The Rôle of Domain Engineering in Software Development

DINES BJØRNER[1 2 3]

We outline the concept of domain engineering and explain the main stages of developing domain models. Requirements engineering is then seen as an intermediate stage where domain models are "transformed" into requirements prescriptions. Software Design concludes development — and we comment on software correctness with respect to both requirements prescriptions and domain descriptions. We finally overview this new phase of development: domain engineering and argues its engineering virtues while relating them to object-orientedness, UML, component-based SE, aspect-orientedness and intentional software development.

1 Introduction

1.1 Triptych Dogma

Traditionally, today, software development starts with expressing requirements and then goes on to design software from the requirements. In this paper we shall explain why **this is not good enough**. First we express **the triptych dogma**: *Before software can be developed we must understand its requirements. Before requirements can be expressed we must understand the domain in which the software (plus the hardware) is to reside. Therefore we must first develop an understanding of that domain.*

1.2 Triptych of Software Development

We therefore develop software as follows: First we develop a domain description. Then, from the domain description, we develop a requirements prescription. And, finally, from the requirements prescription we develop a software design. While developing we verify and validate the domain description, verify and validate the requirements prescription with respect to the domain description and requirements stakeholder statements, and verify the software design with respect to the requirements and domain specifications.

[1] School of Information Science, Japan Advanced Institute of Science and Technology, 1-1, Asahidai, Tatsunokuchi, Nomi, Ishikawa, Japan 923-1292

[2] Dept. of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kgs.Lyngby, Denmark

[3] Invited keynote paper for the Special Interest Group of Software Engineering (SIGSE), Information Processing Society of Japan (IPSJ) IPSJ/SIGSE Software Engineering Symposium 2006, Oct. 21, Tokyo, Japan

2 An Example: Railway Nets

Before we delve into too much "talking about" domain descriptions let us show a tiny example. The example covers a description of just a small part of a domain: the net of rails of a railway system. There are only two parts to the description: A systematic, "tight", precise English narrative, and "its" corresponding formalsation in the specification language, RSL of RAISE [13, 14, 5, 6, 7]. We do not show "all the work" that precedes establishing this description.

2.1 Narrative

1. A railway net is a net of mode railway.
2. Its segments are lines of mode railway.
3. Its junctions are stations of mode railway.
4. A railway net consists of one or more lines and two or more stations.
5. A railway net consists of rail units.
6. A line is a linear sequence of one or more linear rail units.
7. The rail units of a line must be rail units of the railway net of the line.
8. A station is a set of one or more rail units.
9. The rail units of a station must be rail units of the railway net of the station.
10. No two distinct lines and/or stations of a railway net share rail units.
11. A station consists of one or more tracks.
12. A track is a linear sequence of one or more linear rail units.

13. No two distinct tracks share rail units.
14. The rail units of a track must be rail units of the station (of that track).
15. A rail unit is either a linear, or is a switch, or a is simple crossover, or is a switchable crossover, etc., rail unit.
16. A rail unit has one or more connectors.
17. A linear rail unit has two distinct connectors. A switch (a point) rail unit has three distinct connectors. Crossover rail units have four distinct connectors (whether simple or switchable), etc.
18. For every connector there are at most two rail units which have that connector in common.
19. Every line of a railway net is connected to exactly two distinct stations of that railway net.
20. A linear sequence of (linear) rail units is an acyclic sequence of linear units such that neighbouring units share connectors.

