Competence-based Requirements Engineering

Grady H. Campbell, Jr.
domain-specific.com
Annandale, VA, USA

Abstract*—The potential exists to streamline the initial requirements engineering effort involved in building a software product. Most software today is, in many respects, highly precedented, differing only with respect to certain particular capabilities. Recognizing that an envisioned product is similar to previously developed products enables an inverted, more aggressive approach to defining requirements. Three elements contribute to this streamlining: (1) a reformulation of the development process as an opportunistically-ordered, information-synchronized, iterative coalition of concurrent activities, (2) a presumption of developer competence (knowledge, expertise, and experience) in the problem-solution space of the relevant application domain, and, (3) a conception of requirements as being a model of an envisioned product's expected observable behavior. The requirements takes a bipartite form consisting of an under-constrained customer expression of product behavior and an over-constrained developer expression of product behavior which must be kept mutually consistent. Based on the product line premise that similar problems are amenable to similar solutions (in this case, the need for and expression of similar behavior), this approach can expedite attaining an initial expression that approximates the envisioned product's requirements, entailing less effort on precedented elements while giving more attention to less well understood elements. The concurrent process and bipartite form of the requirements also enable greater flexibility for accommodating changing requirements, as customer needs evolve during development and over the subsequent lifetime of the product.

***Index Terms*—requirements, domain-specific, competence, similarity, uncertainty, change, model, process.**

I. Introduction

In every respect, software engineering is an immature discipline. Properly balancing cost-schedule, functionality, and quality is relatively unpredictable, being highly dependent on customer and developer organizational circumstances and the *competence* (knowledge, expertise, and experience) of individual participants.

The concept and practice of requirements is exhibit one. The requirements for a software product is seldom well-understood or properly communicated before, during, or after development. Documentation that may exist is typically out of date or otherwise inaccurate. The implication of this is that an understanding of the intended software behavior of a product is unduly dependent on access to a small number of people or inferences from the details of complex software code without insight into determinant tradeoffs or rationale.

This paper suggests an alternate vision for the practice of software engineering, specifically the treatment of requirements in a disciplined product development effort.

II. The Vision

The proposed vision has three elements:

- Developers possessing competence in the relevant problem-solution domain
- An information-driven concurrent development process
- A conception of requirements as a model of an envisioned product's expected behavior

Any organization that builds software has particular areas of competence, consisting of the knowledge, expertise, and experiences of its developers and expressed in the products that it has previously built. These areas of competence include both technology (solution) areas and subject matter (problem) areas. An effective product results from a proper problem-solution alignment. An organization that has existing competence in specific areas will be successful more quickly in building a product that embodies those areas than in building a product for which they must first acquire needed competence.

A traditional software development view of requirements engineering is that the developer will have a general software engineering competence that will suffice for a solution but that problem domain competence that frames the solution must come from the customer. The developer acquires domain and problem understanding through an elicitation-analysis process with the customer and then builds a solution based on that information. However, experience tells us that a developer having existing problem-solution competence, including awareness of domain-specific terminology, tacit knowledge, and past solutions, will more quickly understand the needs of the customer's business and what to build based on similarities to previously-known problem-solutions. A developer with such competence is necessarily better prepared to more rapidly build an acceptable product than one lacking such competence.

The activities for developing a software-based product are best not rigidly ordered – the order should be driven by the availability and needs for coherent capture of relevant information. Such a process mirrors the natural way that an individual working alone would

build a product. Each activity is distinguished by separation of concerns to be uniquely responsible for information related to specific aspects of a product; the information associated with each activity constitutes a distinct partial model of the envisioned product. Nominally, the overarching activities of this process correspond to models of the requirements, design, implementation, verification, and delivery of the product. The proper time for working on an activity should be determined opportunistically by the availability of information that it depends upon or the need for information that it produces.

Requirements in particular is a model of the expected observable behavior of the envisioned product. This model has a bipartite realization:

- (outer requirements) The capabilities that the customer expects of the product so as to effect needed changes in how their business operates
- (inner requirements) The behavior that a product must exhibit to satisfy its intended purpose

Outer requirements are meant to be a customer view of the requirements, a communication between the customer (representing users) and the developer that defines what capabilities the customer expects the product to give them. Ideally, outer requirements should be under-constrained so as to not impose non-essential criteria on the solution, leaving the developer adequate leeway to resolve tradeoffs among functionality, cost-schedule, and quality. It represents the customer's perceived needs or preferences and, as such, is an expression of the criteria against which the customer will validate and determine acceptability of the product.

