2012 Joint Working Conference on Software Architecture & 6th European Conference on Software Architecture

In Search of a Metric for Managing Architectural Technical Debt

Robert L. Nord, Ipek Ozkaya
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
{rn, ozkaya}@sei.cmu.edu

Philippe Kruchten, Marco Gonzalez-Rojas
Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada
{pbk, marcog}@ece.ubc.ca

Abstract— **Practices designed to expedite the delivery of stakeholder value can paradoxically lead to unexpected rework costs that ultimately degrade the flow of value over time. This is especially observable when features are developed based on immediate value, while dependencies that may slow down future development efforts are neglected. The technical debt metaphor conceptualizes this tradeoff between short-term and long-term value: taking shortcuts to optimize the delivery of features in the short term incurs debt, analogous to financial debt, that must be paid off later to optimize long-term success. In this paper, we describe taking an architecture-focused and measurement-based approach to develop a metric that assists in strategically managing technical debt. Such an approach can be used to optimize the cost of development over time while continuing to deliver value to the customer. We demonstrate our approach by describing its application to an ongoing system development effort.**

Keywords: technical debt; software architecture; software economics; cost of rework; total cost of ownership

I. INTRODUCTION

The term 'technical debt' describes an aspect of the tradeoff between short-term and long-term value in the development cycle. Recently the concept has gained increased visibility, advanced mainly by promoters of agile software development approaches. An ongoing focus on the management of technical debt is perceived as critical to the development of high-quality software. While expressing the value of user-visible features may be straightforward, a key challenge in iterative development is the ability to quantify the value of infrastructure and quality-related tasks, which quite often are architectural. Technical debt also closely relates to system evolution challenges, as often an organization finds itself with the need to deal with accumulated technical debt when upgrading technology or adding new features to a legacy environment. Left unmanaged, technical debt causes projects to face significant technical and financial problems, leading to increased maintenance and evolution costs. Although agile practices of refactoring, test-driven development, and software craftsmanship are often sufficient to manage technical debt on small-scale projects, sound principles for managing debt on large-scale, enterprise-level projects are lacking. Project owners drawn to agile practices by the allure of quicker time to market and improved responsiveness can find the promise of agility negated by increasing mountains of technical debt.

Within the last few years technical debt in system development has received increased attention as a result of a growing need to manage the degrading quality of software and avoid costly reengineering and maintenance. Cunningham introduced the technical debt metaphor in defense of relentless refactoring as a means of managing software debt [1].

"Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

Cunningham's description has served as the starting point and as a definitional reference in describing technical debt [2]. Both McConnell [3] and Fowler [4] categorize technical debt into distinct types, separating issues arising from mere recklessness from those decisions that are made strategically. Work to date on technical debt has mainly focused on uncovering and paying back unintentional or reckless technical debt, mostly found in the form of degrading code quality [4]. Underlying the strategic type of technical debt is the prudent and deliberate aspect of debt that can speed the development effort [3]. While there is increasing recognition within the software engineering community that architecting and the delivery of high-value customer features must go hand-in-hand, there have not been quantitative approaches that provide guidance on monitoring technical debt strategically in order to manage the total cost of ownership across a project.

In this paper we'll endeavor to provide a foundation for incurring and managing prudent, deliberate, intentional technical debt. In that regard we embrace the following definition by McConnell:

"*Technical debt is a design or construction approach that's expedient in the short term, but that creates a technical context in which the same work will cost more to do later than it would cost to do now.*" [5]

We use software architecture as a means to identify and monitor architectural technical debt, or architectural debt, so that re-architecting and refactoring decisions can be made in a timely manner. We illustrate our approach by describing an ongoing development effort: the Disaster Response Network-Enabled Platform (DRNEP). DRNEP system integrates a set of independently developed infrastructure and disaster simulators [6]. The DRNEP system is currently undergoing major re-architecting efforts. We compare these efforts' alternative development paths: one that takes on debt and one that does not, and we explore their economic implications. In doing so, we focus on development of a metric that can provide insights into the accumulating rework cost that is factored into a debt payback strategy.

The rest of the paper is organized as follows: Section 2 presents our architectural debt rework model. We illustrate how architectural debt can be elicited and monitored based on rework in Section 3, using the ongoing system development effort of DRNEP as an example. We review related work in strategic technical debt analysis in Section 4. In Section 5 we discuss the implications of our findings, and Section 6 concludes the paper.

II. ARCHITECTURAL DEBT: MODELING REWORK

A key enabler to managing architectural debt is the ability to quantify degrading architecture quality and the potential for future rework cost during iterative release planning. Engaging strategic design shortcuts, which can be interpreted as going into architectural debt, requires effective characterization of the economics of architectural violations across a long-term roadmap, rather than enforced compliance for each release.

A key challenge in agile development is in acquiring the ability to quantify the value of infrastructure and quality-related tasks that are quite often architectural. For example, under the widely adopted Scrum practices, such as managing a product and sprint backlogs [7], zero value points are typically assigned to infrastructure development, re-architecting, and maintenance tasks, including paying back technical debt, which are all perceived as costs only. This assignment results largely from an inability to value the long-term impact of these infrastructure or architectural elements on customer-visible features and overall system health as the system grows. These elements appear to have no customer value—only development cost. Other approaches aim to adjust this fallacy by calculating customer-visible-feature percentages based on value points in order to adjust the overall value assigned. For example if there are five features with value points 5, 3, 2, 2, 1, then their overall percentages would be 40%, 23%, 15%, 15%, 7% [8]. This does not capture, however, the impact of architecting and related rework and maintenance.

The cost models of implementing a design choice today, versus retrofitting a solution much later, are often wrongly compared. Consider an internationalization feature of a web site. The development team may decide to implement full flexible language support that would enable the system to be sold, possibly in international markets today, under a given cost model; let us say the cost is *C*. Alternatively, the team may decide to postpone internationalization to a later date when it becomes absolutely necessary (YAGNI, or "defer to the last responsible moment"). Until that later date the system continues to grow. When in fact the time to absolutely introduce internationalization arrives, the cost is no longer *C*, but it is *C* + *R* where *R* represents the impact of retrofitting the grown system in order to introduce internationalization. As this canonical example demonstrates, modeling architectural debt requires modeling the dependencies between features and architectural elements, and using that information to calculate the cost of implementing each feature based on the existing capabilities of the infrastructure.

