On the Role of Requirements in Understanding and Managing Technical Debt

Neil A. Ernst
Department of Computer Science
University of British Columbia
nernst@cs.ubc.ca

***Abstract*—Technical debt is the trading of long-term software quality in favor of short-term expediency. While the concept has traditionally been applied to tradeoffs at the code and architecture phases, it also manifests itself in the system requirements analysis phase. Little attention has been paid to requirements over time in software: requirements are often badly out of synch with the implementation, or not used at all. However, requirements are the ultimate validation of project success, since they are the manifestation of the stakeholder's desires for the system. In this position paper, we define technical debt in requirements as the distance between the implementation and the actual state of the world. We highlight how a requirements modeling tool, RE-KOMBINE, makes requirements, domain constraints and implementation first-class concerns. RE-KOMBINE represents technical debt using the notion of optimal solutions to a requirements problem. We show how this interpretation of technical debt may be useful in deciding how much requirements analysis is sufficient.**

I. Introduction

Technical debt is, per Brown et al., "a situation in which long-term code quality is traded for short-term gain" [1]. Consequently, much attention on managing technical debt focuses on problems with intrinsic software quality, such as code, that must eventually be re-factored. However, the paper by Brown et al. also mentions the issue of debt in non-code artifacts, such as requirements, which may manifest as "the relationship between numbers of TBDs [to-be-determineds] and maintenance costs or stability."

Technical debt in the requirements phase of system design is different from that in the implementation phases. Technical debt in the implementation refers to choices, deliberate or otherwise, to sacrifice intrinsic quality for short-term expediency. This may take the form of violations of design principles such as low coupling or higher-level architectural decisions that favour performance over long-term scalability.

One way to understand the relationship between technical debt in requirements and technical debt in design or code is by looking at the relationship between product value (possibly measured in satisfied requirements) and intrinsic software quality. If the product is not delivering value to the client, then high intrinsic quality (i.e., low technical debt) is irrelevant. If intrinsic quality is low, i.e., has a lot of debt, than the likelihood of easily delivering valuable products in the future is likewise low.

Technical debt in requirements is incurred when we decide to prioritize requirements (and thereby build a particular product) which are ultimately neither necessary nor deliver the most value to the customer. Often, this occurs because of inadequate or poorly conducted requirements elicitation and analysis. Debt incurred in the requirements phase refers to tradeoffs on what requirements the development ought to prioritize. If requirements later change, we must update the design; but it is possible that this update would not have been necessary had we correctly predicted what requirements would be needed in future (for example, adding the capability to perform wireless updates to a television). In the entirety of the product's lifespan, then, technical debt and 'gold-plating' are thus duals of each other.

In this position paper, we discuss the nature of technical debt in requirements. We discuss what forms this debt may take, and suggest some ways to understand and manage it. We make the following claims:

- That technical debt in requirements is the distance between the optimal solution to a *requirements problem* and the actual solution, with respect to some decision space;
- That interest on the debt can be characterized as the rate of increase in this distance;
- That being optimal is inherently uncertain, because of unanticipated changes due to software evolution;
- That requirements must be treated as first-class objects in system design, and preserved throughout the lifecycle.

We first introduce our formulation of the requirements problem as a tool for understanding where debt arises (§II), and one approach to using this formulation to model the technical debt present in requirements tradeoffs (§III). We then briefly introduce RE-KOMBINE (§IV), which can aid in the design process. We discuss related work that explores the issues of technical debt (or other names under which it appears), and conclude with implications for future work in this area.

II. The Requirements Problem

To ground our discussion, we begin by introducing our view of how requirements are used in building software. We start from Zave and Jackson's [2] definition of the *requirements problem*: given requirements G, and domain

assumptions D, find specification S, such that together, D and S satisfy the requirements G.

We represent requirements G as goals, which capture desired properties of the system, including whether satisfying a goal is mandatory or is preferred to other goals. Goals capture what the customer considers valuable. The goals may be satisfied by specifications S, represented as sets of *tasks S* referring to behaviours of the system-to-be. Notice that the specifics of S are deliberately vague: a set of tasks might take the form of delegations, commitments, services, or code, among others. Finally, *domain assumptions D* are conditions believed to hold in the world, as well as statements that describe how the problem itself is structured. This includes statements such as "Task T will satisfy Goal G" and "Task T and Task V cannot co-exist".