2.2 Formalisation

```
type
1. RN  = {| n:smN • obs_M(n)=railway |}
2. LI  = {| s:S • obs_M(s)=railway |}
3. ST  = {| c:C • obs_M(c)=railway |}
   Tr, U, K

value
   4. obs_LIs: RN → LI-set
   4. obs_STs: RN → ST-set
   5. obs_Us: RN → U-set
   6. obs_Us: LI → U-set
   8. obs_Us: ST → U-set
   11. obs_Trs: ST → Tr-set
   15. is_Linear: U → Bool
   15. is_Switch: U → Bool
   15. is_Simple_Crossover: U → Bool
   15. is_Switchable_Crossover: U → Bool
   16. obs_Ks: U → K-set

   20. lin_seq: U-set → Bool
   lin_seq(us) ≡
      ∀ u:U • u ∈ us ⇒ is_Linear(u) ∧
      ∃ q:U* • len q = card us ∧ elems q = us ∧
         ∀ i:Nat • {i,i+1} ⊆ inds q ⇒ ∃ k:K •
            obs_Ks(q(i)) ∩ obs_Ks(q(i+1)) = {k} ∧
         len q > 1 ⇒ obs_Ks(q(i)) ∩ obs_Ks(q(len q)) = {}

axiom
4. ∀ n:RN •
         card obs_LIs(n) ≥ 1 ∧ card obs_STs(n) ≥ 2

6. ∀ n:RN, l:LI •
         l ∈ obs_LIs(n) ⇒ lin_seq(l)

7. ∀ n:RN, l:LI •
         l ∈ obs_LIs(n) ⇒ obs_Us(l) ⊆ obs_Us(n)

8. ∀ n:RN, s:ST •
         s ∈ obs_STs(n) ⇒ card obs_Us(s) ≥ 1

9. ∀ n:RN, s:ST •
         s ∈ obs_LIs(n) ⇒ obs_Us(s) ⊆ obs_Us(n)

10. ∀ n:RN,l,l′:LI •
         {l,l′}⊆obs_LIs(n)∧l≠l′⇒obs_Us(l)∩ obs_Us(l′)={}

10. ∀ n:RN,l:LI,s:ST •
         l ∈ obs_LIs(n)∧
         s ∈ obs_STs(n)⇒obs_Us(l)∩ obs_Us(s)={}

10. ∀ n:RN,s,s′:ST •
         {s,s′}⊆obs_STs(n)∧s≠s′⇒obs_Us(s)∩ obs_Us(s′)={}

11. ∀ s:ST•card obs_Trs(s)≥1

12. ∀ n:RN,s:ST,t:Tr • s ∈ obs_STs(n)∧t ∈ obs_Trs(s)⇒lin_seq(t)

13.  ∀ n:RN,s:ST,t,t′:Tr •
         s ∈ obs_STs(n)∧{t,t′}⊆obs_Trs(s)∧t≠t′
            ⇒ obs_Us(t) ∩ obs_Us(t′) = {}

18. ∀ n:RN • ∀ k:K •
         k ∈ ∪{obs_Ks(u)|u:U•u ∈ obs_Us(n)}
            ⇒card{u|u:U•u ∈ obs_Us(n)∧k ∈ obs_Ks(u)}≤2

19. ∀ n:RN,l:LI •  l ∈ obs_LIs(n) ⇒
   ∃ s,s′:ST • {s,s′} ⊆ obs_STs(n) ∧ s≠s′ ⇒
      let sus=obs_Us(s),sus′=obs_Us(s′),lus=obs_Us(l) in
         ∃ u,u′,u″,u‴:U • u ∈ sus ∧
            u′ ∈ sus′ ∧ {u″,u‴} ⊆ lus ⇒
            let sks=obs_Ks(u),sks′=obs_Ks(u′),
                  lks=obs_Ks(u″),lks′=obs_Ks(u‴) in
      ∃!k,k′:K•k≠k′∧sks ∩ lks={k}∧sks′ ∩ lks′={k′}
         end end
```

2.3 References

We can refer to more complete descriptions of railway domains: There is a "grand challenge" network: www.railwaydomain.org. There are some publications: [8, 3, 9, 4] and a "book": "The TRain Book": http://www.railwaydomain.org/book.ps A "larger", more encompassing description of transportation nets in general has been started: www.jaist.ac.jp/~bjorner/transnets.

3 Domains

3.1 Examples of Domains

There are basically three kinds of domains, sometimes called application domains or business domains. These are: base systems software such as compilers, operating systems, database management systems, data communication systems, etc., "middleware" software packages: Web servers, word/text processing systems, etc., and the real end-user applications. That is, software for airlines and airports; banks and insurance companies; hospitals and healthcare in general; manufacturing; the market: consumers, retailers, wholesaler, the distribution chain; railways; securities trading: exchanges, traders and brokers; and so forth.

3.2 Domain Description

3.2.1 What Is a Domain Description

What do we mean by a domain description? By a domain description we mean a document, or a set of documents which describe a domain as it is, with no references to, with no implicit requirements to software. The informal language part of a domain description is such that a reader from, a stakeholder of that domain recognizes that it is a faithful description of the domain. So, a domain description describes something real, something existing. Usually a domain description describes not just a specific instance of a domain, but a set of such, not just one bank but a set of "all" banks!

3.2.2 How Is a Domain Description Expressed?

How is a domain description expressed? By a domain description we mean any text that clearly designates an phenomena, an entity, or a function (which when applied to some entities become an action), or an event, or a behaviour (i.e., a sequence of actions and events) of the domain, or a concept defined, i.e., abstracted from other domain descriptions.