Inner requirements are a developer view of the requirements, a communication among the development team that elaborates upon the outer requirements. It should be over-constrained such that it eliminates all essential uncertainties concerning the expected observable behavior of the product. This expression can be freely changed based on feedback among development activities as long as it is kept consistent with the outer requirements (including any changes negotiated with the customer during development). Inner requirements constitutes a build-to specification of the envisioned product, is an expression of verification criteria for the product during development, and, upon delivery, constitutes an as-built specification of the product.

Motivating Assumptions

For various reasons, software developers have tended to hold some unrealistic notions about the nature of requirements, such as that customers should "know what they want" and that this should not change during development. Somewhat different assumptions would engender more realistic expectations, leading to more effective software development practices:

- Requirements is simply a model of the expected behavior of an envisioned product. Being a model, it does not describe a specific product; it could be satisfied by many differing but model-equivalent products.
- The perceived nature of the product will change through its development even if the requirements never change. A model inherently omits essential and incidental problem-solution knowledge (e.g., tacit problem knowledge, engineering alternatives and tradeoffs for a solution); some implications of this will not be known until the product has been concretely realized and experienced in a facsimile of its operational context.
- Uncertainty and change are a normal, inevitable and unavoidable, aspect of building a product that will meet actual needs. Actual needs are likely initially to be poorly understood and poorly communicated. The actual needs will also change over time due to changing business circumstances and tend to also change due to the injection of the product itself into the customer's business process. A product that meets poorly understood "requirements" will be a poor fit to actual customer needs.

III. INFLUENCES

The proposed vision and approach is a distillation of 30 years of software engineering explorations by many different people. These explorations have focused on both theory and practice in software processes, requirements methods, and software product line methodology.

Jackson argued that the future of effective software development lies in specialized (domain-specific) knowledge in the same sense that all other branches of engineering are specializations [2]. An underlying justification for this view is that explicit communications about requirements are based on assumptions, arising from tacit knowledge, that are shared among domain experts. For software developers, the concern is understanding the aspects of the problem in context, its processes and tradeoffs, that affect the solution.

The significance of tacit knowledge, about both problems and their solutions, is broadly recognized. A product based only on explicitly communicated information from a customer without awareness of underlying tacit knowledge and assumptions will be a fragile solution. Similarly, as a model of the product, requirements can only approximately represent the product as a whole, and is necessarily dependent on being consistency with tacit knowledge.

Faulk provides a good characterization of a sound software requirements discipline, including problem analysis and specification [10]. However, that

conception can be streamlined if developers have competence in the domain of the envisioned product, including problem-solution knowledge of previously developed similar products.

Approaches that emerged in the early 1990's were conceived to focus development organizations on leveraging their competencies and institutionalizing domain problem-solution knowledge as the basis for more quickly building high-quality solutions (e.g., experience factory [3] and software product lines [4]). These approaches were based on the belief that effective development organizations tend to have greater expertise in the understanding of problems and development of solutions for products in particular application domains. The most successful organizations are ones that are able to align their business objectives to match their expertise and to enhance their expertise to the benefit of their business objectives.

Recent work has accepted the benefits of developer domain competence but has advocated participation by domain-ignorant engineers in requirements elicitation to better expose overlooked tacit assumptions [7]. This is a sound observation but the reality in practice is that domain competence remains too scarce among developers. Even more so, traditional development practices are formulated with the assumption that development will start with a blank slate that is collaboratively elaborated into a product based on the domain knowledge of the customer and the general software competence of the developer. A process that assumes developer domain competence has the potential to be more effective.

IV. The Role of Competence in Software Development

The base level of software competence is the ability to build software in general, to build for use in a particular computational environment, and to perform the full scope of activities entailed in creating a complete software product. As with any complex activity, the totality of this competence takes a good many years to learn, involving knowledge of both theory and practice. Today, this learning is augmented by the existence and use of various libraries of generally useful software components.