We have captured this modeling approach in a tool, RE-KOMBINE, which is discussed at length in [3]. It is based on the Techne language [4]. In RE-KOMBINE, requirements problems are structured, representing notions ranging from high-level objectives ("sell more products") to low-level tasks ("use Moneris payment terminals"). We must represent these requirements, and their eventual refinement into tasks, using the statements from the stakeholders and knowledge about domain constraints. And, as the next section will discuss, this information is subject to change.

In addition to modeling requirements problems, RE-KOMBINE can also find solutions to them. First, it identifies all *potential* solutions to the problem—all sets T which achieve the goals G. The final step in RE-KOMBINE is the selection of a single solution, using some techniques to rank alternatives—for example, cost. The key insight here is that there are possibly many, equally optimal solutions to a given requirements problem.

III. Representing Technical Debt in RE-KOMBINE

RE-KOMBINE can also manage evolving requirements problems. This is important for managing technical debt, since it is only when the requirements problem changes that we can truly capture the presence of technical debt. This is because RE-KOMBINE will always present the optimal solutions at a given point in time. This has an analogue in technical debt in the implementation: even though we may implicitly know that a particular architectural approach is not optimal in the long-term, presumably we choose this approach precisely because it works at that point in time. It is only when time passes that debt begins to accrue.

Suppose that we have selected a particular set of tasks S as the initial implementation. It is commonly accepted, e.g., [5], that we are likely to encounter unanticipated changes to all aspects of requirements models expressed using RE-KOMBINE, including the goals, tasks, and even the structure of the problem itself (i.e., our beliefs about how goals can be satisfied). Suppose this results in a new requirements problem. We could (naively) search for a new solution as before. However, this view is unrealistic in the real wold: for example, one does not throw away an entire software system and start design from scratch, when a new feature is desired. Instead, the solution is likely to be *incremental*: start from the current solution and try to move towards one that satisfies the changed problem.

Let us define a distance metric, Π, which captures the difference between two sets of tasks T. In previous work, we have used various choices for this metric Π, which either emphasize the preservation of previously implemented features, or seek to minimize the overall implementation effort, even at the expense of 'legacy' code. Other metrics come to mind: we might be fortunate enough to have data on cost, on implementation time, or on value to the stakeholders.

We can now define the issue of technical debt as captured in RE-KOMBINE. It is the distance δ between the original solution, S_1, and the changed requirements problem. In other words, if the set of tasks identified as the initial solution, and then implemented, still optimally satisfies the new requirements problem, then there is no technical debt incurred in the time since the solution was initially delivered. Despite the changes in (say) the requirements, the original implementation satisfies those new requirements as well (which may indicate over-engineering, a separate issue). Thus the amount of debt incurred in making the choices captured by S_1 is equivalent to the eventual distance S_1 drifts from the optimal solution S_2 at some time later in the future.

With respect to "interest" on the debt, which we loosely translate as the cost of neglecting the debt, this is the rate of change between the current solution and the current state of the requirements problem. In other words, *interest* captures the rate of change in debt.

Example. Consider the requirements for a retail IT system which must respect payment card security standards (PCI-DSS). One requirement might be the need to accept VISA cards. Additionally, a constraint in this case is that networking relies on a system of WEP-secured wireless routers to send payment information from terminals to back-end servers. S_1 is the implemented IT system (at time t_1) that is created to respect these properties; among other capabilities, S_1 accepts VISA cards and also relies on the pre-existing wireless infrastructure. At time t_2, changes to the PCI-DSS standard have since forbidden the use of WEP-encryption, so our current solution S_1 is not optimal with respect to the updated requirements problem. In this scenario, technical debt was incurred when the decision was made to accommodate the use of legacy WEP-encryption.

How might this debt have been avoided? In this situation, there are two possible approaches (at t_1): one, ignore the obvious problems with WEP (which are well-documented) and choose the most expedient solution. Well-managed IT departments would insist on parallel development of a

backup solution, perhaps using the WPA standard. Two, insist on requirements forecasting and remove the WEP constraint. This can be done, but at the expense of up-front costs and the small probability that the PCI-DSS does not forbid it at t_2, thus making the up-front costs redundant. The distance δ captures the difference between S_1 and S_2, and can (for example) capture the cost of the work-around, or the parallel development.

