Requirements Engineering:  
A Perspective Through Theory-Building

M.J. Loomes and S. Jones  
Department of Computer Science,  
University of Hertfordshire, Hatfield AL10 9AB, UK  
M.J.Loomes, S.Jones@herts.ac.uk

Abstract

This paper puts forward the view that the life-cycle model of system development has been allowed to make the transition from a useful intellectual tool for discussing specific aspects of the process to a definitive statement of what the process actually “is”. We argue that this hinders the discussion of Requirements Engineering in fundamental ways, and that challenges to the supremacy of this model are needed to open up effective debate. This is illustrated by the introduction of a model based on the construction of theories, which shifts the emphasis from a technocentric view of the process, where requirements are encoded in initial forms of a system, to one focused on the human understanding of situated systems. Thus requirements become the forces and constraints which influence the designer. We conclude by presenting samples of areas that can be discussed more effectively from this viewpoint.

1. Background and introduction

The design of computer-based solutions to modern problems is a highly complex task. The underlying technology is both sophisticated and complicated, the contexts within which the solutions must be embedded are often ill-defined and transient, and the solutions may not be neutral, in the sense that they may bring about change in the context. One of the major tools that software designers have at their disposal to handle this complexity is the use of models, where abstractions and simplifications are made to enable the designer to conceptualise aspects of the problem in a holistic fashion, omitting reference to details and relationships that are not immediately pertinent to the task in hand. The use of models, however, carries with it a major risk, namely the over-extension of the model so that it becomes used for tasks for which it is not appropriate. Where a model is being used for an obviously technical task, such as establishing dependencies in a class hierarchy, the dangers are well-understood, and most professional Software Engineers are alert to the possible problems and navigate safely around them. We would argue, however, that there is one model that virtually every Software Engineer uses on every project, without ever pausing to consider its validity for the task in hand, the “life-cycle model”. We recognise that there is no such thing as “the life-cycle model”, but we use the term to denote a typical member of the family of such models, the precise details being largely irrelevant for the purposes of this paper. It may be that the need to consider the fitness for purpose of life-cycle models is overlooked because the life cycle is not perceived as a model of some aspect of the system under consideration, but rather as a model of the development process itself, and hence such reflection appears to belong in the meta-level discussions of philosophers and managers, rather than the pragmatic world of the engineer. In this paper, we take the view that the use of models, at any level, potentially has implications for real engineering practice, and hence properly belongs within the engineering debate.

One of the reasons why the life-cycle model is so important at a practical level is that it has served to forge the very vocabulary which engineers use to discuss all aspects of the software design process, including posing problems to be solved, building tools to support the process and motivating research into new areas. Thus many engineers seem to accept that the life-cycle is a given within the problem domain, rather than a model which we may or may not choose to use. As Alexander has noted,

Caught in a net of language of our own invention, we overestimate the language’s impartiality. Each concept, at the time of its invention no more than a concise way of grasping many issues, quickly becomes a precept. We take the step from
and often slides into either vague agreement over discussion of "requirements engineering" is so difficult, and "implementation", and "requirements" have become over-extended, beyond its level of fitness-for-doubt and thus our current paradigm is due for revision. which our current discussions are based are called into such as how would we react if the very models upon ripples within-paradigm, but by asking heretical questions, instructive, to provoke debate occasionally not by creating subscribe to the view that it is constructive, and hopefully intended to provoke thought, not provide solutions. We do propose an additional model, but this is intended to provoke debate, not provide solutions. We subscribe to the view that it is constructive, and hopefully instructive, to provoke debate occasionally not by creating ripples within-paradigm, but by asking heretical questions, such as how would we react if the very models upon which our current discussions are based are called into doubt and thus our current paradigm is due for revision.

2. Life-cycles and technocentricity

As a starting point, let us consider how the term “life-cycle” is used in its original discipline. Macmillan’s Dictionary of Life Sciences provides the following definition of “life cycle”:

The various stages of development through which organisms of a species may pass from the fertilised egg cell of one generation to the same stage in the next generation. There are often various alternatives by which the adult stage is reached, and the term therefore describes the reproductive potential of the species. The term is less commonly used for the various stages through which an individual organism may passed from birth to death [17:page 207]

In both meanings of the term an essential feature is that of some form(s), developing through several stages, and this notion has certainly carried across to Software Engineering. It is not clear, however, which of the two meanings most accurately reflects the intuitive understanding of the typical software engineer when “life cycle” is used to apply to system development. The fact that life-cycle models are usually applied to the development of a single system seems to suggest that it is the “less commonly used” meaning that is being adopted, but the numerous feedback loops present in every form of the model also suggests that there is some notion of many generations being present within any given development. This should alert us to the possible risk that software engineers will carry across some inconsistent ideas, by accepting aspects of both meanings of the term in an ad hoc fashion.