A strategic perspective of prudent, deliberate, technical debt encompasses key aspects of total cost of ownership management in large-scale, long-term projects:

- Optimizing for the short term puts the long term into economic and technical jeopardy when debt accumulates and is unmanaged.
- Design shortcuts can give the perception of success until their consequences start slowing projects down.
- Development decisions, especially architectural ones, require active management and continuous quantitative analysis, as they incur implementation and rework cost to produce value.

Technical debt management involves making choices between a focus on value versus on cost throughout the development cycle as decisions are made (often in the planning cycle of each iteration or sprint). Some agile approaches, with a focus on value, result in an implementation that accumulates debt. Other more phase-based approaches, with a focus on cost, result in a delay of value as releasing the product takes longer. These are the boundaries of the decision space.

Technical debt management is about navigating a path that considers both value and cost, to focus on overall return on investment over the lifespan of the product. Within that path, debt has a lifecycle: the time it is incurred, the time frame during which it accumulates interest, and the time required to develop a payback strategy and put it into action. Managing debt is dependent on knowing the impact of key requirements and architectural steps.

Our position and experience is that there can be value at times in making suboptimal decisions to support the overall goals of the business and the project. This decision-making process requires creating a metric for debt management that takes into account the interest as well as repayment of the borrowed amount. The metric we use to model this metric is based on architectural rework.

We model rework as an aspect of changing dependencies. In a previous study, we conducted an exploratory analysis of a model problem to quantify the technical debt outcomes of alternate release strategies. We engaged this approach to establish metrics for quantifying architecture quality where we used architecture structure metrics based on dependency analysis [9]. This model computes the rework cost associated with each new architectural element E_i implemented in release n.

The Total Cost T of release n is a function of the implementation and rework costs, Ci and Cr :

$$T = F(Ci, Cr)$$

In this paper, we assume that it is simply the sum.

The implementation cost Ci for release n is computed as

$$\sum_k Ci(E_k) \text{ for all new elements } E_k$$

where the implementation cost $Ci(E_k)$ is given for all individual architectural elements k.

The rework cost Cr for release n is computed as

$$\sum_k Cr(E_k) \text{ for all new elements } E_k$$

and

$$Cr(E_k) = \sum_j Cr(E_j) \text{ for all pre-existing elements } E_j$$

If E_j is an element implemented in a prior release,

$$Cr(E_j) = D(E_j, E_k) \times Ci(E_j) \times Pc(n-1)$$

where $D(u,v)$ is the number of dependencies from u to v, Ci is the implementation cost, and Pc (n - 1) is the change propagation of release n - 1. Change propagation is a metric introduced by MacCormack et al. [10] that captures the percentage of system elements that can be affected, on average, when a change is made to an element. The change propagation metric of a system is computed as the density of the visibility matrix that captures all the direct and indirect dependencies in the system architecture, or in other words, the transitive closure of the dependency relationship.

III. ARCHITECTURAL TECHNICAL DEBT IN ONGOING DEVELOPMENT: A CASE STUDY

The Disaster Response Network-Enabled Platform (DRNEP) is a system that integrates a set of independently developed infrastructure and disaster simulators [6] (see Fig. 2). Developed primarily at the University of British Columbia, with collaborators in various parts of the world, its purpose is to provide government agencies means to prepare and respond to large disasters, such as earthquakes, tsunamis, flooding, or hurricanes. It provides insights into the interoperation of various critical infrastructures: electrical grid, transportation, communication, water, and so on. Such infrastructures often are run by various public or private organizations that don't recognize the cascading effects of major failures or the decisions to remediate them or the propagation of their impact [11]. The implementation of DRNEP enables knowledge gathering on the interoperation of interdependent infrastructures specific to a given area from expert organizations around the world, in order to assist local emergency responders. To achieve this end, organizations rely on simulators that help them to create disaster scenarios for a geographical zone (e.g., a municipality, region, or province) and elicit strategies on how to better respond to them.

DRNEP grew initially with relatively little thought on an overall architecture. Different simulators were developed by independent research groups over a period of several years, each catering to a different concern or a different infrastructure. Central to DRNEP is the Infrastructure Interdependency Simulator (I2Sim) [12] [13] as the integrating simulator that models a given geographical area in case of disaster. I2Sim interfaces with other specialized simulators by exchanging information at a given sampling time. The simulators can interact with transforming elements, or control points in the model and are able to model a given physical entity, a human, or set of humans making decisions. To integrate the simulators in DRNEP, there are two possible paths: one, which seems easy and incremental, is to make ad hoc adaptors and translators between the concepts and data of the simulator and DRNEP.

A second, more ambitious path is to define a standard architecture with mechanisms to achieve a more systematic integration, using a mediator pattern, with a common canonical data model, and using an Enterprise Service Bus (ESB) for implementation [6].

This second path looks more costly at first. We show here that it will pay off rapidly with subsequent releases, as more simulators are integrated in DRNEP. If the project were to embark on the first path for easy early successes, it would accumulate heavy architectural technical debt with high rework costs (55%).

A. *Architecture Debt Analysis*

The DRNEP system will connect different simulators and requires the ability to add simulators to work collaboratively; value increases as more simulators are added. We analyze two paths:

Path #1: Deliver soon. New simulators are integrated one by one, and ad hoc translators for data and communications are built incrementally. Adding a new simulator requires building a new translator for each existing simulator and reworking each of the existing simulator adaptors.

Path #2: Reduce rework and enable compatibility. An Enterprise Service Bus is put in place and a central common data model is designed. Adding a new simulator requires building one additional element on top of the common infrastructure to provide adaption and transition services.

1) *Path #1: Deliver soon.*

In order to deliver a working version of the system quickly, the plan calls for making the minimum required effort at the beginning. This implies that the elements will be coupled with each other. The emerging architecture in this minimal architecture version is shown in Fig. 1.