IV. Using RE-KOMBINE

Underlying RE-KOMBINE is a knowledge-base approach to managing requirements. It stores the main components of the requirements problem as propositional variables connected by either inferences or conflicts. This provides us with a relatively light-weight formalism that can manage the three aspects of the requirements problem, and also perform automated reasoning to find solutions.

Like all knowledge bases, RE-KOMBINE defines two primary operations. TELL operations are for adding new or changed information to the knowledge base, and ASK operations are for obtaining possible solutions to our requirements problems. In particular, we want to know three things: (1) given a set of tasks S, do those tasks satisfy our requirements G? (2) Given a set of requirements G, which sets of tasks S satisfy those requirements? (3) Given a change in the requirements problem (i.e., a change in goals, domain assumptions, or implementation), what are the new tasks which satisfy that changed problem?

Figure 1 illustrates a possible software development approach using RE-KOMBINE. We begin by eliciting some set G of requirements, in the form of goals, and domain assumptions D, which constrain the environment. We then refine these desires and constraints into tasks, producing the initial knowledge base (1). The ASK operations of RE-KOMBINE are then used to identify whether there exists a viable set of implementation tasks to satisfy the goals (2). If we get no solution (3) we must revise our initial model. Otherwise (4) we obtain multiple answers and choose one, which we then mark in the knowledge base (5). This allows us to compare one implementation to new, proposed implementations, as in software repositories. The system is then implemented, and post-release, we monitor the requirements, assumptions and implementation for any changes. If we anticipated the change, we can self-adaptively find the best new solution (6); otherwise, we must revise the REKB manually (7). Technical debt measures, specifically the distance metric Π, are calculated at step 3, and allow us to select the best new solution, as well as monitor historical performance. For example, if we measure development size (in LOC) we minimize Π to select the new solution with the fewest additional lines of code. Tracking Π over time allows us to understand our historical performance over the series of iterations.

V. Challenges and Future Work

We now discuss next steps for understanding technical debt in requirements by considering some of the challenges this approach poses.

Biased to up-front planning. An objection to this model is that it requires designers to decide up-front those requirements which will be necessary, lest technical debt be incurred. Up-front analysis is necessary since technical debt is capturing the *distance* between the initial solution and the current state of affairs, which in turn argues for up-front estimates and planning to try to minimize this distance, e.g., more effort be expended on requirements elicitation. Ideally, we would balance the need to accurately capture the requirements up-front against measures like cost of delay and probability of success (e.g., in a relatively simple system in a well-known domain, the probability of successfully understanding the requirements is high).

We would argue that this particular notion of technical debt does not make a value judgement on incurring debt; rather, it is a way of measuring the extent to which a given system changes over time. By tracking the historical tendency of these distance measures, however, we can learn something about our tradeoff decisions. If historically our analysts are always under-estimating the changes which happen (δ is large), this suggests more time could be spent in the initial requirements analysis. It is also important to consider the opposite case, of gold-plating: in these cases we handle many possible changes, some of which will never happen. A way of distinguishing between good leverage and bad is important.

Difficult to track requirements. Another objection is that few teams track requirements and other artifacts at the level which RE-KOMBINE would seem to demand. However, while the representation is formal, and benefits from automated reasoning support to explore the solution space, the requirements themselves can be as vague and ambiguous as one might wish, since we only capture them as natural language statements. Furthermore, RE-KOMBINE supports paraconsistency, which allows for the modeling of conflicting requirements. We are exploring tools which can semi-automate the capture of the three elements of the requirements problem in order to assist the user in modeling the problem. A related challenge is the monitoring component, which is an active area of research.

VI. Related Work

Frameworks like the NFR approach [6] explicitly model requirements tradeoffs, using qualitative reasoning to express degrees of satisfaction of stakeholder goals. Contributions to the satisfaction of non-functional requirements are used to guide the ultimate result of the design process. Goal oriented requirements frameworks such as that described in [7] build on these ideas. Most recently, there has been work on adaptive systems using "requirements at runtime", e.g.,