With only this general level of competence, people have built many complex systems. Doing this has involved large amounts of effort to learn enough about the problem domain of a particular customer's business to understand what software needs to be built. In doing this without prior knowledge and expertise, there are inevitably many misconceptions, missteps, and misunderstandings. As a result, software developers have come to recognize that continuous rapid iteration through a well-conceived series of activities (notionally, requirements, design, coding, testing, and installation) is necessary to discover and correct for these shortcomings.

It is known empirically, however, that a developer who has built multiple products of the same general type will need to spend much less time learning about the problem domain to build another product of the same type. There is still the need to determine how the specific needs of a particular customer fit within the problem domain but this is much less when the developer is already familiar with other similar problems and solutions.

V. Requirements as a Model of Behavior

The concept of a software "requirement" in practice is a somewhat vague and confusing term. As a general concept, we can think of software requirements as an expression of the criteria that a product must meet to satisfy its intended purpose. The premise of this paper is that we can improve the predictability of software development, improve the quality of the software, and reduce its cost if we correct the confusion that surrounds requirements.

It is not sufficient to conceive of requirements as being a simple list, of "shall" statements or "features", that can be ticked off, ignoring interactions and dependencies and mixing essential and incidental aspects. Instead, requirements is an abstract expression of a coherent whole, describing the expected observable behavior of an envisioned product, and corresponding to some one of a set of similar products. It is resolved to a particular product through engineering judgement based on a systematic exploration of alternatives and tradeoffs. A product is an elaboration of requirements augmented by tacit domain and engineering knowledge.

So, *requirements* as a model describes the observable behavior expected of a referenced product. A *model*, in turn, is a representation of a product that is sufficient to provide approximate answers to a designated set of questions about that product. Requirements, being a model, should provide answers to four categories of question:

- Concept: What is the purpose, objectives, and use to be made of the referenced product?
- Context: What is the nature and composition of the system environment(s), including user roles, connected systems, and devices, into which the product is to be injected so as to induce modified capabilities and behavior in that system?
- Content: What is the observable behavior, including functionality and qualitative/ quantitative properties, to be exhibited by the product in its operational context?
- Constraints: What are any externally imposed limitations, including legal, regulatory, industry, or business considerations, on the construction or composition of the product?

There are questions about the product that requirements should not answer. For example, it should not express the internal structure of the product, the nature of its constituent elements, how the product is verified as being properly built, or how much effort would be required to modify the product's behavior. Other models of the product should answer these and other such questions.

Expressing Bipartite Requirements

The bipartite realization of requirements (outer and inner views, as described above) reflects the differing concerns of customer and developer. The outer view is concerned with ensuring that the product will fit into and support operation of an envisioned new or improved business practice, providing capabilities that users will need to perform their work. The inner view is concerned with establishing a coherent definition of the precise behavior to be expected of the product, providing a common and consistent basis that all activities can reference as authoritative. These two views establish, respectively, the lower and upper boundaries on what constitutes acceptable behavior, with the space between representing the flexibility that the developer has to make needed economic and engineering tradeoffs.

Outer requirements that are over-constraining, specifying acceptance criteria that are incidental, not essential to the product's intended purpose, can unnecessarily inhibit the developer from considering what could be better alternatives. Conversely, inner requirements should be over-constraining, resolving (even arbitrarily) any uncertainties to preclude their being resolved inconsistently across activities.

Outer requirements can be determined only in consultation with the customer and subsequently changed only to correct for misunderstandings, uncertainties, or changing business circumstances. Changes to outer requirements will propagate as changes to inner requirements. In specific cases, development tradeoffs may warrant negotiating with the customer to change outer requirements (e.g., to avoid undesirable effects on cost, schedule, or product quality).

For a customer, requirements is an abstract characterization of their needs whose implications will not be entirely clear until the customer is able to experiment with trial versions of the product. Such trials may expose misconceptions in the outer requirements, failures to account for tacit knowledge, or new insights into how the product can enable improvements in business practices. Such insights will often lead to revisions in the requirements, leading to delivery of a better final product.

The inner requirements needs to give developers having appropriate solution-space competence enough problem-specific information to enable them to build a suitable product. Inner requirements must be kept consistent with outer requirements but can otherwise be changed freely based on feedback about tradeoffs among development activities.

