SYSTEMS ANALYSIS

DOMAIN SCIENCE AND ENGINEERING FROM
COMPUTER SCIENCE TO THE SCIENCES
OF INFORMATICS. PART I: ENGINEERING

D. Björner

Abstract. In this paper, we wish to advocate that departments of computer science put emphasis on teaching programming and software engineering based on formal methods; and more emphasis on research into formal methods for the trustworthy development of software. We also wish to advocate that the concepts of domain science and domain engineering become an indispensable part of the science of informatics and of software engineering.

Keywords: domain, domain description, domain engineering, domain modeling, software development, software engineering.

1. INTRODUCTION

The background postulates of this paper are the following: (i) half a century of computer science research may very well have improved our understanding of computing devices (automata etc.), but it has yet to contribute significantly to the quality of software products; (ii) our students, the future leading software engineers, those of them who go into industry rather than “remaining” in academia, are being mislead by too many foundational courses to believe that these are relevant for the practice of software engineering; (iii) a significant re-orientation of university teaching and research into both ‘computer science’ and software engineering must occur if we are to improve the relevance of ‘computer science’ to software engineering. In this paper we shall, unabashedly, suggest the kind of re-orientation that we think will rectify the situation alluded to in Items (i)–(iii).

1.1. Some Definitions of Informatics Topics

Let us first delineate our field of study. It first focuses on computer science, computing science, software, and software engineering.

Definition 1 (Computer Science). By computer science we shall understand the study and knowledge of the properties of the ‘things’ that can ‘exist’ inside computers: data and processes.

Examples of computer science disciplines are: automata theory (studying automata [finite or otherwise] and state machines [without or with stacks]), formal languages (studying, mostly the syntactic the “foundations” and “recognisability” of abstractions of computer programming and other “such” languages), complexity theory, type theory, etc.

Some may take exception to the term ‘things’ (and also to the term ‘exist’) used in the above and below definition. They will say that it is imprecise. That using the germ conjures some form of reliance on Plato’s Idealism, on his Theory of Forms. That is, “that it is of Platonic style, and thus, is disputable. One could avoid this by saying that these definitions are just informal rough explanations of the field of study and further considerations will lead to more exact definitions.” Well, it may be so. It is at least a conscious attempt, from this very beginning, to call into dispute and discuss “those things”. Part II of this paper (“A Specification Ontology and Epistemology”) has as one of its purposes to encircle the problem.

Definition 2 (Computing Science). By computing science we shall understand the study and knowledge of the how to construct the ‘things’ that can ‘exist’ inside computers: the software and its data.

Cf. personal communication, 12 Feb. 2010, with Prof. Mykola Nikitchenko, Head of Theory and Technology of Programming Department, Faculty of Cybernetics, National Taras Shevchenko University of Kyiv, Ukraine.

Technical University of Denmark, Lyngby, Denmark, bjorner@gmail.com. Translated from Kibernetika i Sistemnyi Analiz, No. 4, pp. 100–116, July–August 2010. Original article submitted April 9, 2010.

1060-0396/10/4604-0609 ©2010 Springer Science+Business Media, Inc.
Conventional examples of computing science disciplines are: algorithm design, imperative programming, functional programming, logic programming, parallel programming, etc. To these we shall add a few in this paper.

**Definition 3 (Software).** By software we shall understand not only the code intended for computer execution, but also its use, i.e., programmer manuals: installation, education, user and other guidance documents, as well all as its development documents: domain models, requirements models, software designs, tests suites, etc. “zillions upon zillions” of documents.

The fragment description of the example Pipeline System of this paper exhibits, but a tiny part of a domain model.

**Definition 4 (Software Engineering).** By software engineering we shall understand the methods (analysis and construction principles, techniques, and tools) needed to carry out, manage and evaluate software development projects as well as software product marketing, sales and service — whether these includes only domain engineering, or requirements engineering, or software design, or the first two, the last two or all three of these phases. Software engineering, besides documents for all of the above, also includes all auxiliary project information, stakeholder notes, acquisition units, analysis, terminology, verification, model-checking, testing, etc.

1.2. The Triptych Dogma

**Dogma 1 (Triptych).** By the triptych dogma we shall understand a dogma which insists on the following: Before software can be designed one must have a robust understanding of its requirements; and before requirements can be prescribed one must have a robust understanding of their domain.

**Dogma 2 (Triptych Development).** By triptych development we shall understand a software development process which starts with one or more stages of domain engineering whose objective it is to construct a domain description, which proceeds to one or more stages of requirements engineering whose objective it is to construct a requirements prescription, and which ends with one or more stages of software design whose aim it is to construct the software.

1.3. Structure of This Paper

In Sect. 2 we present a non-trivial example. It shall serve to illustrate the new concepts of domain engineering, domain description, and domain model. In Sect. 3 we shall then discuss ramifications of the triptych dogma. Then we shall follow-up, in Part II of this paper, on what we have advocated above, namely a beginning discussion of our logical and linguistic means for description, of “the kind of ‘things’ that can ‘exists’ or the things (say in the domain, i.e., “real world”) that they reflect”.

2. EXAMPLE: A PIPELINE SYSTEM

The example is to be read “hastily”. That is, emphasis, by the reader, should be on the narrative, that is, on conveying what a domain model describes, rather than on the formulas.

The example is that of domain modeling a pipeline system. Figure 1 show the planned Nabucco pipeline system.

2.1. Pipeline Basics

Figure 2 conceptualizes an example pipeline. Emphasis is on showing a pipeline net consisting of units and connectors (•). These are some non-temporal aspects of pipelines, nets and units: wells, pumps, pipes, valves, joins, forks, and sinks; net and unit attributes; and units states, but not state changes. We omit consideration of “pigs” and “pig”-insertion and “pig”-extraction units.

**Pipeline Nets and Units.**

1. We focus on nets, \( N \), of pipes, \( \pi : \Pi \), valves, \( v:V \), pumps, \( p:P \), forks, \( f:F \), joins, \( j:J \), wells, \( w:W \), and sinks, \( s:S \).
2. Units, \( u:U \), are either pipes, valves, pumps, forks, joins, wells or sinks.
3. Units are explained in terms of disjoint types of pipes, \( \Pi \), valves, \( V \), pumps, \( P \), forks, \( F \), joins, \( J \), wells, \( W \), and sinks, \( S \).

**type**

\[
\begin{align*}
1 \ & N, \Pi, \ V, \ PU, \ FO, \ JO, \ WE, \ SK \\
2 \ & U = \Pi \ | \ V \ | \ P \ | \ F \ | \ J \ | \ S \ | \ W \\
2 \ & \Pi = \text{mk}\Pi(\pi:\Pi) \\
2 \ & V = \text{mk}\Pi(\pi:\Pi) \\
2 \ & P = \text{mk}\Pi(\pi:\Pi) \\
2 \ & F = \text{mk}\Pi(\pi:\Pi) \\
2 \ & J = \text{mk}\Pi(\pi:\Pi) \\
2 \ & W = \text{mk}\Pi(\pi:\Pi) \\
2 \ & S = \text{mk}\Pi(\pi:\Pi) \\
\end{align*}
\]