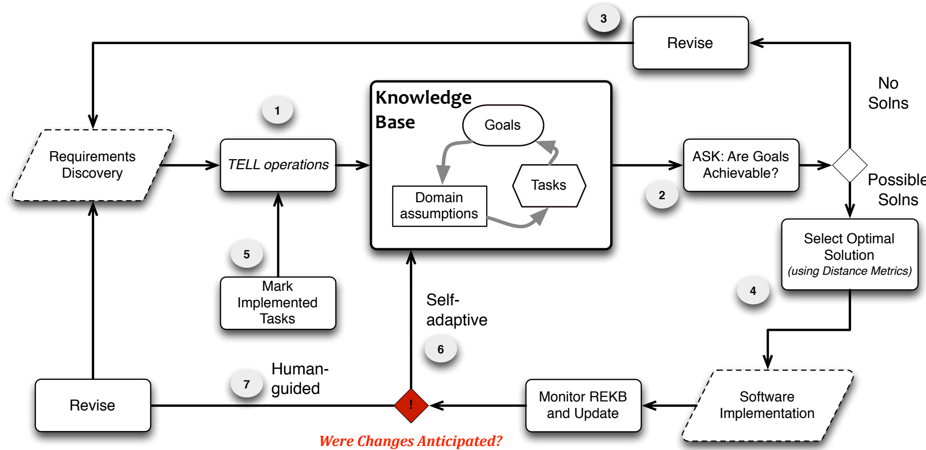

Figure 1. Using RE-KOMBINE to manage requirements problems

[8]. However, this approach insists that requirements artifacts are present at the initial design of the system itself.

A recent industrial case study on requirements change pointed out that while requirements changes do indeed have great impact on project cost, they also produce great value [9]. This is an important consideration in understanding the notion of technical debt in requirements. Finally, there is evidence that shows that many requirements often lead to features which see little use in practice, representing a waste of investment [10]. This is debt realized as an opportunity cost: the loss of the ability to spend developer time on other projects or requirements. In other words, choosing to prioritize requirements that result in unused features is to incur prioritize the short-term over the longer-term (even if longer-term, in this case, may mean other projects entirely).

VII. Conclusions

In this paper we have described our view of how technical debt is manifested in software requirements. We introduced RE-KOMBINE, which is a lightweight formalism for understanding how implementations match stakeholder goals. Technical debt in requirements is the distance between the current implemented system and the current state of the requirements problem. This is a way of understanding the relationship between the value a system delivers, and the internal quality it exhibits. Tracking requirements throughout the software lifecycle enables useful measures of technical debt in requirements. Finally, we argued that it is the historical performance with respect to requirements debt that is of interest, since that can inform us about the degree to which we over- or under-analyse our requirements.

References

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing Technical Debt in Software-Reliant Systems," in *FSE Workshop on the Future of Software Engineering Research*, Santa Fe, 2010.

[2] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *Trans. Soft. Eng. Method.*, vol. 6, pp. 1–30, 1997.

[3] N. Ernst, A. Borgida, J. Mylopoulos, and I. J. Jureta, "Agile Requirements Evolution via Paraconsistent Reasoning," in *International Conference Advanced Informations Systems Engineering*, Gdansk, Jun. 2012.

[4] I. J. Jureta, A. Borgida, N. A. Ernst, and J. Mylopoulos, "Techne: Towards a New Generation of Requirements Modeling Languages with Goals, Preferences, and Inconsistency Handling," in *International Conference on Requirements Engineering*, Sydney, Australia, Sep. 2010.

[5] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 3, pp. 225–252, 1976.

[6] L. Chung, J. Mylopoulos, and B. A. Nixon, "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach," *Trans. Soft. Eng.*, vol. 18, pp. 483–497, 1992.

[7] R. Sebastiani, P. Giorgini, and J. Mylopoulos, "Simple and Minimum-Cost Satisfiability for Goal Models," in *International Conference Advanced Informations Systems Engineering*, Riga, Latvia, Jun. 2004, pp. 20–35.

[8] N. A. Qureshi, A. Perini, N. A. Ernst, and J. Mylopoulos, "Towards a Continuous Requirements Engineering Framework for Self-Adaptive Systems," in *Requirements at Run-time at RE*, Sydney, Sep. 2010.

[9] S. Mcgee and D. Greer, "Software Requirements Change Taxonomy : Evaluation by Case Study," in *International Conference on Requirements Engineering*, Trento, Italy, Sep. 2011.

[10] D. Cohen, G. Larson, and B. Ware, "Improving software investments through requirements validation," in *26th Annual NASA Goddard Software Engineering Workshop*, Greenbelt, MD, Nov. 2001, pp. 106–114.