The inner requirements, as a build-to specification of expected product behavior, is the basis for ongoing product verification and, upon delivery of the product, becomes an as-built specification of the product's expected observable behavior. The outer requirements should be the agreed basis for product validation and customer acceptance.

VI. A Concurrent Software Process

A software process is a partially ordered set of activities for the development and evolution of a software product. Each activity is concerned with particular related portions of information about a problem and its solution in a customer context.

The tradition of organizing development into phases has had the effect of imposing a mentality of false sequentiality on the process, that activities must be completed in a fixed order. This has been countered with a discipline of repeated iteration over activities so that feedback from insights gained in subsequent activities and changes in understanding of needs could be better accommodated.

If we consider how an individual developer would naturally approach this, we can see the potential for a more naturally opportunistic ordering of effort based on information dependencies. Each activity is associated with a model of the product that expresses particular aspects that are relevant to the purpose of the activity.

In this view, all activities, including requirements, can occur concurrently with associated information flowing through the product to be shared among all activities as needed (Fig. 1). In this context, "product" consists not only of materials that are to be delivered to the customer but all information referenced or created during the development process, particularly information useful in future evolution of the product such as reference materials, analyses of alternatives and tradeoffs, and rationale for alternatives chosen.

All of the models associated with activities in aggregate comprise the product. In a natural (what an individual would do naturally) and effective process,

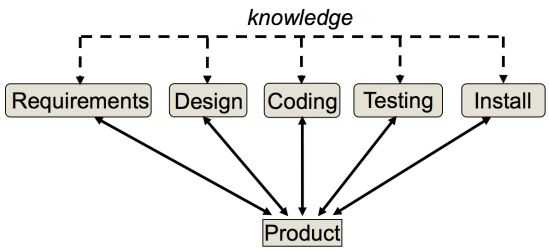

Figure 1. A Concurrent Process

the developer will move freely among the various activities depending on where particular information is best captured or produced. Each action tends to trigger propagation of information (forward or back) to other activities as required to maintain product consistency. While a single developer would have to simulate concurrent activity by interrupting one activity to work on another, a developer team can be working simultaneously on different activities or iterations.

This sort of iteration among activities requires a strong discipline of baselining and version management but is more likely to result in a product that satisfies actual changing customer needs. This discipline also fosters the expectation that the product should rapidly attain a state of partial completeness sufficient for being demonstrated or even delivered on demand rather than according to an artificially prescribed schedule. With continuous verification to ensure consistency across the product, this also provides more flexibility to engage the customer in making informed tradeoffs for the best balance among cost-schedule, functionality, and quality.

The ordering of the activities of a concurrent software process is neither fixed nor predetermined; it is derivative of actual informational dependencies determined in performance of the activities. Activities are characterized by the product information they consume and/or produce. An activity can be performed anytime after information that it needs becomes available (and/or when information it produces is needed by other activities). As with any iteratively performed process, each activity is performed repeatedly, both to address different subsets of the type of information that it references (e.g., different code components or test sets) and to iteratively extend, refine, or revise previously considered information (e.g., changes in the product's design due to changes in requirements or vice versa).

The means to control the effort required to reach a state of completeness with this approach comes not from the fitting of activities into an arbitrary schedule but through the imposition of bounds on the requirements model from which appropriate progress milestones can then be inferred. A product is considered conformant to the requirements if it satisfies any consistent subset of the behavior expressed in the requirements model. Failure to meet any milestone would be addressed by a further subsetting of the requirements model to establish an achievable target within an acceptable timeframe.

The notion that the requirements activity should occur prior to other activities is somewhat true in the sense that it should be the definitive source as to what needs to be built but this does not mean that other activities must wait for requirements. Similarly, no activity, including requirements, is finished until the product as a whole is complete.

The process can start with any (one or more) of the activities, based on what information is available about the envisioned product and the competence possessed by available developers. For example, an initial increment of the process could consist of a first iteration of the requirements activity to determine initial outer requirements, the design activity to create a preliminary architecture based on past similar solutions, the coding activity to create or obtain low-level components that are likely to be useful, and/or the testing activity to start creating a suitable testing infrastructure. Information used and produced in iterations of each activity must be version-managed for alignment with iterations of related activities. The aggregate results of a set of related activity iterations comprise an interim version of the product. Any interim version of the product can be delivered for use if it can be determined by the customer to be acceptable.