Domain Descriptions Are Indicative: Domain descriptions described what there is, the domain as it is, not as the stakeholder would like it to be.

Informal and Formal Domain Descriptions: Domain descriptions come in four, mutually supportive forms, three informal texts and one formal: rough sketches are informal, incomplete and perhaps not very well structured descriptions; terminologies — explain all terms: names of phenomena or concepts of the domain; narratives — "tell the story", in careful national/natural and professional language; and formal specification — formalises in mathematics the narrative and provides the ultimate answer to questions of interpretation of the informal texts. Initial descriptions necessarily are rough sketches. They help us structure our thinking and generate entries for the terminology. Terminologies, narratives and formalisations are deliverables.

3.2.3 Existing Descriptions

Are there accessible examples of domain descriptions? Yes, there are descriptions now of railway systems, transportation nets, financial service industries, hospital healthcare, airports, air traffic, and many other domains. Some are in the form of MSc theses, some are part of PhD theses. Some fragment domain descriptions are published in journal papers, some in conference papers. And several are proprietary — having been developed in software houses. For all the cases implied above the descriptions include formal descriptions. At JAIST students are currently developing such formalisations in CafeOBJ.

3.3 Domain Engineering

3.3.1 How to Construct a Domain Description?

In the following we will briefly outline the steps — stakeholder identification, acquisition, acquisition analysis, modelling, verification and validation — that it takes to construct a domain description.

3.3.2 Domain Stakeholders

All relevant stakeholders must be identified. For, say a railway domain, typical stakeholder groups are: the owners of a railway, the executive, strategic, tactical and operational management — that is several groups, the railway ("blue collar") workers — station staff, train staff, line staff, maintenance staff, etc. potential and actual passengers and relatives of these, suppliers of goods and services to the railway, railway regulatory authorities, the ministry of transport, and politicians "at large". Liaison with representatives of these stakeholder groups must be regular — as some, later, become requirements stakeholders.

3.3.3 Domain Acquisition

The domain engineer need acquire information ("knowledge") about the domain. This should be done pursuing many different approaches. Two most important are: reading literature, books, pamphlets, Internet information, about the domain; and eliciting hopefully commensurate information from stakeholders. From the former the domain engineer is (hopefully) able to formulate a reasonable questionnaire. Elicitation is then based on distributing and the domain engineers personally "negotiating" the questionnaire with all relevant stakeholders. The result of the latter is a set of possibly thousands of domain description units.

3.3.4 Domain Analysis

Description Unit Attributes: The domain description units are then subjected to an analysis. First they must be annotated with attribute designators such as entity, function, event, behaviour; and intrinsics, support technology, management & organisation, rules & regulation, script, human behaviour: and source of information, date, time, locations, who acquired, etc.: etcetera.

Problems: Analysis of the description units involve looking for and resolving incompleteness, inconsistency and conflicts.

Concepts: Analysis of the description units primarily aims at discovering concepts, that is notions that generalises a class of phenomena, and for discovering meta-concepts, that is "high level" abstractions that together might help develop as generic and hence, it is believed, applicable, reusable domain model as possible.

3.3.5 Domain Modelling Proper

Domain modelling is then based on the most likely database handled domain description units. The domain model, that is, the meaning of the domain description must capture: **intrinsics**: that which is at the basis of, or common to all facets, **technologies** which **support** phenomena of the domain, **management & organisation**: who does what, who reports to whom, etc., **rules & regulations** — governing human behaviour and use of technologies — sometimes manifested in scripts, and **human behaviour** — diligent, sloppy, delinquent or outright criminal. It must all be described!

3.3.6 Domain Verification

Verification — only feasible when a formal description is available — proves properties of the domain model not explicitly expressed, and serves to ensure that we **got the model right**.

3.3.7 Domain Validation

Validation is the human process of "clearing" with all relevant stakeholders that we **got the right model**.

3.3.8 Discussion

Thus domain engineering is a highly professional discipline. It requires many talents: interacting with stakeholders, ability to write beautifully and concisely, ability to formalise and analyse formal specifications, etc. Domain engineers are also researchers: physicists of human made universes.

