Requirements Eng (2002) 7:172–175
Ownership and Copyright
© 2002 Springer-Verlag London Limited

Viewpoints

Domain Understanding is the Key to Successful System Development

Ray Offen
Division of ICS, Macquarie University, New South Wales, Australia

Developers all too often are in a hurry to start software design and construction and, as a result, have an unfortunate tendency to overlook, to their cost, the very significant advantages offered by developing a prior and adequate understanding of the surrounding application domain. There is an alternative, but one that is still not taken nearly as seriously as it should be.

Most practising engineers, be they civil, aeronautical, electrical, chemical, maritime or whatever, would accept without argument that a clear, concise and unambiguous understanding of a development project's specific application domain is an essential precursor to successful systems development. A civil engineer, tasked with constructing a hydro-electric dam across a river, would concentrate initially on investigating the geology of the area, drilling test cores, determining the river's flow characteristics, checking the seismic history and so on, before starting any serious work on specification and design. Similarly, designers of continental shelf oil drilling platforms spend a lot of their time carefully characterising local weather/storm patterns and the wave/swell dynamics of the surrounding ocean before starting to specify in any detail the requisite platform and rig. This process of domain exploration and characterisation, prior to detailed specification and design, is a central, generic activity that is taken very much for granted in all traditional engineering construction disciplines.

So, given the ever increasing centrality of software as the system component that mediates the bulk of the functionality, performance, safety and reliability in so many contemporary systems, it is remarkable indeed that the roles and utility of this initial *domain modelling* process, alluded to above, are generally so poorly understood and manifestly underutilised by the software development community. I may well be biased, given my background as a systems designer (computers, airborne radars, robotics, medical imaging. etc.) but I must confess that I still find it extremely disconcerting to talk to professional software developers who really do not seem to understand or appreciate what domain modelling involves, nor want to recognise its very specific importance in effective requirements engineering and risk management.

Consequently, for this particular *Viewpoints* column, I want to offer a few off-the-cuff comments on the nature and importance, to software and systems developers, of an adequate degree of domain understanding, usually represented via some form of domain modelling (or, less frequently, *domain engineering*). For present purposes, I shall make no distinction between the terms domain modelling and domain engineering, assuming them, to all intents and purposes, to mean fundamentally the same

Correspondence and offprint requests to: Ray Offen, Director of JRCASE, E6A 209, Division of ICS, Macquarie University, NSW 2109, Australia. Email: roffen@ics.mq.edu.au

thing. Only once domain characterisation becomes much more widely used and appreciated by software developers will it become worthwhile to debate the domain modelling versus domain engineering distinction, if, indeed, any exists.

By the term model, in this context, I mean an abstract representation of phenomena. Abstraction is central to the whole notion of models, as well as their derivation and use. A model only portrays aspects of world phenomena which are relevant to the model's purpose, and, as such, the model stands apart from both the phenomena it portrays and a person's conceptualisation of such phenomena. Given that software is characterised by a high degree of complexity, is, to all intents and purposes, invisible, and that we build software solely by describing it, the issue of how we use models to represent, give meaning to and share descriptions of software artefacts must be seen as absolutely fundamental. The three key models that, as a rule, we have to build in practice, are those corresponding to the application domain, the requirements specification and the system design (see Fig. 1). The specification occupies a special position in this triptych, as a domain descriptive theory, in that it describes, usually through a variety of models, phenomena shared by both the application domain and the resulting system.

My particular emphasis, on making use of models of the application domain, may seem eccentric to the current generation of software developers,[1] but it must be understood as being grounded very much in my own personal experience. Born before the first true digital, electronic, stored-program, computer ever flashed a 'run' light on its console, I spent a couple of decades designing and building a wide variety of computer systems before I finally became involved in software engineering in the mid-1980s. Over the years, in a variety of roles, I have been required to resolve a lot of problems with software[2] development projects that have gone off the rails. One of the things that has always surprised me, and continues to surprise me, is how often the concomitant root causes were, and continue to be, requirements related. More specifically, how frequently a project's difficulties involved an unfortunate lack of understanding, by the developers, of key characteristics