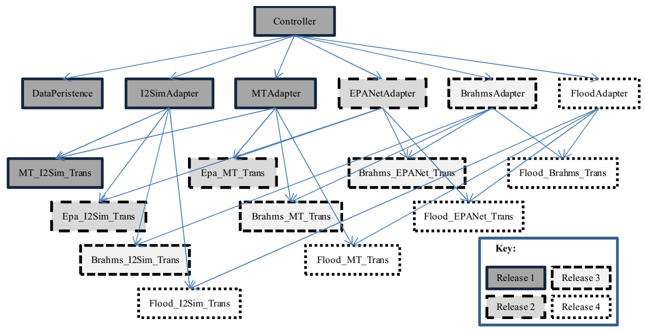


Figure 1. Path #1 dependencies

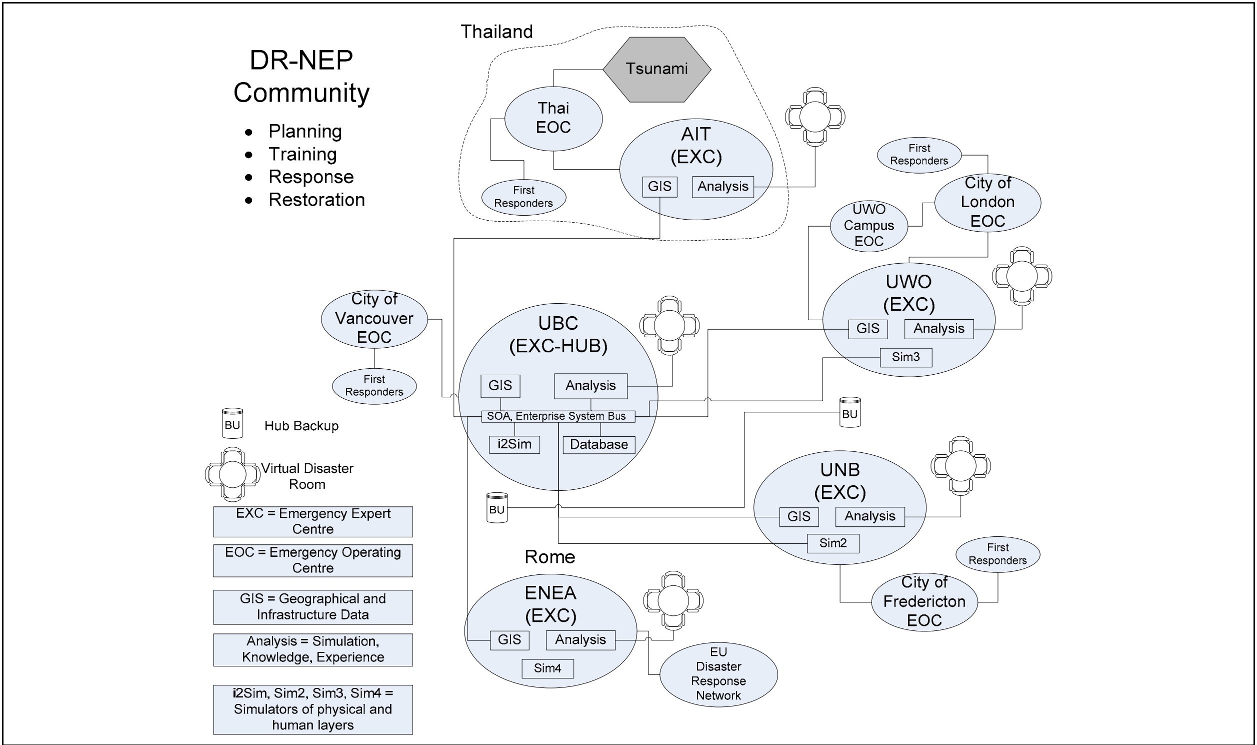


Figure 2. DRNEP Network Configuration. Canadian institutions are linked through CANARIE and associated networks' dedicated light paths to exchange and distribute data, simulation, and analysis functions. ENEA is in Rome, Italy, and University of New Brunswick (UNB) is in Fredericton, NB. Thailand is the disaster source for the tsunami exercise. University of Western Ontario (UWO) in London is the disaster source for the UWO Campus exercise. Sources: [6][11]

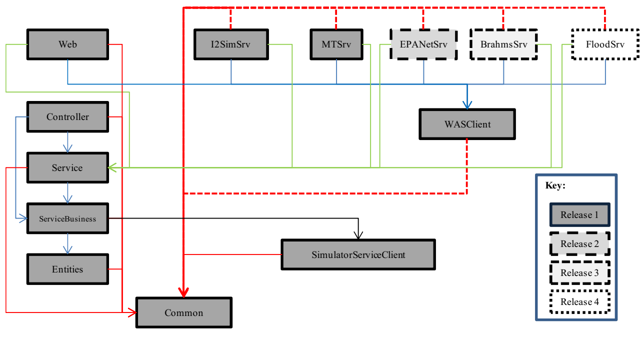


Figure 3. Path #2 dependencies

2) Path #2: Reduce rework, enable compatibility.

One of the objectives of DRNEP is to enable compatibility of simulators and, to achieve this, mediators are used to decouple elements in the system. This increases compatibility and reduces maintenance in the subsequent releases. However, this approach requires an investment in infrastructure during the first deliveries, and so delays delivery. The design uses an ESB in order to mediate data exchange and a common data model in order to decouple data representation within elements of the system. The common data model is defined in the service module and each service has a copy of it to avoid repeating the definition, this is captured by the dependency of each service into the service module. The emerging architecture is shown in Fig. 3.

The dependencies shown in Figs. 1 and 3 are depicted as a design structure matrix (DSM) in Fig. 4 to facilitate a comparative analysis. A DSM is a matrix that maps dependencies between items in a given domain [14]. All elements appear both in the rows and the columns and dependencies are signaled at the intersection points of the items in the matrix. For instance, in Fig. 4 the mark at the intersection of row 3 with column 1 means that element in column 1 (Controller) depends on element in row 3 (MTAdapter). DSMs are single-domain square matrices, meaning that relations are defined between instances of the same type (architectural elements in Fig. 4).