3.4 Professionalism of SE

Mechanical engineers are fully versant in the laws of the domain for which they create artifacts (Newton's Laws, etc.). Radio engineers, when hired, a fully versant in Maxwell's Equations — laws governing their application domain. And so it goes for all other professional engineers than SEs. Sometimes their basis in theoretical computer science is rather shaky. And always they know little or nothing about the business domain for which they develop software: financial services, transportation, healthcare. It is not becoming of a professional. Domain engineering brings professionalism into SE.

4 "Deriving" Requirements

4.1 "The Machine"

By "the machine" we understand that computing system hardware and software which is to be inserted in the domain in order to support some activities of the domain.

4.2 Three Kinds of Requirements

There are basically three kinds of requirements: **domain requirements** — those which can be expressed solely using terms of the domain; **machine requirements** — those which can be expressed without using terms of the domain; and **interface requirements** — those which must be expressed using terms both of the domain and the machine. We treat these in a slightly changed order.

4.2.1 Domain Requirements

One can rather simply, that is very easily, develop the domain requirements from the domain description. Here is how it is done: With the various requirements stakeholders, one-by-one "go through", i.e., co-read the domain description, line-by-line while seeking answers to the following sequentially order questions: **Projection**: should this "line" (being read) be part of the requirements? **Instantiation**: if so, should what is described be instantiated from it usually generic form? **Determination**: and — if it is expressed in a loose or non-deterministic, i.e., under-specified manner, be made more determinate? **Extension**: Are there potential phenomena or concepts of the domain which were not described because they were infeasible in the domain — if so the machine make it feasible? **Fitting**: Are there other requirements development, elsewhere, with which the present one could be "interfaced"?

The result of a domain requirements development phase is a (sizable) document that is expected to (functional) requirements specify that which can be computed.

4.2.2 Interface Requirements

The interface requirements development stage now starts by identifying all the phenomena that are to be **shared** between the domain ("out there") and "the machine" ("in here")!

The shared phenomena and (now machine) concepts are either entities, functions, events or behaviours. Each such shared phenomenon leads, respectively, to interface requirements concerning **entities**: bulk data (database) initialisation and refreshment; **functions**: man/machine dialogue concerning computational progress; **events**: handling of interrupts and the like; and **behaviours**: logging and replay monitoring and control. For each of the four classes due consideration is paid wrt. use of visual displays, tactile instruments ("mouse", keyboard, stylos, sensitive screens or pads, etc.), audio equipment: sound recognition and production, smell, taste, and physics measurements.

The result of an interface requirements development phase is a (sizable) document, adjoint to the domain requirements document, that is expected to (user) requirements specify that which can be interchanged (input/output between man or machine and machine).

4.2.3 Machine Requirements

Machine (or systems) requirements deal with such matters as **performance**: dealing with concerns of storage and response times (hence equipment "numbers"); **dependability**: accessibility, availability, reliability, fault tolerance; security, etc.; **maintainability**: adaptive, perfective, corrective and preventive maintenance; **portability**: development, demonstration, execution, and maintenance platform issues; **documentation**: installation, training, user, maintenance and development documents.

The machine requirements are developed "against" a check-list of all these requirements possibilities and focusing on each line of the domain requirements and interface requirements documents.

The result of a machine requirements development phase is yet a (sizable) document — adjoint to the domain and interface requirements documents — that is expected to (system) requirements specify that which can be also implemented.

4.3 Further SE Professionalism

Requirements engineering has been made easy. The domain is very stable. There is now a clear, well-defined path from domain models to requirements models. The adage, i.e., the common observation, *"requirements always change"* need no longer be true. For a software engineer to command the process of creating domain models and comfortably transforming them into requirements with input from requirements stakeholders signifies professional SE.

5 Software Design

To round off the triptych approach to software development, such as advocated here, we briefly mention that the requirements specifications (which prescribe **what**), are now the basis for refinement into software designs (the **how**). We shall not go into these aspects in this paper but refer the reader to the rather fully comprehensive [5, 6, 7] other than recalling

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

Correctness of $\mathcal{S}$ with respect to $\mathcal{R}$ can be proven using recorded assumptions about the $\mathcal{D}$.

6 Rôle of Domain Descriptions

6.1 A Science Motivation

One rôle of domain modelling is that of obtaining and recording understanding. The domain engineer is a researcher: studies "new territory". Just as physicists for centuries have studied "mother" nature, so it is high time we study the universes of man-made structures.7

6.2 A Engineering Motivation

Another rôle of domain modelling is the engineering one. We present an elegantly formulated summary of the rôle of domain descriptions in software engineering. It was expressed by Sir Tony Hoare — in an exchange of e-mails in July 2006.