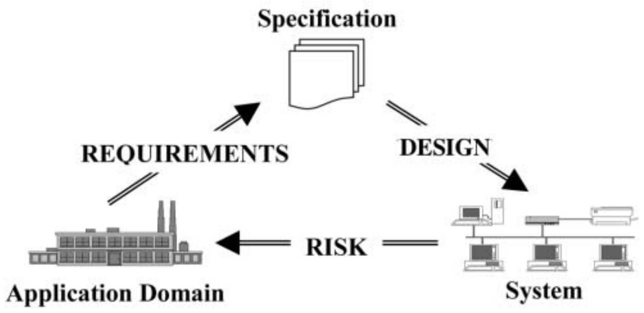

Fig. 1. An illustration of the three key questions that arise in effective systems engineering. Are the requirements (*abstraction*) appropriate for properly reflecting end-user tasks; is the design (*reification*) adequate for the purpose(s) in hand; and have we managed all of the risks inherent in the introduction of the new system into the application domain? Note that this diagram emphasises the central position occupied by the specification, as a domain descriptive theory of *both* the application system *and* its associated domain.

of the application domain, exemplifying again and again, in my mind, Michael Jackson's notion of 'denial by prior knowledge' [1].

Most of us would accept that successful software engineering is strongly predicated, in the first instance, as is all engineering, on effective specification. In fact, a specification can most conveniently be viewed as nothing more, and nothing less, than a domain-descriptive theory of the application system and its associated domain (Fig. 1). As a consequence, the activity of specification must always involve careful description, definition and model building, particularly of the application domain itself, which serves to ground the theory (i.e. the specification). So from this perspective, domain modelling becomes an essential exercise in theory building, as a vital precursor to effective specification, system construction and system delivery.

The question that immediately arises is that of how we should describe the domain. In that we build software solely by describing it, the issue of how we represent, give meaning to and share descriptions of software artefacts looms large indeed throughout the development process. Indeed, in software development we need to describe, interpret and share the multitudinous abstract models we construct as developers – domain models, analysis models, specification models, design models, cognitive models, code models (yes, code is still an abstract model – i.e. a description of the machine states a (physical) computer will eventually execute) and so on. Given this unavoidable plethora of abstractions, a suitable domain model, in an appropriate representation, can provide a very important mechanism for anchoring all these abstractions in a concrete reality – i.e. the actual application domain. This anchoring can be achieved with

[1]For many years it has been a concern of mine that computer science and computing undergraduates are not taught about models and modelling in a much more formal fashion, as opposed to the ad hoc, piecemeal approach that usually prevails.
[2]One should note that software is always just one component, albeit a complex and essentially intangible one, of a wider system. Software, as a description, without an underlying computer system, as a means of execution, is just so much useless abstraction; hence the importance to software developers of understanding how to build systems, not just produce software. From now on, when I refer to 'software', it will always be in the context of it being a significant system component of a computer-based system and not an isolated abstraction.

something as simple as a glossary or as complex as a formal mathematical description. A further advantage of domain models is that they can embody important organisational knowledge, particularly in times of rapid change when such knowledge can become disconcertingly transient. Activities such as downsizing, business process re-engineering initiatives and corporate mergers can all be highly destructive of critical, business, organisation and environment-related tacit knowledge. Traditional industry structures, characterised by their organisational and technological longevity have, over the last couple of decades, been giving way to companies with flat management hierarchies and transient staff. In the process, middle managers, so often an organisation's 'gatekeepers', are seen as expendable, so ensuring the loss of, often critical, long-standing, tacit organisational knowledge. This can serve to make effective domain modelling more difficult, but it can also be ameliorated by timely and appropriate domain modelling. Given that many computer systems embed significant amounts of domain knowledge within their internal data models, this is not just a theoretical issue; rather, it is one with very significant practical implications for systems developers.