Comparing the two DSMs of the alternative architecting approaches reveals that the approach that focuses on delivering sooner incurs more dependencies between the adapters and translators of the individual simulators of the DRNEP system. In contrast, the cost reduction approach results in an architecture with fewer dependencies between the individual simulators, and those dependencies are well encapsulated by separating concerns within WASClient, Service, and Common elements.

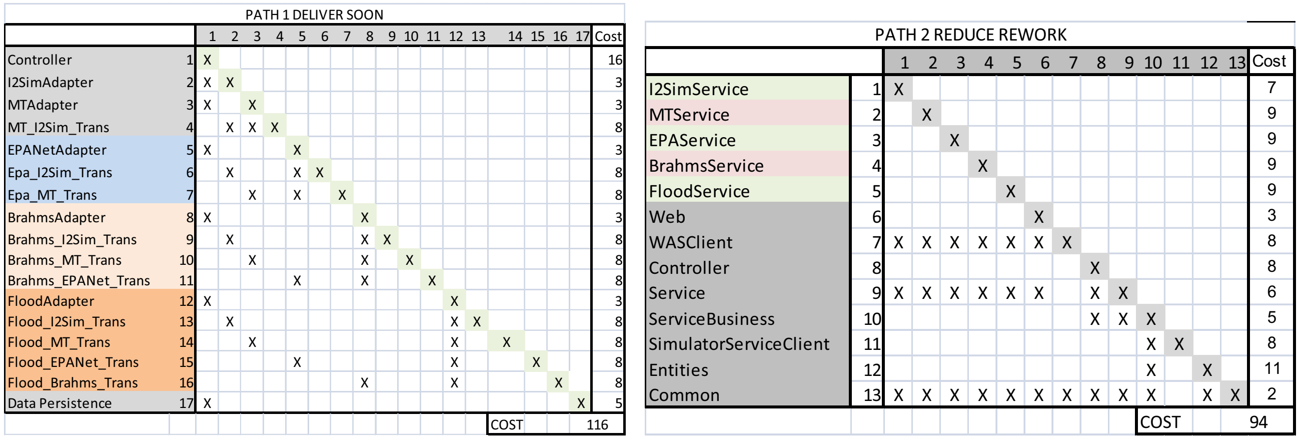
The comparative analysis of the two paths is shown in Table 1. For our example, value reflects the priority points of the features as the weighted sum of benefits to the end user when implemented, and penalties incurred by the end user if postponed [15].

Cost is the combination of the cost to implement the architectural elements selected to be added in a current release, based on an estimate of the effort, plus the cost to rework pre-existing elements.

Table 2 shows the breakout of the individual costs summarized in Table 1. Rework cost is incurred when new elements are added to the system during this release, and one or more of the pre-existing elements must be modified to accommodate the new ones. This includes elements that can be identified with their direct dependencies on the new elements as well as those with indirect dependencies represented by the change propagation metric.

Each path issues four releases of the system over the course of development. Table 3 shows the allocation of features and architectural elements to releases in each development path of the DRNEP.

Fig. 5 depicts the data from the Tables as the value of features delivered over total effort for each of the two paths over four releases.



PATH 1 DELIVER SOON

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Cost
Controller	1	X																	16
I2SimAdapter	2	X	X																3
MTAdapter	3	X		X															3
MT_I2Sim_Trans	4		X	X	X														8
EPANetAdapter	5	X				X													3
Epa_I2Sim_Trans	6		X			X	X												8
Epa_MT_Trans	7			X		X		X											8
BrahmsAdapter	8	X							X										3
Brahms_I2Sim_Trans	9		X						X	X									8
Brahms_MT_Trans	10			X					X		X								8
Brahms_EPANet_Trans	11					X			X			X							8
FloodAdapter	12	X											X						3
Flood_I2Sim_Trans	13		X										X	X					8
Flood_MT_Trans	14			X									X		X				8
Flood_EPANet_Trans	15					X							X			X			8
Flood_Brahms_Trans	16								X				X				X		8
Data Persistence	17	X																X	5
															COST				116

PATH 2 REDUCE REWORK

		1	2	3	4	5	6	7	8	9	10	11	12	13	Cost
I2SimService	1	X													7
MTService	2		X												9
EPAService	3			X											9
BrahmsService	4				X										9
FloodService	5					X									9
Web	6						X								3
WASClient	7	X	X	X	X	X	X	X							8
Controller	8								X						8
Service	9	X	X	X	X	X	X		X	X					6
ServiceBusiness	10								X	X	X				5
SimulatorServiceClient	11										X	X			8
Entities	12										X		X		11
Common	13	X	X	X	X	X	X	X	X	X	X		X	X	2
											COST				94

Figure 4. Dependency analysis of the two DRNEP paths

TABLE I. COMPARISON OF TWO PATHS FROM THE PERSPECTIVE OF TECHNICAL DEBT

		Release 1	Release 2	Release 3	Release 4
Path #1	Cumulative value	36	81	135	197
	% of total value	18%	41%	68%	100%
	Cost (Ci + Cr)	35	64	101	145
	% of total implementation cost	37%	68%	108%	155%
Path #2	Cumulative value	36	81	135	197
	% of total value	18%	41%	68%	100%
	Cost (Ci + Cr)	67	76	85	94
	% of total implementation cost	71%	81%	90%	100%