6.2.1 Tony Hoare's Assessment

"There are many unique contributions that can be made by domain modelling.

1. The models describe all aspects of the real world that are relevant for any good software design in the area. They describe possible places to define the system boundary for any particular project.
2. They make explicit the preconditions about the real world that have to be made in any embedded software design, especially one that is going to be formally proved.
3. They describe the whole range of possible designs for the software, and the whole range of technologies available for its realisation.
4. They provide a framework for a full analysis of requirements, which is wholly independent of the technology of implementation.
5. They enumerate and analyse the decisions that must be taken earlier or later any design project, and identify those that are independent and those that conflict. Late discovery of feature interactions can be avoided."

6.2.2 Structuring of Rôles

We rephrase our and Tony's formulation as follows: Domain models represent theories of human organisations — and as such they are interesting in and by themselves. Domain models also represent a first major result in a software development: In proving correctness of $\mathcal{S}$oftware with respect to $\mathcal{R}$equirements assumptions are repeatedly made about the $\mathcal{D}$omain. We can summarise the engineering rôle of domain engineering as follows:

$$\mathcal{D}, \mathcal{S} \models \mathcal{R}$$

6.3 Conventional SE Paradigms

We are presenting a novel theory-based approach to software development. This, the triptych approach has to compete with conventional software development methods, that are currently en vogue in the industry. Let us try relate these conventional methods to what we are advocating: Examples are: **object-oriented programming** (OO), **unified modeling language** (UML), **component-based programming** (CBSE), **aspect-oriented software engineering** (AOS), and **int∃ntion∀l software development** (∃∀).

I believe that you will find that some of the strengths of OO, CBSE and especially ∃∀ are occurring "naturally" in both the domain engineering and in the "derived" requirements engineering. That is, we wish to point out how we can understand basic traits of OO, CBSE, AOP and especially ∃∀ and and thus explain why these approaches have won adherents.

6.3.1 OO Programming

We assume that the audience is well familiar with the OO programming paradigm. We shall briefly list some of the OO "quarks"[4]: **class**, **object**, **method**, **message passing**, **inheritance**, **encapsulation**, **abstraction** and **polymorphism**. Several of these "quarks" are specifically oriented at programming rather than specifying.

The foremost feature of OO that are of interest in the context of domain and requirements engineering is the concept of objects. OO objects are also present in the triptych approach to formalisation: the encapsulation of related entities, functions and events in a module, with that module now denoting possible behaviours. Many object modelling techniques, essentially all discovered by the Simula 67 originators, carry over to domain and requirements specification.

So first we remark that OO primarily addresses programming assuming given requirements, whereas the triptych approach advocated here address the entire span of software develop ment from, and significantly focused on domain modelling, and on deriving requirements from domain descriptions. The triptych approach advocated here does not prescribe which combination of coding paradigms you may wish to use: OO, AOP, CBSE, XP (Expert Programming), etc.

6.3.2 UML

UML, to us, is a confused approach to software development. Yet, however, it has some appealing features. It mixes informal textual specifications with several graphical techniques (Petri nets, MSCs, Statecharts). It supposedly has a "powerful" data schema concept (class diagrams). In UML you are programming more than you are specifying.

But UML, to us, has problems: It has no notion of abstraction. It thus has no notion of stepwise development — necessary to conquer complexity with a phase, and necessary to separate the necessarily separated phases of domain modelling (the why), requirements modelling (the what), and software design (the how) One cannot reason logically over UML specifications. It is placed somewhere between requirements and software design specifications. For these reasons we cannot take UML serious and we wonder why professional software engineers do?

[4]Term used by Ms Deborah J. Armstrong in naming fundamental OO concepts (http://en.wikipedia.org/wiki/Object-oriented_programming)

6.3.3 CBSE: Component-based SE

A basic concern of CBSE is building software systems from reusable components. There seems to be many different "schools" of CBSE. Being in Japan it is reasonable to follow that of the CBSE Group, Fukazawa Laboratory, Waseda University[5]: In a narrow sense,a software component is defined as a unit of composition, and can be independently exchanged in the form of an object code without source codes. The internal structure of the component is not available to the public. The characteristics of the component-based development are the following:

- Black-box reuse
- Reactive-control and component's granularity
- Using RAD (rapid application development) tools
- Contractually specified interfaces
- Introspection mechanism provided by the component systems
- Software component market (CALS)