One important feature of the model that is central to this paper is its technocentricity, using Papert’s phrase [21], for it places the developing system firmly at the centre of any discussion. One of the authors has adopted the name “parasitic-embryonic model” in other publications to caricature the life-cycle model - put crudely, the developing system starts from an embryonic form, then develops by sucking in resources from its environment until it becomes a mature system (or resources run out!) [16]. This caricature is well-received by many practising software engineers, who seem to recognise the sense of losing control that comes from such a technocentric approach.

The Software Engineer within this model is primarily a carer, charged with the task of ensuring the system develops properly through its various stages. At the outset, the embryonic form must somehow encode the essential information required for the development to take
place, so that the “right sort of system” is produced. That is, we need some software equivalent of DNA present from the outset to constrain and control the growth. Assuming that the embryonic form encodes its essential nature, any deviations from what is expected is likely to be blamed on the carer, whose only defence is to blame the initial encoding, the “requirements”. Unfortunately, the life-cycle model pre-supposes the existence of such an entity (typified by a requirements specification or statement of user requirements), and does not offer any real assistance in understanding or discussing how this initial encoding is carried out. It is significant that we have chosen to label mistakes that we make “bugs”, so that they too can have a technocentric existence, and we can set out to “find” them. So both the desired properties of our system, and things we should avoid, are given an ethereal existence.

With this current view of software development, many of the difficult challenges for Software Engineering are obscured by being posed within a model which is not really adequate for the purpose. Attempts made to discuss these challenges within-paradigm can lead to a complex web of linguistic problems, as we struggle to interpret terms which are essentially outside of the model in ways which are largely defined by the model itself. For example, if we seek to use multiple loops within a life-cycle to allow the engineer to revisit any stage of the life-cycle at any time, we completely destroy the notion of well-defined intermediate forms and specific well-understood transformations between them, yet we still seek to preserve the notion that a single entity somehow sits at the centre of the process. We seem to have managed to conflate the two meanings given for “life-cycle” above, so that we focus attention on the development of an individual organism (or system), but the existence of multiple feedback loops in our models suggest that we want our system to be able to change the circumstances of its own creation. We seem undecided whether we are discussing the evolution of a species of systems or the development of a single system! Moreover, we still want to preserve aspects of the technical meanings of terms such as “specification” and “design”, even though these meanings may be inconsistent with such a complex version of a life-cycle. We can, of course, fall back to common meanings for terms, but then we run the risk that semantic shifts will subsequently take place, insidiously pulling the interpretation back into technical usage in an ad hoc fashion.

When the life-cycle analogy was first made in the mid 60s, software developers were largely involved in solving problems which were well-understood, using technology which was new and challenging. They really were in a role of taking some encoding of a solution, typically solving a computationally intensive mathematical problem or automating an existing paper-based information system, and nurturing it so that it emerged in a fully-fledged executable form. In a survey of the first ten years of commercial computing, for example, Kilner notes (in 1960) that “computers are only just being envisaged in the role of corner-stone of the work in which they are employed” [8:page 89]. It is against this background that the life-cycle analogy was developed. By the early 80s, the much broader role of computers was being appreciated, but by then the life-cycle was well established in a dominant role. Lehman, for example, in his important work on the classification of different types of program [11], raises many of the issues that are still of paramount importance in Requirements Engineering today, but expresses these within the context of life-cycles at several levels, even though this means taking some rather unusual interpretations of terms. He also has a significant sentence that starts “One of the main concerns of life-cycle process methodology research must be to develop ...”, which seems to suggest that life-cycles have made the shift from “description to criterion” warned against by Alexander above: no longer are we using them if appropriate for our own ends and under our professional control, they now have a separate existence that needs researching. One implication of this is that, by making our process models part of the technology rather than recognising them as meta-level artefacts, we can make our study of process itself technocentric, the irony being that as long as we don’t challenge life-cycles in any fundamental way we can research them endlessly as a valid part of the discipline. As long as we protect them from difficult challenges that may render them inconsistent, they will remain internally consistent!