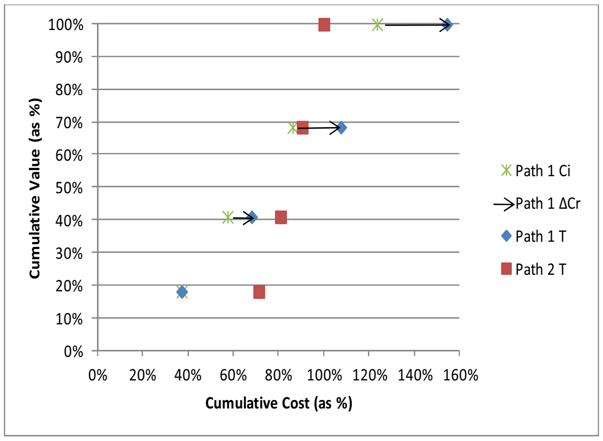


Figure 5. Value of features delivered over total effort

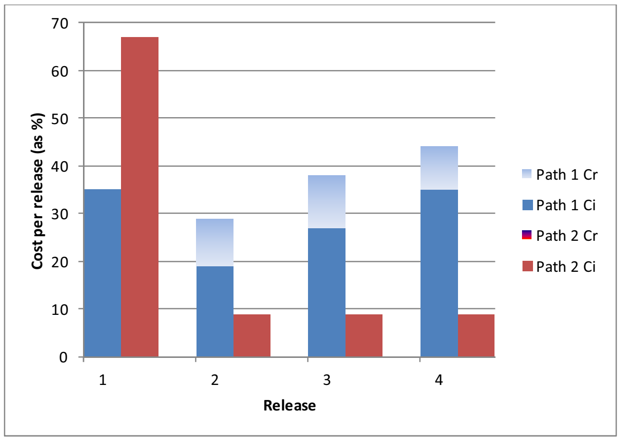


Figure 6. Release Cadence

While the delivered features have the same value at each release for the two approaches, there are certain trade-offs taken in each. In path #1, the first release is out of the door with less cost, also an indicator of quicker delivery. The impact of this is later realized with increased rework cost that starts with release 3. In path #2, the initial release takes most of the implementation cost; after the first release the delta implementation cost is small and the features can be developed with less cost starting with release 2. Until release 2, path #1 delivers more value for the cost; starting with release 3 there is a switch.

Comparing the two paths provides some insight into the challenge of balancing rapid deployment and long-term value. Path #1, which is aimed at delivering value soon to the user, starts creating a debt right after the first release and incurs 18% more cost than path #2 on release 3. For release 4, the additional cost jumps to 55% and rises higher and higher with each future increase in functionality.

Figs. 5 and 6 show the impact of rework on each release. The architectural debt accumulates over time. If this debt is not repaid by re-architecting the system, new services added to the system incur increasingly higher implementation costs. In the case of DRNEP, where the system will continue to grow with additional services, the increasingly high cumulative impact of rework is significant. While the delta cost can initially be neglected, it reaches a significant amount by release 4. The increasing rework cost also demonstrates the interest of the unpaid debt piling up as time progresses. Table 2 provides the details of how implementation cost on each release significantly decreases on path #2 once a basic needed infrastructure effort takes place during release 1, representing most of the implementation cost.

The release cadence is shown in Fig. 6. Given the debt that is accumulating in path #1, we see the effort increasing over time after the initial release, since the number of interconnections among the simulators is exponential. Path #2 has significant effort in the first release and then settles into a regular rhythm of releasing features on top of the infrastructure that's been put into place.

Revisiting the architectural diagrams of the two approaches in Figs. 1 and 3 reveals that the pattern demonstrated with the rework analysis is also observable in the implementation times of the architectural elements. The majority of the architectural elements in path #2 are implemented during the initial release, while in path #1 the system emerges gradually through the releases. The resulting architectures differ as well, reflecting the contrasting approaches taken.

TABLE II. RELEASE REWORK COSTS

Path #1	Implementation Cost	Propagation Cost	Rework Cost
Release 4	35	0.35	9
Release 3	27	0.33	11
Release 2	19	0.42	10
Release 1	35	0.44	0
Total Ci	116		
Path #2	**Implementation Cost**	**Propagation Cost**	**Rework Cost**
Release 4	9	0.56	0
Release 3	9	0.56	0
Release 2	9	0.56	0
Release 1	67	0.35	0
Total Ci	94		

It is critical to note that the point of creating a path comparison is not to find the absolute truth, but to make the actual trade-offs explicit. Path #1 is representative of the case when the system is new and in a business value exploration phase. In such a stage it is critical to get some functionality out of the door and to test it in the field. As the use of the system proves valuable, the system begins to grow. In our example, following the initial strategy created uncontrolled implementation costs, requiring a significant re-architecting effort, and was suboptimal for the long-term sustainment of the system. Performing such analysis on potential rework as

TABLE III. ALLOCATION OF ARCHITECTURAL ELEMENTS TO RELEASES IN EACH PATH IN DRNEP

Release	Feature implemented	Path #1	Path #2
1	US01 Two Simulators Introducing Simulator I2 and MT *Cumulative Value 18%*	Architecture Elements: 01 Controller, 02 I2SimAdapter, 03 MTAdapter, 04 MT_I2Sim_Trans, 17 Data Persistence *Cumulative Cost 37%*	Architecture Elements: 01 I2SimService, 02 MTService 06 Web, 07WASClient, 08 Controller, 09 Service, 10 ServiceBusiness, 11 SimulatorServiceClient, 12 Entities, 13 Common. *Cumulative Cost 71%*
2	US02 Three Simulators Adding Simulator EPA *Cumulative Value 41%*	Rework of the previous elements: 01 Controller, 02 I2SimAdapter, 03 MTAdapter. New elements: 05 EPANetAdapter, 06 Epa_I2Sim_Trans, 07 Epa_MT_Trans. *Cumulative Cost 68%*	New elements: 03 EPAService *Cumulative Cost 81%*
3	US03 Four Simulators Adding Simulator Brahms *Cumulative Value 68%*	Rework of the previous elements: 01 Controller, 02 I2SimAdapter, 03 MTAdapter, 05 EPANetAdapter. New elements: 08 BrahmsAdapter, 09 Brahms_I2Sim_Trans, 10 Brahms_MT_Trans, 11 Brahms_EPANet_Trans. *Cumulative Cost 108%*	New elements: 04 BrahmsService *Cumulative Cost 90%*
4	US04 Five Simulators Adding Simulator Flood *Cumulative Value 100%*	Rework of the previous elements: 01 Controller, 02 I2SimAdapter, 03 MTAdapter, 05 EPANetAdapter, 08 BrahmsAdapter. New elements: 12 FloodAdapter, 13 Flood_I2Sim_Trans, 14 Flood_MT_Trans, 15 Flood_EPANet_Trans, 16 Flood_Brahms_Trans *Cumulative Cost 155%*	New elements: 05 FloodSimServ. *Cumulative Cost 100%*

the system grows can enable timely decisions about when to start re-architecting the system or when to pay back the interest on the borrowed time of the earlier design decisions. Our approach offers considerations for making rework explicit at each release at the architecture level.