It is natural to model and implement components in an object-oriented paradigm/language. Therefore, when understanding the component, the traditional techniques in the OO paradigm such like OO framework, design patterns, architecture patterns and meta-patterns are very important

We shall now try evaluate basic tenets of CBSE in light of the triptych approach. We strongly advice that search for components start at the level of domain modelling. Once identified in the domain, requirements may project, instantiate, determinate, extend and fit them in a variety of ways. Such domain-to-requirements operations may be expressed in the form of adjusting suitable parameters to a schema/module/object like abstract formalisation of the domain component. From here on many of the intriguing issues of CBSE can be better understood, either as basic abstraction-to-concretisation refinements or as simple coding tricks. CBSE certainly has a rôle in making the triptych approach even more viable.

6.3.4 AOP: Aspect-oriented Programming

A basic concern of AOP is that some code is scattered or tangled, making it harder to understand and maintain. It is scattered when one concern (like logging) is spread over a number of modules (e.g., classes and methods). That means to change logging can require modifying all affected modules. Modules end up tangled with multiple concerns (e.g., account processing, logging, and security). That means changing one module entails understanding all the tangled concerns.

AOP attempt to aid programmers in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization. AOP does so using primarily language changes, while AOSD (aspect-oriented software development) uses a combination of language, environment, and methodology.

Separation of concerns entails breaking down a program into distinct parts that overlap in functionality as little as possible. All programming methodologies — including procedural programming and object-oriented programming — support some separation and encapsulation of concerns (or any area of interest or focus) into single entities. For example, procedures, packages, classes, and methods all help programmers encapsulate concerns into single entities. But some concerns defy these forms of encapsulation. Software engineers call these cross cutting concerns, because they cut across many modules in a program.

So, really, AOS, is primarily a coding discipline. So why do we bring it up here, in a presentation which is primarily not about coding, but about domains and requirements. Some software engineers (may) ask: *What is relation between the triptych approach and AOS?*Our answer is: The cross cutting concerns appear not to be caused by domain requirements, nor by interface requirements, but by machine requirements. Thus problems of cross-cutting concern appears to be introduced in a serious, but not really user-oriented stage of requirements development. This "discovery" might enlighten researchers in the AOS community.

[5]http://www.fuka.info.waseda.ac.jp/Project/CBSE

6.3.5 ∃∀: Intentional Software Development

The intentional software development paradigm is the creation of Charles Simonyi[6].

It appears that little if any literature is readily accessible [35, 1, 36, 37]. So we shall resort to quoting from *Intentional Software*'s Web page (http://intentsoft.com/technology/glossary.html). The quotes are in sans serif.

Domain: A domain is an area of business, engineering or society for which a body of knowledge exists. Examples include health care administration, telecommunications, banking, accounting, avionics, computer games and software engineering.

Domain Code: Domain code is the structured code to represent the **intentions** contributed by subject matter experts for the problem being solved. Domain code includes contributions from all domains relevant to the software problem. Domain code is not executable (as traditional source code is - by compilation or interpretation), but it can be transformed into an implementation solution when it is input to a generator that has been programmed to perform that transformation process.

Domain-Oriented Development: Domain-oriented development is the process of separating the contributions of subject matter experts and programmers to the maximum extent so that generative programming can be applied to structured domain code. This greatly simplifies improvements to the domain and implementation solutions.

Domain Schema: A domain schema is a schema for a specific domain. The domain schema defines the domain terminology and any other information that is needed — for the intentional editor and generator to work — such as parameters, help text, default values, applicable notations and other structure of the domain code. Domain schemas are created by the subject matter experts and programmers working together, and are expressed in a schema language.

Domain Terminology: Domain terminology means the terms of art (words with a special meaning) in a domain, for example "claim payment" in health care administration. Domain terminology is important because it is the usual way to express **intentions**. Broadly speaking, terminology includes notations normally used by a subject matter expert, such as tables, flowcharts and other symbols. The meaning of the terms is part of the domain knowledge that is shared between subject matter experts and programmers to the extent necessary and ultimately designed into domain schemas and the generator.

Discussion: Intentional software development, it should be clear from the above builds on a number of software development tools which are provided with domain description-like information and which can then significantly automate code generation. Other than that shall neither comment nor speculate on Charles Simonyi's characterisations.[7] We believe that the reader can easily see the very tight relations to the triptych phases of development. We find them fascinating and will try communicate our own observations to Charles Simonyi before commenting in depth. ∃∀ certainly has a rôle in making the triptych approach even more viable.