Requirements for a product as initially conceived informs other activities but must be revised as circumstances and understanding of customer needs change. The implications of such revisions must then be accommodated in other activities. Similarly, insights gained in performing other activities, including resolution of uncertainties and exploring alternatives, can lead to changes in the requirements that must then be reconciled to customer needs. This sort of iteration among activities, with strong baselining and version management, is an effective way to avoid building products that fail to meet current needs when delivered. Each instance of an activity is also performed iteratively but internal iterations need not be synchronized across activities (e.g., requirements can progress through several iterations while other activities reference only a preceding baselined version).

As activities proceed and constituent product information is acquired or created, other activities become feasible to perform using that information. Natural information dependencies are directional and define information flow among activities. Every information path between activities has a reverse path for feedback to reflect insights gained in the use of information shared. For example, the requirements expresses information about expected behavior that influences design, coding, and testing activities. Performance of those activities will expose issues or tradeoffs that will need to be resolved in a subsequent iteration of the requirements activity, followed by additional iterations of those dependent activities.

VII. Inverting Requirements Elicitation

In the software process, the purpose of the requirements activity is to determine the behavior (functionality and properties) that the customer needs the product to exhibit and to express that in a bipartite specification of the requirements.

Requirements, being a model of the software product, does not determine a single specific product but rather expresses behavior that many conceivable products would satisfy. Furthermore, this model is not static, it expresses perceptions that are susceptible to uncertainty, misinformation, and changing circumstances. These issues are made worse by a desire for certainty in customer needs as fully known, understood, and fixed but such certainty would be an illusion [6]. The necessary response is a process that views uncertainty and change as unavoidable and is organized to anticipate and accommodate changes to the product during development.

Traditionally, the requirements activity for a custom product starts by asking the customer to describe their needs from a blank slate. The developer then laboriously elicits more details, gaining a largely superficial understanding of the customer's actual needs. Customers know what they need in broad terms that are easily expressed and understood but the details are often misleading, with omissions, misunderstandings, and premature or ill-founded decisions about the product. If developers lack domain competence and rely on the customer's characterization of needs, there can be a significant effort required to understand how those needs should be expressed in a solution. It may be difficult to correlate a particular customer's view of a problem to that of other customers and past solutions. Although the potential always exists to leverage knowledge of past similar problems and their solutions, this can be impeded by a given customer's parochial view of their own circumstances, needs, and potential solutions.

A developer with appropriate domain competence may be able to invert the traditional approach to requirements elicitation. The basis for this inversion lies in product line practices. The value of applying domain-specific problem-solution knowledge to software engineering motivated the development and use of software product line methods, including a streamlined approach to requirements definition based on perceived similarities in the needs of different customers.

The goal in the conception of product lines was to establish a practical approach to building multiple similar high-quality products for a coherent market (i.e., multiple customers having similar needs), but with less effort in aggregate, only marginally greater than that required to build a single product. The resulting requirements method focused on identifying commonalities and variabilities in similar products that would reduce the requirements activity to that of resolving a discrete set of essential decisions that differed among customers. These are decisions that express differing customer needs sufficiently to characterize a particular product to be built, when presented to a developer with appropriate domain competence.

Generalizing from this perspective, a developer who has expertise in a relevant business domain already knows, with a reasonable degree of certainty and precision, what the customer is going to request, including many aspects about which the customer will have uncertainties including aspects that different customers may approach differently. A developer having appropriate domain competence is able to ask questions that will be sufficient to resolve many of these uncertainties and differences, while already knowing how different choices will affect the solution.

A developer's domain-specific competence includes familiarity with

- relevant enabling technology,
- prior problem-solutions in that domain,
- the business context in which a product is to be used, and
- how the customer's enterprise and business practices correspond to others.

Being familiar with past problems and their solutions, the developer is able to create a generalized expression of requirements for products of the type to be built. This avoids having the user express what are actually common needs in an incidentally different form but also exposes differences in customer needs from those with which the developer is familiar. By understanding how one customer's needs differ from others and how such needs have been addressed in previous products, the developer can identify appropriate opportunities for applying, and modifying, past solutions to current needs, allowing for more effort on other aspects that are less well understood.