IV. RELATED WORK

We review the related work based on the following three aspects: (1) foundations of technical debt and approaches to managing it, (2) metrics to guide the refactoring and re-architecting process, and (3) tool support that can provide increased visibility, agility, and timely information for managing technical debt effectively.

A. Foundations on technical debt

Theoretical foundations for identifying and managing technical debt have focused mostly on the unintentional debt that accumulates as a consequence of unexpected environment changes, suboptimal engineering practices, or simply unmanaged growth of system size over time. While Cunningham used the metaphor in reference to coding practices, today it is applied more broadly across the software development project lifecycle and may include structural, requirements, testing, or documentation debt [16]. There is a key difference between debt that results from employing bad engineering practices and debt that is incurred through intentional decision-making in pursuit of a strategic goal. Martin Fowler details this distinction by using four quadrants to describe technical debt.

TABLE IV. FOWLER'S TECHNICAL DEBT TAXONOMY [4]

	Reckless	Prudent
Deliberate	*We don't have time for design.*	*We must ship now and deal with the consequences.*
Inadvertent	*What's layering?*	*Now we know how we should have done it.*

Technical debt resonates with maintenance and evolution challenges when it needs to be repaid, especially when repayment involves refactoring and re-architecting. Lehman [17] observes that for systems to remain useful they must change, and that change will increase their complexity, leading to software decay if refactoring is not done as needed. (Parnas [18] calls this phenomenon "software aging," reflecting the failure of a product owner to modify software to meet changing needs.) Lehman's observations about system evolution are currently applicable to projects that follow agile software development approaches [19]. While his laws of evolution provide insight into the inevitability and necessity of re-architecting (and the potential of high debt accumulation), work in this area has not addressed abstraction between code and architecture beyond the application of pattern-based approaches [20]. Increased understanding of architectural debt captured with rework can help in addressing this abstraction.

An empirical study conducted with architects at IBM concluded that the ability to assess debt does matter. Yet there exist significant and widespread gaps in the demonstration of this ability. The interviews found the following experiences to be common: induced and unintentional debts created significant challenges, decisions were managed in an ad-hoc manner, and stakeholders lacked effective ways to communicate and reason about debt [21].

B. *Metrics to guide refactoring and re-architecting process*

While there has been significant progress regarding code quality, existing metrics for providing visibility into overall system quality are insufficient and unproven, especially for providing architecture-level analysis. Code-level refactoring techniques do not scale effectively to support decision-making at the system architecture level.

Most approaches to debt focus on defect detection and avoidance, rather than a strategic management of key infrastructure decisions, especially in the context of architecture. Concepts like “code smells” and “spaghetti code” have been used to address code-level technical debt [22]. Clone and defect detection that infers change patterns [23] are among concerns relevant to code-level analysis. Debt accumulation due to such defects is often reckless and inadvertent, as opposed to prudent, deliberate, and strategic.

Refactoring techniques [22] [24] address paying back technical debt through local changes to the code base. Work on defect analysis and software maintainability focus on metrics-based analysis to uncover such debt. For example, static analysis tools provide measurement of duplicate code, cyclomatic complexity, or the use of god classes of the code base to provide insight into the potential debt the system might have already incurred. Such analysis aims to focus the refactoring to improve overall code quality using these metrics [25] [23].

There are two drawbacks with these approaches. First, defect analysis, code-quality measurement, and related work look at analysis of code artifacts after the fact [26], when the system is delivered or close to being delivered. While such analysis provides insights about reducing defects over time, it provides no guidance for adjusting a course of action as the system is being developed, or for recognizing debt as it accumulates rather than being surprised by it later. Secondly, while it is often possible to refactor for small changes, it is quite difficult to claim legitimately that the system can be refactored to include a key design concern after the fact, without significant redesign.

C. *Tool support*

A little technical debt may not be a problem, but it becomes a problem when there is “too much” debt. This implies that there must be some rules about what “too much” debt looks like, that is, there must be acceptability thresholds. Static code analysis and plug-ins for supporting technical debt analysis have started to gain attention in the tool space for their promise to provide insights into thresholds of debt. While these tools currently focus on code analysis and provide an overall uniform debt analysis on all code quality drawbacks found (as discussed in the previous section), their rising attraction is due to the promise of automating the process and improving the visibility of the system during development.

For example, an existing debt visualization plug-in by Sonar demonstrates how to monitor coding rules (duplicate code and cycles) violations and provides measures using an estimated amount of person-effort that it would take to correct such violations [27]. While Sonar focuses on static code, analysis tools such as Lattix and SonarGraph also provide visualization assistance of the dependencies. Using SonarGraph, it is possible to organize elements such that one can define rules among such elements and monitor their violations. Based on such violations, again along with code cycles, SonarGraph calculates a structural debt index, SDI [28]. While not offering an explicit debt calculator, Lattix enables a dependency view, which can be used to assist focusing on tasks and avoiding costly decisions [29].

The ability to elicit and improve the visibility of the state of the project is an area of increased research and practical interest, both from a project management perspective and as a system quality perspective. Collective dashboard and visualization approaches assist managing technical debt in this regard. Recent approaches also include visualization techniques that use software maps structured according to the modularity of the system. Complex files (as indicated by their McCabe complexity measure) are highlighted in 3D and by color on the maps. Some challenges in visualization techniques involve integrating time and making the technique fully interactive. The ultimate goal is to provide early warnings to detect costs and risks (e.g., “watch out for this class, it might be growing too big”) [30].