[6]Intentional Software, Bellevue, Washington, USA; http://intentsoft.com

[7]Well, I cannot, of course, refrain from saying that my students have founded a number of Danish software companies whose corporate asset it is that they generate code for application of the domain specific area which is their company's hallmark.

7 Conclusion

7.1 What Have We Achieved?

We have outlined two of the major phases of an extended)read: new) approach to software development: domain engineering — primarily — and requirements engineering — as it relates to domain engineering. We have not really covered the relation of requirements engineering to software design, i.e., programming — other than now saying: software design is then a further refinement of the requirements, and, well, read next! We have then related this, the triptych approach to some current programming and software development paradigms: OO, CBSE, AOS — as mostly programming cum coding paradigms, and **Intentional Software Development** which, to us, have a much clearer and cleaner understanding of the domain, with the domain intentions, when being edited, probably having the editing stage amount to, or being based on some form of requirements development. We shall certainly look into Intentional Software Development more closely.

7.2 What More Need be Achieved?

Well, on the basis of three volumes,

- D. Bjørner: *Software Engineering, Vol. 1: Abstraction and Modelling* (Springer, 2006)
- D. Bjørner: *Software Engineering, Vol. 2: Specification of Systems and Languages* (Springer, 2006)
- D. Bjørner: *Software Engineering, Vol. 3: Domains, Requirements and Software Design* (Springer, 2006)

supported by almost 6,000 lecture slides and supported extensive RAISE tools, what more could one wish?

The answer is: more tools, tools to support documentation: creation, editing, versioning, etc.; tools to support domain and requirements acquisition and analysis; tools to extend the use of RAISE; as well as tools to **integrate** the **formal use** of RAISE with the formal use of Petri nets [25, 30, 31, 32, 33], MSCs [22, 23, 24], LSCs [12, 20, 26], Statecharts [16, 17, 19, 21, 18], Duration Calculus and TLA+ [38, 39], [27, 28, 29], etc. [2, 15, 11, 10, 34], further theorem proving, proof checking, model checking and testing tools.

7.3 Acknowledgments

The author gratefully acknowledges the support of JAIST and of his colleague, Prof. Kokichi Futatsugi. The author also gratefully acknowledges Prof. Hironori Washizaki, National Institute of Informatics , for so kindly having invited me to write and present this paper at the IPSJ/SIGSE Software Engineering Symposium in Tokyo in Oct. 2006.

References

[1] W. Aitken, B. Dickens, P. Kwiatkowski et al: Transformation in intentional programming. In: *Fifth International Conference on Software Reuse, ICSR'98* (Victoria, Canada)

[2] K. Araki, A. Galloway, K. Taguchi, editors. *IFM 1999: Integrated Formal Methods*, volume 1945 of *Lecture Notes in Computer Science*, York, UK, June 1999. Springer. Proceedings of 1st Intl. Conf. on IFM.

[3] D. Bjørner: Formal Software Techniques in Railway Systems. In: *9th IFAC Symposium on Control in Transportation Systems*, ed by E. Schnieder (2000) pp 1–12

[4] D. Bjørner: Dynamics of Railway Nets: On an Interface between Automatic Control and Software Engineering. In: *CTS2003: 10th IFAC Symposium on Control in Transportation Systems* (Elsevier Science Ltd., Oxford, UK 2003)

[5] D. Bjørner: *Software Engineering, Vol. 1: Abstraction and Modelling* (Springer, 2006)

[6] D. Bjørner: *Software Engineering, Vol. 2: Specification of Systems and Languages* (Springer, 2006)

[7] D. Bjørner: *Software Engineering, Vol. 3: Domains, Requirements and Software Design* (Springer, 2006)

[8] D. Bjørner, C. George, S. Prehn. *Scheduling and Rescheduling of Trains*, chapter 8, pages 157–184. Industrial Strength Formal Methods in Practice, Eds.: Michael G. Hinchey and Jonathan P. Bowen. FACIT, Springer–Verlag, London, England, 1999.

[9] D. Bjørner, C.W. George, S. Prehn: Computing Systems for Railways — A Rôle for Domain Engineering. Relations to Requirements Engineering and Software for Control Applications. In: *Integrated Design and Process Technology. Editors: Bernd Kraemer and John C. Petterson* (Society for Design and Process Science, P.O.Box 1299, Grand View, Texas 76050-1299, USA 2002)

[10] E.A. Boiten, J. Derrick, G. Smith, editors. *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, London, England, April 4-7 2004. Springer. Proceedings of 4th Intl. Conf. on IFM. ISBN 3-540-21377-5.