As I am sure all the readers of this journal are very well aware, writing an acceptable requirements specification is a process fraught with many intrinsic difficulties. A good number of these difficulties can be significantly ameliorated if an adequate domain model exists. I will return shortly to the question of what, in a given context, an 'adequate domain model' actually is. For example, typical problems confronting writers of requirements specifications include dealing with multiple perspectives, stakeholder identification, conflict detection and resolution, priorities, changing context, changing assumptions and the recording of rationale. A suitably detailed pre-existing domain model can make a very significant contribution to dealing constructively with problem areas such as these. Clearly, domain models are not, and cannot be, all the same in terms of the underlying representations and their relative semantic and pragmatic strengths. Two well-known examples, at the opposite ends of what one might call a hard–soft spectrum of domain models, are the use of formal mathematical methods (e.g. set theory and logics) to model the properties of highly structured domains (e.g. railways and airspace) and Checkland's 'rich picture' [2], which serves a crucial role as a domain model in his soft-systems approach to requirements elicitation and modelling.

It is not as if we haven't been told frequently, by authoritative and well-respected figures, about the clear advantages of domain modelling in reducing the risks of failure in software and systems development projects. Over the years, inter alia, Michael Jackson, Dines Bjørner and Peter Checkland have all eloquently and cogently argued, in their unique ways, the case for domain modelling as an essential precursor to successful software and system development. In the future, Bjørner's papers/reports [3] on modelling the domains of railway systems and airline logistics are likely to be recognised as classics in the practical application of formal methods to the characterisation and understanding of safety-critical domains. At what might be regarded as the other end of the systems spectrum, Checkland's soft-systems 'rich picture' constitutes a domain model that readily encapsulates complex social, cultural, political and organisational aspects of the application domain.

In practice, one can probably characterise domain models along two dimensions: the degree of application domain coverage (or conceptual depth) and the strength/weakness of the modelling notation's associated semantics and pragmatics. The notion of some sort of 2×2 classification grid then emerges fairly naturally, if one happens to be interested in that sort of labelling. Despite the fact that the use of the formal mathematical methods and the soft-systems approaches to requirements elicitation and modelling will probably appear to many readers to be poles apart, they are, in fact, a lot closer to each other than any software/system development paradigm that does not make any explicit and constructive use of a domain model.

It is often not clear as to what an appropriate domain model might be in a given development context, and I do not have the space here to explore the issue of domain model choice – I am allowed only enough space here to offer a particular 'viewpoint', but not to provide solutions. In practice, one can start to build useful domain models in a fairly minimalist manner. Terminology is crucial in grounding a domain model. Descriptions and their scope/span, designations, definitions, clear distinctions of mood (e.g. indicative versus optative) [4], glossaries, and so on, are all essential elements of using terminology to identify domain invariants. The use of standard development notations, such as UML, can also be effective – the key is focusing initially on the domain, with a view to understanding it, and not the eventually-to-be-delivered system. Once you understand the perspective I am attempting to present, it will become fairly obvious as how best to describe the particular domain that you are concerned with in a specific application development. In the end, the one really crucial advantage domain modelling brings you is that it provides concrete knowledge, based in the world, that unambiguously grounds the multitudinous abstractions that developers must perforce grapple with as they define and build new systems. To the extent that you can achieve this, then you have a lot of the associated development risks well under control.

About the Author

Professor Offen holds the Chair of Information Technology at Macquarie University and simultaneously fills the roles of Director of the CSIRO–Macquarie University Joint Research Centre for Advanced Systems Engineering (JRCASE) and Honorary Chief Research Scientist in the CSIRO Division of Mathematical and Information Sciences (CMIS). Over many years he has had extensive first-hand experience of most aspects of systems engineering, especially with respect to computing and communications technology and its effective utilisation and commercialisation. His previous positions include a joint professorial post in the Electronic and Electrical Engineering and Computer Science Departments at University College London, Assistant Technical Director of the General Electric Company plc, UK, Operations Manager of Imperial Software Technology Ltd, UK, Manager of the Computer Systems Research Laboratory at GEC Research Ltd, UK and Senior Lecturer in Physics at the University of Otago, New Zealand.

References

1. Jackson M. Software requirements and specifications: a lexicon of practice, principles and prejudices. Addison-Wesley, Reading, MA, 1995
2. Checkland P, Scholes J. Soft systems methodology in action. Wiley, New York, 1999
3. Bjørner D. http://www.it.dtu.dk/~db/
4. Jackson M. The world and the machine. In: Proceedings of the 17th international conference on software engineering (ICSE), Seattle, WA, April 1995, pp 283–292