V. Discussion

The algorithm for rework is directional in nature and represents an initial attempt to formalize the impact of architectural dependencies upon effort to address technical debt. The cost of each architectural element, the number of dependencies impacted by each architectural change, and the overall change propagation metric of the system may all be seen as proxies for complexity, which is assumed to affect the cost of change. The relative weighting and relationship between these factors, however, is a subject for future research efforts. Therefore, within the context of our analysis, rework cost is interpreted as a relative rather than an absolute value, used to compare alternative paths and to provide insight into the improvement or degradation of architectural quality across releases within a given path.

The Achilles’ heel of our approach is its reliance on estimates, and in particular estimates for rework effort. But this is a much wider issue in software engineering that we are not attempting to resolve here. Providing such estimates is best done by the people directly involved, based on experience with previous work, on the same system in prior iterations or some other system of the same nature. This is how the numbers were derived in Table 1 and Table 2. The cost values in Figs. 5 and 6 were actual values.

Another potential weakness is the change propagation metric (Pc) defined by MacCormack [10]. This metric is probably a bit too crude: it does not reflect the "strength" of a dependency between modules, nor the likelihood that modules are affected by successive evolutions of the system; the DSM we use in Fig. 4, for example, has only a "depend or not" indicator. We are currently evaluating more sophisticated alternatives to the change propagation metric that take into account these two factors, and that possess a higher predictive power of the potential impact of future changes across the system.

Change propagation is a modularity metric. Used alone it suffers from drawbacks, such as treating the existence of dependencies without any emphasis on their weight or importance, assuming that a change will propagate when there is a dependency, and treating all kinds of dependencies as equal. However, such metrics together with a visual representation, such as the DSMs, demonstrates the potential power of the approach and the justification for us to continue investigating whether (1) such metrics provide insight in terms of measuring the impact of rework, when we look at module structures of systems, and (2) such metrics can be augmented to also account for runtime behavior and the impact of rework and technical debt.

Questions arise, such as "Do such metrics provide absolute or relative values across systems? Do they provide insight for triggering re-architecting activities, that is, for paying back technical debt? Can they be applied at the architecture level effectively to reduce the complexity of required continued empirical analysis?" To date most of such analysis has been conducted in the context of open source systems [31], which provides one validation that it can guide the development effort in reduction of technical debt.

VI. Conclusion

In this paper, we argue that a focus on architecture and architecture-related technical debt can assist in optimizing releases for agility. As our analysis of the two development efforts on the DRNEP system demonstrates, a focus on the immediacy of the short-term value of features to be delivered must be widened before we can consider their consequences in ensuring long-term success. In the DRNEP case study, the two paths are normalized based on the priority of the features they deliver at each release, but the choice and the cost of the structural elements implementing the features diverge. At release 3 the need to pay back the technical debt resulting from the speedy delivery choices of the previous two releases is already apparent. Actively monitoring and making these decisions visible could have resulted in a re-architecting effort earlier that could still deliver a lower cost system for the value.

The essence of our approach is thus: the value of the delivered features and the impact of cost to be incurred must be taken into account in decision-making related to delivering a product. Making the architectural debt visible provides the necessary information for making informed decisions for managing the potential impact of rework over time.

Gaining visibility into system development requires a measurement-focused mindset at the system architecture level and the willingness to adjust courses of action during development. We demonstrated calculating rework cost based on the architecture rather than a focus on static code analysis. Calculating the rework cost is based on detecting the changing dependencies of the system that create an interest payment, which we accounted for with the change propagation metric. We used this rework cost in combination with calculating the value of the features delivered in the case study, to demonstrate how total ownership cost of system development can be effectively managed.

We posit that rather than viewing technical debt as a reflection of low code quality in retrospect, it's possible to leverage the notion of debt to manage system delivery. Deliberately borrowing time enables faster delivery and quicker value. The emerging system architecture must be visible to the development team to allow for calculating the accumulating potential rework cost as the system evolves. The architecture needs to be monitored over the course of development to support analysis on how debt is building up with interest and when to pay back the debt, also with consideration for how the cost and benefit trade off from the business perspective.

Our future directions in this area include the following.

- *Feature and architectural element slicing.*

Story slicing in agile development involves determining the simplest thing the user can do that is of value, and how that function cuts across the system. Slicing cross-cutting quality attribute requirements, however, is more challenging than slicing functional stories because quality attribute requirements typically support multiple functionality, whereas realizing the value might take multiple iterations. Taking on architectural debt on particular slices might speed realizing the value.

- *Extensions to the economic model.*

We account for debt at the time it is paid in rework cost. For prudent and deliberate debt we plan to extend the economic model to account for the future cost of paying back debt, making the debt visible, and providing information on the consequences of payback or carrying the debt. We also plan to investigate incorporating uncertainty in the economic framework.

- *Managing debt.*

This paper's contribution to the area of technical debt management is its discussion of analysis and visibility of dependencies and their implications for technical debt. Next steps in our research will focus on moving from monitoring to more actively and strategically incorporating technical debt into the architectural design process. DRNEP is an ongoing and long-term project conducted with a consortium of researchers. We will be able to collect more information about the trade-offs to be made along the way, including actual numbers for implementation or rework costs.

Our experience has shown us that organizations have started building into their software development approaches active strategies about how debt will be elicited and kept active as needed. To date these approaches are mostly focused on exposing the fact that particular suboptimal decisions were made, in order to at least make sure that the decision is captured; however, payback and monitoring are still not common practice and techniques for both are lacking. It is this gap that our ongoing research aims to fill, contributing to improving the practice of software economics and software architecture practices with a quantifiable basis.

ACKNOWLEDGEMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

REFERENCES

[1] W. Cunningham, “The WyCash portfolio management system,” Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA ’92) (Addendum), ACM Press, Apr. 1993, pp. 29–30, doi: 10.1145/157709.157715.

[2] C. Sterling, Managing Software Debt: Building for Inevitable Change. Boston: Addison-Wesley Professional, 2010.