[11] M.J. Butler, L. Petre, K. Sere, editors. *IFM 2002: Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, Turku, Finland, May 15-18 2002. Springer. Proceedings of 3rd Intl. Conf. on IFM. ISBN 3-540-43703-7.

[12] W. Damm, D. Harel: *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design **19** (2001) pp 45–80

[13] C.W. George, P. Haff, K. Havelund et al: *The RAISE Specification Language* (Prentice-Hall, Hemel Hampstead, England 1992)

[14] C.W. George, A.E. Haxthausen, S. Hughes et al: *The RAISE Method* (Prentice-Hall, Hemel Hampstead, England 1995)

[15] W. Grieskamp, T. Santen, B. Stoddart, editors. *IFM 2000: Integrated Formal Methods*, volume of *Lecture Notes in Computer Science*, Schloss Dagstuhl, Germany, November 1-3 2000. Springer. Proceedings of 2nd Intl. Conf. on IFM.

[16] D. Harel: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming **8**, 3 (1987) pp 231–274

[17] D. Harel: *On Visual Formalisms.* Communications of the ACM **33**, 5 (1988)

[18] D. Harel, E. Gery: *Executable Object Modeling with Statecharts.* IEEE Computer **30**, 7 (1997) pp 31–42

[19] D. Harel, H. Lachover, A. Naamad et al: *STATEMATE: A Working Environment for the Development of Complex Reactive Systems.* Software Engineering **16**, 4 (1990) pp 403–414

[20] D. Harel, R. Marelly: *Come, Let's Play – Scenario-Based Programming Using LSCs and the Play-Engine* (Springer-Verlag, 2003)

[21] D. Harel, A. Naamad: *The STATEMATE Semantics of Statecharts.* ACM Transactions on Software Engineering and Methodology (TOSEM) **5**, 4 (1996) pp 293–333

[22] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992.

[23] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[24] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC), 1999.

[25] K. Jensen: *Coloured Petri Nets*, vol 1: Basic Concepts (234 pages + xii), Vol. 2: Analysis Methods (174 pages + x), Vol. 3: Practical Use (265 pages + xi) of *EATCS Monographs in Theoretical Computer Science* (Springer–Verlag, Heidelberg 1985, revised and corrected second version: 1997)

[26] J. Klose, H. Wittke: An Automata Based Interpretation of Live Sequence Charts. In: *TACAS 2001*, ed by T. Margaria, W. Yi (Springer-Verlag, 2001) pp 512–527

[27] L. Lamport: *The Temporal Logic of Actions.* Transactions on Programming Languages and Systems **16**, 3 (1995) pp 872–923

[28] L. Lamport: *Specifying Systems* (Addison–Wesley, Boston, Mass., USA 2002)

[29] S. Merz: *On the Logic of TLA+.* Computing and Informatics **22**, 1–2 (2003)

[30] C.A. Petri: *Kommunikation mit Automaten* (Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962)

[31] W. Reisig: *Petri Nets: An Introduction*, vol 4 of *EATCS Monographs in Theoretical Computer Science* (Springer Verlag, 1985)

[32] W. Reisig: *A Primer in Petri Net Design* (Springer Verlag, 1992)

[33] W. Reisig: *Elements of Distributed Algorithms: Modelling and Analysis with Petri Nets* (Springer Verlag, 1998)

[34] J.M. Romijn, G.P. Smith, J.C. van de Pol, editors. *IFM 2005: Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, Eindhoven, The Netherlands, December 2005. Springer. Proceedings of 5th Intl. Conf. on IFM. ISBN 3-540-30492-4.

[35] C. Simonyi: The Death of Computer Languages, the Birth of Intentional Programming. In: *NATO Science Committee Informatics Conference* (1995)

[36] C. Simonyi: *The Future is Intentional*. Computer **32**, 5 (1999) pp 56–57

[37] C. Simonyi: Intentional Programming: Asymptotic Fun? In: *Position Paper, SDP Workshop, Vanderbilt University* (December, 2001)

[38] C.C. Zhou, M.R. Hansen: *Duration Calculus: A Formal Approach to Real–time Systems* (Springer–Verlag, 2004)

[39] C.C. Zhou, C.A.R. Hoare, A.P. Ravn: *A Calculus of Durations.* Information Proc. Letters **40**, 5 (1992)

Contents