3. Theory building

This paper proposes an additional model of the development process, which is not technocentric, but rather focuses attention on the emerging knowledge relevant to situated problem solving as central to the enterprise. This model has been called the “theory-building view”, and is described in more detail in previous publications [12][14]. The motivation behind this model came from seeking to explore three areas, each of which seemed obscured rather than clarified by existing terminology and models.

- Are there ways of modelling the software development process that allow us to discuss the claims being made during the 80s by both the “formalists” and the “soft systems” people within a common framework? Acceptance of life-cycles seem to force these two views apart, although it is difficult to establish any particular areas of disagreement within the model...
itself. Burstall and Goguen, in 1977, discussed the idea of specifications as theories, in a paper that many would consider seminal for “formalists” [4]. In 1985, Naur stressed the importance of human-held theories in design in a paper that many took as a call to adopt a non-formal approach. He claimed that “an essential part of any program, the theory of it, is something that could not conceivably be expressed, but is inexorably bound to human beings” [20: page 258]. The common use of the term “theory” was seen as a possible way of reconciling these two views, but as theories exist outside of the technocentric circles of the life-cycle, an alternative model was required to carry out this exploration.

• What is meant by statements frequently made in the 80s that software development should become more “scientific”? These statements were usually defended in very vague ways, often seeming to use “scientific” as a synonym for “better”. Hoare’s famous paper, “Programming: Sorcery or Science” [6], for example, manages to debate the issue without using the word “science” in the text at all, but seems to rest on the notion that we should have ways of nurturing the system through its development so that, assuming the properties are encoded properly at the outset, we are guaranteed correctness. This seems at odds with most modern views of science, however, where scientists accept that their theories must be refutable. As soon as we start to explore the question of what it means to be “scientific”, the issue of theories rapidly becomes central once more.

• How do we capture the notion that software development is about solving situated problems [5]? The life-cycle focuses attention on the transition from “what” we are trying to build (the “specification”) to the “how” it is to be constructed (the “implementation”), but does not seem to admit the discussion of “why” our ways of proceeding are actually sensible for solving the problem (the “explication”) [13]. Ryle has discussed such issues in his book [23], where he draws the distinction between “knowing how” and “knowing that”, and once again the existence of theories at higher levels than simple system description is needed for such a discussion.

There is no attempt in this short paper to provide details of what we mean by the theory-building model, or to pin down our notions of theories. These have been discussed elsewhere, and no doubt variations on these themes could give rise to the same sort of multiplicity of variants as there are for the life-cycle models. The essence of the model, however, is that what “develops” is not the system per se, but the understanding of the issues required to solve some problem. A technical implementation of the solution is simply one of the many side-effects of this development that happens en route (alongside documentation, arguments for correctness and other system properties, training programmes, changes to working practices, ideas for new projects, and many more things that frequently arise from modern system developments). For the purposes of this paper, an intuitive understanding of what constitutes a “theory” will suffice (although many of the fundamental questions of Software Engineering surface when you start to refine these further). It is important, however, to stress two points. First we may need to use various representations of theories at different times in our discussions. We might want to consider completely formal theory presentations as used in algebraic specifications, for example, when discussing the role of formal methods, or we might want to consider how knowledge elicitation can be used to externalise users’ theories of existing systems. We would argue that this is a strength of the model, not a weakness, as it allows currently disparate activities to be discussed in a common framework. Second, we must stress that our model must be kept to the status of an intellectual tool, which we find helpful when discussing systems development. There is no suggestion that this model should replace the life-cycle, as that would simply lead to over-extension of a different model and cause the next generation of problems!