A significant challenge with this approach is to express the customer's needs in a structure and language/vocabulary that the customer understands. For this, the developer needs to create a canonical expression of requirements that can be systematically translated into the terminology that a particular customer uses. However, there is often a market-level terminology that is shared among customers, reducing this concern.

Seven steps comprise this inverted approach to initial requirements determination by the developer:

- Determine a canonical form of requirements expression (outer and inner views) that is characteristic of the customer's relevant business domain.
- Enter a dialog with the customer to confirm common aspects and identify any discrepancies in the canonical requirements form.
- Elicit preliminary answers to key decision criteria that distinguish the customer's perceived needs and any associated uncertainties.
- Create a customized expression of the canonical requirements, based on customer-specific resolution of decision criteria.

- Review and revise the customized requirements expression until the customer is satisfied that the outer view is a reasonable approximate expression of their needs.
- Build product prototypes that satisfy the inner requirements, modifying the customized requirements based on developer and customer evaluations of each prototype.
- Deliver the product including the associated final outer requirements expression for validation, acceptance, and deployment.

A Few Expository Examples

A look at examples of a few simple requirements elements will illustrate how developer domain-specific competence can enable rapid convergence on a good first approximation of a customer's needs. By recognizing needs that recur broadly for a type of product, analysis efforts can quickly narrow to focus on singularities, uncertainties, and tradeoffs that are most significant to that customer. Examples from a report on real-project experiences of a device producer illustrate how domain competence can expedite the elicitation and specification of requirements [9].

The implication with each of these examples is that a developer with appropriate domain expertise would be able to formulate a canonical expression of requirements for a customer based on such examples. This formulation would be customized with the customer based on their answers to a characteristic set of discriminating decisions, including any necessary localization of terminology. This is meant to give a first approximation to customer needs much more quickly, eliminating the need to spend effort on what are predictable needs, and permitting more effort on unprecedented and uncertain or poorly understood needs. It also supports the possibility of earlier prototypes of the product that would both increase customer confidence and expose residual misconceptions and misunderstandings.

The first example concerns a requirement that is deemed as expressing a design constraint, that products need to be designed and packaged so as to achieve maximal fit on standard sized shipping pallets. This is possibly a universal requirement: it is unlikely that any customer would choose to forego this as a requirement unless it conflicted with some more compelling tradeoff. Furthermore, it is likely in any case that a designer would anticipate this as a constraint. It is presumably in the requirements only so that it will not be overlooked. In a conventional elicitation, this might not arise as a requirement if the customer happens to take it as a given and fails to express it as such. Leaving it to the customer to specify this as a need has the added detriment that different customers may express this in different terms even though a single canonical expression might in fact suffice for everyone. Alternatively, perhaps there are alternatives that the customer has not realized simply because this has always been their practice. If there are credible alternatives to this constraint, the developer should be offering them to the customer as an option, rather than just assuming that the customer might have already considered and dismissed such alternatives.

A second example concerns how instructions are to be attached to the product. The customer may be accepting past practice without a proper analysis of alternatives and tradeoffs. Certainly different customers might prefer to attach instructions differently or enclose them unattached in packaging or not enclose instructions at all. However, the choices in such a case are not very numerous or unpredictable. There might very well be, for example, only four choices: attached to front, attached to back, enclosed loose, or omitted. Left entirely to the unguided discretion of the customer, they might arbitrarily require a totally different option even if one of the limited standard choices would have been perfectly acceptable or even preferred but not considered. By offering the customer limited choices, those will often suffice and be quickly settled but still nothing precludes adding new options for one customer.

A third example concerns the ways in which a product can be influenced (providing capabilities needed for interaction with another product) or constrained by other interacting products (lacking capabilities that might otherwise be used). Knowledge of these influences and constraints are not particular to the immediate customer but are probably known to all producers of similar products. Having each producer describe these constraints separately, assuming all descriptions were correct, would differ only because they were written by different people or because each omitted different information that was not relevant to their particular product's use. With the developer already knowing about these, the customer would not have to spend time describing them but only needs to relate any special concerns about them.