[3] S. McConnell, "Technical Debt," Software Best Practices, Nov. 2007. Available: http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (Accessed: Mar. 2012).

[4] M. Fowler, Technical Debt Quadrant, Oct. 2009. Available: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html (Accessed: Mar. 2012).

[5] S. McConnell, Managing Technical Debt [Webinar], Sep. 2011. Available: http://www.youtube.com/watch?v=lEKvzEyNtbk (Accessed: Mar. 2012).

[6] M. A. Gonzalez, J. R. Marti, and P. Kruchten, “A canonical data model for simulator interoperation in a collaborative system for disaster response simulation,” Can. Conf. Electrical and Computer Engineering (CCECE 2011), IEEE Press, May 2011, pp. 1519– 1522, doi: 10.1109/CCECE.2011.6030719.

[7] M. Cohn, Agile Estimating and Planning. Englewood Cliffs, NJ: Prentice Hall, 2005.

[8] J. Highsmith, Agile Project Management: Creating Innovative Products. Boston: Addison Wesley Professional, 2004.

[9] N. Brown, R. Nord, I. Ozkaya, and M. Pais, “Analysis and management of architectural dependencies in iterative release planning,” 9th Work. IEEE/IFIP Conf. Software Architecture (WICSA 2011), IEEE Press, Jun. 2011, pp. 103–112, doi: 10.1109/WICSA.2011.22.

[10] A. MacCormack, J. Rusnak, and C. Baldwin, Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis (Version 3.0). Boston: Harvard Business School, 2008.

[11] P. Kruchten, C. Woo, K. Monu, and M. Sotoodeh, “A conceptual model of disasters encompassing multiple stakeholder domains,” Int. J. Emergency Manage., vol. 5, pp. 25–56, 2008.

[12] Infrastructure Interdependencies Simulation Team, Infrastructure Interdependencies Simulator (I2Sim), 2007. Available: http://www.i2sim.ca (Accessed: Mar. 2012).

[13] J. R. Marti, J. A. Hollman, C. Ventura, and J. Jaskevitch, “Dynamic recovery of critical infrastructures: Real-time temporal coordination,” Int. J. Critical Infrastructures, vol. 4, pp. 17–31, 2008.

[14] M. Danilovic and T. Brown, “Managing complex product development projects with design structure matrices and domain mapping matrices,” Int. J. Project Manage., vol. 25, pp. 300–314, Apr. 2007.

[15] K. E. Wiegers, Software Requirements, 2nd ed. Redmond, WA: Microsoft Press, 2003.

[16] B. Barton, et al., How to Settle Your Technical Debt: A Manager’s Guide. Arlington, MA: Cutter Consortium, 2010.

[17] M. M. Lehman, Program Evolution: Processes of Software Change. San Diego, CA: Academic Press Professional, 1985.

[18] D. L. Parnas, “Software aging,” Proc. 16th Int. Conf. Software Engineering, ACM Press, May 1994, pp. 279–287.

[19] R. Sindhgatta, N. C. Narendra, and B. Sengupta, “Software evolution in agile development: A case study,” Companion to the Proc. ACM Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications (SPLASH ’10), ACM Press, Oct. 2010, pp. 105–114, doi: 10.1145/1869542.1869560.

[20] C. J. Neill and P. A. Laplante, “Paying down design debt with strategic refactoring,” Computer, vol. 39, no. 12, pp. 131–134, Dec. 2006.

[21] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, “An enterprise perspective on technical debt,” Proc. 2nd Work. Managing Technical Debt (MTD ’11), ACM Press, May 2011, pp. 35–38, doi: 10.1145/1985362.1985371.

[22] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley Professional, 1999.

[23] M. Kim and D. Notkin, “Discovering and representing systematic code changes,” Proc. 31st Int. Conf. Software Engineering, May 2009, pp. 309–319, doi: 10.1109/ICSE.2009.5070531.

[24] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna, “From retrospect to prospect: Assessing modularity and stability from software architecture,” Proc. Joint 8th Working IEEE/IFIP Conf. Software Architecture and 3rd Eur. Conf. Software Architecture (WICSA/ECSA), IEEE Press, Sep. 2009, pp. 269–272, doi: 10.1109/WICSA.2009.5290817.

[25] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw, “Building empirical support for automated code smell detection,” Proc. 2010 ACM-IEEE Int. Symp. Empirical Software Engineering and Measurement, ACM Press, Sep. 2010, pp. 8:1–8:10, doi: 10.1145/1852786.1852797.

[26] R. P. L. Buse and T. Zimmermann, “Analytics for software development,” Proc. FSE/SDP Work. Future of Software Engineering Research (FoSER ’10), ACM Press, Nov. 2010, pp. 77–80, doi: 10.1145/1882362.1882379.

[27] O. Gaudin, “Evaluate your technical debt with Sonar,” Sonar, Jun. 2009. Available: http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar (Accessed: Apr. 2012).

[28] S. Penchikala, “Architecture analysis tool SonarJ 6.0 supports structural debt index and quality model,” InfoQ, Aug. 2010. Available: http://www.infoq.com/news/2010/08/sonarj-6.0 (Accessed: Mar. 2012).

[29] C. Hinsman, N. Sangal, and J. Stafford, “Achieving agility through architecture visibility,” Proc. 5th Int. Conf. Quality of Software Architectures: Architectures for Adaptive Software Systems (QoSA ’09), ACM Press, Jun. 2009, pp. 116–129, doi: 10.1007/978-3-642-02351-4_8.

[30] J. Bohnet and J. Dollner, “Monitoring code quality and development activity by software maps,” Proc. 2nd Work. Managing Technical Debt (MTD ’11), ACM Press, May 2011, pp. 9–16, doi: 10.1145/1985362.1985365.

[31] R. Milev, S. Muegge, and M. Weiss, “Design evolution of an open source project using an improved modularity metric,” Proc. Int. Conf. Open Source Systems (OSS ’09), Springer, Jun. 2009, pp. 20–33, doi 10.1007/978-3-642-02032-2_4.