Discussing Software Engineering in the light of this model admits many areas that cannot be addressed in life-cycle models, except as issues relating to the nurturing environment in a fairly marginal fashion. For example, the importance of prior knowledge, the need for learning organisations, and the central role that communication between those involved in the development process plays can now all be made explicit in the model. Recognition that designers change in the course of a project as they gain a deeper understanding of the task they are engaged in, and the implications this will have for the artefacts being produced at the end, can now be discussed. The issues surrounding the task of making intuitive theories explicit and possibly formalising them can be put onto a firm basis, for example, and we can recognise that “formal specifications” and “intuitions” need not be seen as disparate things, forcing us to choose between rival camps, but can be viewed as facets of some deeper theory, both of which have their roles in Software Engineering, and the relationship between them can be explored explicitly [15]. We can also recognise that designers make mistakes, at various levels, and therefore discuss the source of errors rather than positing the existence of “bugs” as some strange interlopers in our technocentric world occupied by the embryonic system to be “found” post hoc.
4. Requirements engineering

In this section we will outline a few of the areas where the theory-building view can be used to structure and inform discussion of Requirements Engineering. In particular, we will attempt to flag those areas where current issues of debate, motivated by direct observation of the problems facing designers of complex systems, can be brought into contact with existing bodies of knowledge relating to theory building. Space does not permit a proper review of these issues, however, so this section is presented as a sampler of what sorts of areas open up if you start to consider requirements engineering from this different perspective.

4.1. Philosophical foundations

First, at a very general level, let us observe that there is currently very little literature on the philosophy of technology, and virtually none that addresses Software Engineering explicitly. The software engineer who wishes to consider some of the deeper issues relating to the design process has to work *ab initio*, which for such a large and complex domain is a rather daunting task for the philosophical novice. If we adopt a theory-building view, however, we have at our disposal the vast literature on the Philosophy of Science. This is sometimes dismissed by engineers because they believe that “science” deals with general laws in some natural world, rather than man-made solutions to particular problems. In modern accounts of science, however, there is considerable debate over the extent to which scientists are actually dealing with the natural world, rather than the synthetic knowledge structures built by other scientists, thus many aspects of the Philosophy of Science deal with the development of complex information and knowledge systems, rather than “nature” in some primitive sense. Moreover, there is also considerable debate over the issue of generalisation. Whilst classical physics has usually been presented primarily as the quest for general laws, disciplines such as modern biology focus attention on the ways individual organisms and species develop and evolve through time, and many of the debates surrounding these issues mirror those relating to Software Engineering. Mayr, for example, questions the classical notions of cause and effect as represented in traditional physics, and introduces the concepts of “proximate” and “ultimate” causation to explore the relationships between “how” and “why” questions in developing systems [18: pages 24-37]. Many of the points that he makes seem highly pertinent to discussions in the requirements engineering community relating to the task of explaining “how” information systems might be built within the context of complex, developing, enterprises, and “why” they will contribute to the well-being of the enterprise. It would be an interesting research area, for example, to take some of the work currently being carried out within the Swedish Institute for System Development into enterprise modelling [2], and to relate this to Mayr’s work on teleological systems [18: pages 38-66].

4.2. Requirements as active forces on the designer

The term “requirements” itself can be interpreted in a richer sense using the theory-building view, such that it refers not to encoded properties (the DNA) of the parasitic-embryo at the centre of some technical process, but to factors which influence the construction of theories by the designer. This represents a significant shift of emphasis from product to process, with “requirements” becoming explicit driving forces behind the actions of the designer specifying what he or she must do, rather than an encoding of what some as yet non-existent system must do. The theories which the designer constructs must still, of course, be fit for the purpose of solving the problem, but we can recognise that, as well as the system’s properties, the context within which the problem is posed will determine many of the properties we expect of our theories, and so will also influence the design methodology which surrounds the development process.

4.3. Requirements: gathered, sought or inherited?

A naive view of requirements gathering, for example, held by many students is that the process starts with a set of pre-existing properties which our system should have, and the designer’s task is to “gather these in”, and then produce a system which is consistent with these. Popper refers to this as the “bucket theory of knowledge” and points out that the gathering and interpretation of facts in this way is rarely sufficient as the starting point for theory construction, as we need to make well-informed judgements about the status and meaning of the facts, which requires some pre-existing theoretical understanding. Rather, he suggests, we should adopt a “searchlight theory of knowledge” [22:pages 341-361], where we actively seek out those facts that would enable us to advance our knowledge. In general, these will be the facts that may challenge our existing theories, rather than those that simply increase our confidence in our existing theories. This means we have to have ways of evaluating factual information, and also hypothesising the “existence” of additional facts we should seek. A less naive strategy
would probably use both approaches, a bucket of facts to bootstrap the process, providing a basis for the first inductive step to some initial theory, followed by deductions from a current theory to predict properties that might not hold true, to permit a search for refutations [7]. In practice very few designers actively seek out ways of refuting their design; perhaps bringing about this change of attitude is what we should mean by Software Engineering becoming more scientific.