A final example illustrates a case where user needs are initially valid as specified but are probably over-constrained relative to the long-term evolution of the product. In this example, the customer specified that the software must accommodate either of two particular connector devices. These appear to be arbitrary requirements as no rationale is given for these specific choices; a better approach would be to specify what about these two and only those make them preferred. A less specific requirement might permit building the software so it would be easier to modify to accommodate other devices in the future. Ideally, the software should be built, if feasible, to support any devices having the properties that led to the selection of these if that were known. An alternative would be for the developer to specify particular device types in the outer requirements but define the inner requirements to

accept any devices having the required characteristics, anticipating the possibility of alternative devices as the technology evolves.

It should be clear from these examples that there is a potential to eliminate a good portion of the initial requirements engineering effort if the developer has domain competence. Admittedly, this is not an option for a developer who is only a generalist in software and expects to elicit needed domain knowledge from each customer. This is a disservice to the customer but many customers seem to accept this as the only alternative to buying packaged products that might exist in some form but would in turn either require substantial customization in their own right or require the customer to change their business practices to fit the software.

The lesson of these examples is that within a given domain many decisions as to what is required are foreseeable, predictable and to a greater or lesser degree an expression of common practice or even common sense. Performing a requirements analysis as if every effort should begin with a blank slate that each customer must fill is wasteful and symptomatic of the immaturity of the software engineering discipline. The remediation to this waste is to start from a base of domain-specific competence upon which each customer's particular needs can be consistently focused, exposed, and elaborated. Even a first time effort to build a truly unprecedented product will begin with reference to known similar products – even an entirely new product will be based to a good degree on similar existing products because no product today is unique in all aspects. This has the benefit of limiting effort that needs to be spent on precedented aspects, permitting more effort to be focused on exploring truly new or unique aspects.

VIII. SUMMARY

The development of a software product can be thought of as a search through the space of all possible problems and their solutions. With a traditional approach, the goal is poorly defined and the space is an undifferentiated set of potential products. With the application of domain competence, this amorphous space coalesces into recognizable subspaces of similar problems having similar coherently associated solutions. By first mapping a customer's perceived needs onto a familiar subspace of problems that represent similar jobs to be done, the process of better approximating customer needs and then resolving uncertainties and tradeoffs to efficiently construct a responsive product can be more rapidly and predictably performed.

REFERENCES

1. S. Faulk, J. Brackett, P. Ward, and J. Kirby, "The Core Method for Real-Time Requirements", IEEE Software 9 (5), 1992, 22-33.
2. M. Jackson, "Problems, Methods, and Specialization," IEEE Software 11 (6), 1994, 57-62.
3. V. Basili, G. Caldiera, and H. Rombach, "The Experience Factory," Encyclopedia of Software Engineering, John Wiley & Sons, Inc., 1994, 469-476.
4. G. Campbell et al, *Reuse-driven Software Processes (RSP) Guidebook*. Herndon, Va: Software Productivity Consortium, 1994. <www.domain-specific.com/RSPgb>
5. S. Faulk, et al, "Scientific Computing's Productivity Gridlock: How Software Engineering Can Help", IEEE Computing in Science and Engineering 11 (6), 2009, 30-39.
6. G. Campbell, "The Illusion of Certainty", Naval Postgraduate School, 7th Annual Acquisition Research Symposium, May 2010, 257-264. <www.domain-specific.com/PDFfiles/NPS-AM-10-022-ghc.pdf>
7. A. Niknafs and D. Berry, "The Impact of Domain Knowledge on the Effectiveness of Requirements Idea Generation during Requirements Elicitation", RE 2012, Chicago, IL, 2012, 181-190.
8. R. Salay, M. Chechik, and J. Horkoff, "Managing Requirements Uncertainty with Partial Models", RE 2012, Chicago, IL, 2012, 1-10.
9. J. Savolainen, D. Hauksdóttir, and M. Mannion, "Challenges in Balancing the Amount of Solution Information in Requirement Specifications for Embedded Products", RE 2013, Rio de Janeiro, Brasil, 2013, 256-260.
10. S. Faulk, "Understanding Software Requirements", *Software Engineering Essentials, Volume I: The Development Process*, R. Thayer. M. Dorfman, Eds., Software Management Training Press, Carmichael, CA, 2013, 1-42.