriptions of their domains:
  - for railways cf. `www.railwaydomain.org`,
  - and for pipelines `pipelines.pdf`,
  - logistics `logistics.pdf`
  - and for container lines `container-paper.pdf` –
  - all at `www.imm.dtu.dk/~db/`.
- For the Vienna course the above — and other such — examples are temporarily blocked !

- The process knowledge and "best" practices of the triptych software engineering
  - is well-founded and takes place in
  - the context of established domain model
  - and an established, carefully phrased (and formalised) requirements model.
- The 24 hour 7 days a week trustworthy operation of many software systems
  - is so crucial that utmost care must be taken
  - to ensure that they
    - ∗ fulfill all (and only) the customers expectations
    - ∗ and are correct.

- Barry Boehm has coined the statement: *it is the right software and the software is right.*
- Extra care must be taken to ensure those two "rights".
- And here it is not enough to only follow current "best process, technique and tool practices".

## 1.2. A Triptych of Software Engineering

**Dogma:**

- *Before we can design software*
- *we must have a robust understanding of its requirements.*
- *And before we can prescribe requirements*
- *we must have a robust understanding of the environment,*
  - *or, as we shall call it, the domain in which the software is to serve*
  - *and as it is at the time such software is first being contemplated.*

- In consequence we suggest that software, “ideally”[2], be developed in three phases.

[2]Section [Item 5] will discuss renditions of “idealism”!

- First a phase of **domain engineering.**
  - In this phase a reasonably comprehensive description is constructed from an analysis of the domain.
  - That description, as it evolves, is analysed with respect to inconsistencies, conflicts and relative completeness.
  - $\mathcal{P}$roperties, as stated by domain stakeholders, are proved with respect to the domain description ($\mathcal{D}\models\mathcal{P}$).
  - This phase is the most important, we think, when it comes to secure the first of the two “rights”: that we are on our way to develop the right software.

- Then a phase of **requirements engineering.**
  - This phase is strongly based on an available, necessary and sufficient domain description.
  - Guided by the domain and requirements engineers the *requirements stakeholders* points out which domain description parts are
    - ∗ to be kept (*projected*) out of the *domain requirements*,

    and for those kept in,
    - ∗ what *instantiations*,
    - ∗ *determinations*
    - ∗ and *extensions* are required.

  - Similarly the requirements stakeholders, guided by the domain and requirements engineers, informs as to
    - ∗ which domain *entities: simple, actions, events* and *behaviours*
    - ∗ are *shared* between the domain and the *machine*,
  - that is, the *hardware* and the *software* being required.
- In these lectures we shall only very briefly cover aspects of *machine requirements*.

- And finally a phase of **software design.**
  - We shall not cover this phase in these lectures –
  - other than saying this:
    - ∗ the design is “derived” from the requirements model.

- To ensure that the software being developed is right, that is, correct,
  - we can then rigorously
  - argue, informally,
  - or formally – test, model check and/or prove,
  - that the $\mathcal{S}$oftware is correct
    - ∗ with respect to the $\mathcal{R}$equirements
    - ∗ in the context of the $\mathcal{D}$omain:
    - ∗ $\mathcal{D}, \mathcal{S} \models \mathcal{R}$.

## 1.3. What are Domains ?

- By a domain we shall here understand a universe of discourse,
  - an area of nature subject to laws of physics and study by physicists, or
  - an area of human activity subject to its interfaces with other domains and to nature.
- There are other domains – which we shall ignore.
- We shall focus on the human-made domains.

- “Large scale” examples are
  - *the financial service industry: banking, insurance, securities trading, portfolio management, etc.*;
  - *health care: hospitals, clinics, patients, medical staff, etc.*;
  - *transportation: road, rail/train, sea/shipping*, and *air/aircraft transport (vehicles, transport nets, etc.)*;
  - *oil and gas systems: pumps, pipes, valves, refineries, distribution, etc.*

- "Intermediate scale" examples are
  - *automobiles: manufacturing* or *monitoring and control, etc.*;
  - *heating systems*;
  - *heart pumps*;
  - etc.
- The above explication was "randomised":
  - for some domains, to wit, *the financial service industry*, we mentioned major functionalities,
  - for others, to wit, *health care*, we mentioned major entities.

## 1.4. What is a Domain Description ?

- By a *domain description* we understand a description of
  - the *simple entities*,
  - the *actions*,
  - the *events* and
  - the *behaviours*

  of the domain, including its *interfaces* to other domains.
- A domain description describes the domain **as it is.**
- A domain description does not contain requirements let alone references to any software.

- A description is *syntax*.
- The meaning (*semantics*) of a domain description is usually a set of *domain models*.
- We shall take domain models to be *mathematical structures (theories)*.
- The form of domain descriptions that we shall advocate "come in pairs": precise, say, English text alternates with clearly related formula text.

## 1.5. Description Languages

- Besides using
  - as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements,
  - we "pair" such narrative elements with corresponding enumerated clauses of a formal specification language.
- We shall be using the `RAISE` Specification Language, `RSL` in our formal texts.
- But any of the model-oriented approaches and languages offered by
  - `Alloy`,
  - `Event B`,
  - `VDM` and
  - `Z`,

  should work as well.

- No single one of the above-mentioned formal specification languages, however, suffices.
- Often one has to carefully combine the above with elements of
  - `Petri Nets`,
  - `CSP: Communicating Sequential Processes`,
  - `MSC: Message Sequence Charts`,
  - `Statecharts`,
  - and some temporal logic, for example
    - either `DC: Duration Calculus`
    - or `TLA+`.

## 1.6. Contributions of these Lectures

- We claim that the major contributions of the Triptych approach to software engineering as presented in this paper are the following:
  - (1) the clear *identification* of domain engineering, or, for some, its clear *separation* from requirements engineering;
  - (2) the *identification* and *'elaboration'* of the pragmatically determined domain *facets* of *intrinsics*, *support technologies*, *management and organisation*, *rules and regulations*, *scripts (licenses and contracts)* and *human behaviour* whereby 'elaboration' we mean that we provide principles and techniques for the construction of these facet description parts;

- (3) the *re-identification* and *'elaboration'* of the concept of *business process re-engineering* on the basis of the notion of *business processes*;
- (4) the *identification* and *'elaboration'* of the technically determined *domain requirements facets* of *projection*, *instantiation*, *determination*, *extension* and *fitting* requirements principles and techniques – and, in particular the *"discovery"* that these requirements engineering stages are strongly dependent on necessary and sufficient domain descriptions ;

and

- (5) the *identification* and *'elaboration'* of the technically determined *interface requirements facets* of *shared entity*, *shared action*, *shared event* and *shared behaviour* requirements principles and techniques. We claim that the facets of (2, 3, 4) and (5) are all *novel*.

## 1.7. Structure of Lectures

- Before going into some details on domain enginering and requirements engineering
- cover the basic concepts of specifications, whether domain descriptions or requirements prescriptions.
- These are:
  - entities,
  - actions,
  - events and
  - behaviours.

**End of Lecture 1: SUMMARY & INTRODUCTION**