For most developments, the designers carry theories with them even before any real details of the task in hand are apparent. These may be labelled prejudices (“anything we build for marketing is bound to be problematic”), valued as experience (“every system can be viewed as a scheduler of resources”), based on familiar tools (“there will be data flowing between entities”), embedded in company practice or culture (“it doesn’t matter if we don’t manage to cover all the facts with our theory, because there will be another version along within two months”), or come from many other sources. These theories all serve to influence the ways we behave, as well as providing additional facts to be incorporated. This is important, because it suggests that the traditional maxim “specify what not how” is not as simple as it seems. Knowing that we are building a safety-critical system within a particular organisation, for example, may well signify that we should not use recursive programming techniques. If we “know” this is an implication, are we specifying “how” when we add the phrase “safety-critical” to the set of facts we are amassing? Moreover, should we somehow make this deduction explicit at this stage, so as to educate any new designers that might not be aware of this bit of the theory yet? If a client wants a system implemented using a standard protocol, do we have to pretend that we do not know how this is implemented, or if a particular development method is prescribed do we have to pretend that we could reach non-procedural implementations even though we know this is impossible? In the life-cycle model we can avoid these issues, because we are only interested in the development of the embryonic form, so we can store away things that we need to use later, indeed, we are not able to acknowledge their existence even if we want to. In avoiding them, however, we also avoid all discussion of how to improve working practices in the area. Bubenko and Kirikova note, for example, that “acquisition and analysis of requirements for information systems is a strongly iterative and creative design activity. This activity of the information systems life-cycle is often neglected. Insufficient time and resources are allocated to it” [3]. We would argue that this is an inevitable consequence of the use of the life-cycle model, rather than a value judgement being made by researchers or engineers.

4.4. Do “requirements” really change?

Another area which we can tackle head on with the theory-building view is that of changes in “requirements” during the development, although this is an example of an area where we really lack suitable terminology to carry out sensible discussions. This notion may cover a multitude of scenarios, all represented in the life-cycle model as some change to the DNA determining the embryo's properties, via feedback loops, seemingly winding back time to allow the embryonic form another chance. Perhaps the most likely cause of changes are when existing theories regarding the system or the problem situation change, so that different deductions are made or different questions are asked leading to a refined view. If we leave aside the problems of malicious mis-information, and slips leading to genuine errors, any inconsistencies that arise during the design process point to a problem in existing theories. If two users, for example, give inconsistent answers to a question it may be that we have failed to recognise that the question has different significance to each of them, or that our current theory confuses two concepts by overloading a term. A classic example of this was provided during discussions at the ISTIP89 conference, when one of the participants recounted the tale of designing a system to keep track of buildings owned by a large organisation. He was unable to find out how many buildings were currently on the books, and was given wildly differing estimates by several employees. He eventually discovered that the concept of building, which he was treating in an intuitive fashion, meant very different things to those in the architects office, those providing services such as heating, those responsible for security, and those using the rooms. Even “obvious” facts like ‘there must be fewer buildings than rooms’ turned out not to be true, where several extensions had been built over a period of many years, then subsequently knocked into one room. The concepts of “building” and “room” as bound into the designer’s initial theory were not the same as those in various stakeholders’ theories, even though the words were the same. Any “specification” that was written at the outset using these terms would have meant something different at various stages of the design process, as the designer learnt more about the domain in which he was working, and the stakeholders developed a broader appreciation of each others’ points of view. Even if the words remained unchanged, the meanings they took would have altered, and perhaps words would have been interpreted differently in different contexts. Context-dependent meanings are rarely addressed in current literature on requirements specification.
4.5. Stakeholders and theory-building communities

This leads us on to consider the ways in which different stakeholders are viewed, and view themselves, as participating in the theory-building process. For example, we might consider “users” as empirical objects, experimented on by “designers” and having their knowledge elicited as contributions to some pool of facts being sought by the designer’s searchlight. Alternatively, we might think of users as patients requiring therapy, whose theories of the world need adjusting to fit in with the designer’s view that is about to be imposed or, slightly more benignly, as pupils who have to be taught what is “right” during a training course prior to the system being used. We might take a rather more radical approach such as those proposed, for example, in the Tavistock Institute in the UK, where system design is seen primarily as an ethical challenge and as a route to enabling “users” to exert more influence over the systems that surround them [19]. In this case we are likely to import a number of ethical theories into our development, although for political reasons we might not wish to acknowledge them explicitly.

Once again we can appeal to the Philosophy of Science for insights into the ways in which theories are held and used not only by individuals, but by (potentially heterogeneous) communities. Following lines suggested by Kuhn [9], for example, we might start to search for “paradigms” within which developments are carried out, where aspects of theories are accepted without question, embedded in areas such as ethical values held culturally or professional standards expected by an industrial sector. Developing these ideas further, we might see if we can identify “hard cores of research programmes” which are carried across as accepted bodies of theory between projects, only to be challenged if all else fails [10]. If we adopt these ideas then perhaps we should be recognising that “requirements” in the wider sense are captured not only within specific projects, but also carry across between projects, embedded in cultures, organisations and communities, and that we need to find ways of valuing and recognising this in our approaches to Requirements Engineering and managing projects.

4.6. Learning as a component of design

Another area that is exposed for scrutiny from the theory-building perspective is the importance of learning in the design process, and how requirements engineering should respond to the challenges that this poses. When a development uses machine learning, such as in neural networks or rule induction, it is seen as out of the ordinary, and the life-cycle really has to be distorted in extreme ways to accommodate the processes. When a designer has to learn something in the course of a development, however, the learning is completely ignored, which means it is not managed, validated or controlled in any explicit fashion. Consider, for example, the following pair of problems, which have surface similarity.

1. A rectangle has non-zero integer sides. Whole unit-squares are drawn in the rectangle so as to fill it. Find the area of the rectangle, where area is defined as the number of squares in the rectangle.

2. A rectangle has non-zero integer sides. Whole unit-squares are drawn in the rectangle so as to fill it. A diagonal is drawn across the rectangle. Find the Loomian coefficient of the rectangle, where this is defined as the number of squares cut by the diagonal.

Problem one would rate as trivial, and is a typical task set in older programming books as an early exercise. The “requirements” are fully understood, that is, the designer understands “what” is required in a way that has real meaning, both denotationally (the answer is “length times breadth”) and operationally (“read in the length of both sides and multiply them together”). Problem two is rather different, although at one level it is as simple to understand as the first problem. The terms used are not grounded in theory in the same way. The average software engineer would not have a denotational or operational understanding of the problem, or even know if such a coefficient can be found as a formula. He or she would not be able to check any solution that was offered easily, as there is no general way of knowing whether the answers produced are correct or not. The engineer needs to learn the necessary theory to support the specification (what), in a grounded fashion, the implementation (how) through an operational understanding, and also the explication (why) to show that it works. Thus a requirement of this development is that the engineer learns how to solve this problem, even before the system can be discussed in any meaningful way. We would argue that this is a property of the vast majority of complex system developments currently being undertaken, but that our technocentric viewpoint prevents the implications for the development of human resources from being properly acknowledged.

5. Conclusions

In this paper we have attempted to provoke the reader into stepping outside of the life-cycle framework and considering alternative views of the software system development process that may give new insights to the problems of requirements engineering. It may be that, as a result of this, the reader is confirmed in the opinion that
life-cycles are adequate for the task of discussing these problems, and that new models are not necessary at this time. We would argue that the conscious and deliberate use of life-cycles for these discussions is perfectly acceptable, as the software engineer takes responsibility for controlling the use of the model and understands the limitations implicit in this. Our objection is to the implicit use of the life-cycle model, and the terminology which forms it, in an unquestioning fashion, as if it were a given in the domain and not within the control of the professional engineer.

In order to focus the discussion, we have introduced an alternative model, motivated by the observation that the term “theory”, with related but differing interpretations, is used in a wide variety of areas within Software Engineering. We have not been able to develop the theory-building model in any detail here, but hopefully we have been able to wet the appetite of the reader and set off trains of thought that reach outside those initiated within the life-cycle. It is certainly our experience that this is the case, and there are sufficient examples of theory-building being used elsewhere to give us confidence in this view [4][7][20]. There is, however, a major challenge in bringing together these currently disparate and isolated examples into a coherent framework, and showing that the theory-building view has a role as a general model in the discussion of Requirements Engineering.

References