Analysis and Management of Architectural Dependencies in Iterative Release Planning

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

Manuel Pais
Master of Software Engineering Program
Carnegie Mellon University
Pittsburgh, PA USA
manuelpais@gmail.com

***Abstract*— Within any incremental development paradigm, there exists a tension between the desire to deliver value to the customer early and the desire to reduce cost by avoiding architectural refactoring in subsequent releases. What is lacking, however, is quantifiable guidance that highlights the potential benefits and risks of choosing one or the other of these alternatives or a blend of both strategies. In this paper, we assert that the ability to quantify architecture quality with measurable criteria provides engineering guidance for iterative release planning. We demonstrate the use of propagation cost as a proxy for architectural health with dependency analysis of design structure and domain mapping matrices as a quantifiable basis for iteration planning.**

Keywords: architecture quality; dependency management; design structure matrix; propagation cost

I. INTRODUCTION

Within any iterative and incremental development paradigm, there exists a tension between the desire to deliver value to the customer early and the desire to reduce cost by avoiding architectural refactoring in subsequent releases [1]. The choice between these two competing interests is situational. In certain contexts, early delivery might be the correct choice, to enable for example, the release of critically needed capabilities or to gain market exposure and feedback. In other contexts, however, delayed release in the interest of reducing later rework might be the choice that better aligns with project and organizational drivers and concerns. What is lacking, however, is quantifiable guidance that highlights the potential benefits and risks of choosing one or the other of these alternatives (or perhaps a blend of both strategies). In this paper, we seek to provide such guidance using dependency mapping and analytic techniques that provide insight into the cost and value implications of specific iterative delivery strategies.

In iterative release planning, a range of dependencies[1] must be taken into consideration:

1. Dependencies between capabilities, including both functional and non-functional requirements
2. Dependencies between capabilities and architectural elements
3. Dependencies between architectural elements

Understanding the dependencies between capabilities allows for optimization of development activities within a given release and also ensures that a coherent and useful feature set is released to the end user [2]. Understanding the dependencies between capabilities and architectural elements allows for a staged implementation of the architecture in support of the delivery of customer value. Understanding the dependencies between architectural elements also supports the staged delivery of customer value. In addition, an analysis of dependencies between architectural elements provides insight into potential downstream architectural refactoring costs that may be incurred as a result of choosing to incrementally develop and release the architectural infrastructure.

Depending upon the organization and life cycle adopted, analysis of these dependency types may be seen as the provenance of distinct stakeholder roles, performed by different people, who may even be working in separate organizations. This can result in fragmented communication and sub-optimal solutions and delivery strategies. In our paper, we illustrate the use of a technique that can be used to integrate the analysis of these three types of dependency relationships.

We posit that the ability to quantify architecture quality with measurable criteria provides engineering guidance for iterative release planning. We can improve the visibility of the quality of the architecture by providing quantifiable quality models of the architecture module structure during system development. These models can be applied earlier in the life cycle using the architecture as an analytic model (as opposed to code). We are starting with models of propagation cost as a proxy for architectural health. In this paper, we present the relevance of the propagation cost model and demonstrate its use.

We conducted a study with the goal of identifying whether there are distinct return-on-investment outcomes of development paths if contrasting business goals are at stake and whether it is possible to monitor the outcomes at each delivery release. The two contrasting goals we studied are i) maximizing value for the end user; and ii) minimizing

[1] A dependency exists between two elements if an element requires another element for its specification (semantics) or implementation (structure).

In: Proceedings of the Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA) 2011, June 20-24, 2011, Boulder, CO, USA.

implementation cost due to rework. We analyzed how propagation cost changed from iteration to iteration as we optimized for these different outcomes.

The structure of the paper is as follows. In section 2, we review dependency management as manifested by design structure matrixes. In section 3, we review our development path analysis approach to iteration planning. In section 4, we present the details of our study conducted on the Management Station Lite system [3] and lastly in section 5 we discuss our findings and conclusions.

II. Dependency Management and Quality

In order to reason about alternative development paths for a given project, we must take into account the dependencies between customer requirements (for example sending a command to lower the building temperature requires having knowledge and therefore access to the current temperature). Dependencies between architectural elements are also important, for instance asynchronous updates of the client browser require management of user sessions. Yet another type of dependency occurs between architectural components and the customer requirements they support (either functional or quality related). We review how to capture each of these dependencies using design structure and domain mapping matrices.

A. Design structure matrix (DSM).

A design structure matrix (DSM) provides a simple, compact and visual representation of a system. DSMs to date have not only been used in component-based and architecture decomposition and integration analysis, but also in organization, project, product planning, and management contexts. The use of DSMs in software engineering has mostly focused on understanding design rules and has been increasingly incorporated into reverse engineering and architecting tools [4].

A DSM is a matrix mapping dependencies between items in a given domain. 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 Figure 2, the mark at the intersection of row 3 with column 1 means that element in column 1 (Alarm Notification) depends on element in row 3 (Alarm Engine).

DSMs are single domain square matrices, meaning that relations are defined between instances of the same type (architectural elements in Figure 2). However, dependencies also occur across different domains. Examples include dependencies between development staff technical competencies and the software components to be developed. Another common example is the identification of which software components or modules satisfy which customer requirements. These multi-domain dependencies often cause project delays or even failure when detected too late [5].

B. Domain mapping matrix (DMM)

The term Domain Mapping Matrix (DMM) was coined to refer to rectangular matrices that map the relations between items in two different product development domains (for example task X requires person Y's expertise) [5] [6].

DMMs allow dual-domain analysis by representing dependencies between items in one domain (rows) and items in another domain (columns). The two domains need not have the same number of items, thus the resulting DMM is usually a rectangular matrix. Figure 3 illustrates an example of a DMM applied to the analysis of dependencies between customer requirements and architecture elements.

For DMMs, only the clustering technique has been proved to be applicable and produce meaningful results in a similar way to DSM clustering [5]. The goal of clustering is to find subsets of DSM elements such that the nodes in a DSM cluster are maximally dependent and the clusters are minimally interacting. The main difference is that clusters within a DMM can occur anywhere in the matrix and not just along the diagonal as with a DSM. The clusters will identify areas where the two domains are closely related and/or interdependent.

C. Combining DSM and DMMs

DSM and DMM analysis can be combined to reach deeper conclusions about inter and intra-domain dependencies in a dual-domain context. Bartolomei suggests this can be done in two ways [7].

One approach is to create a DSM for each of the two domains in order to first analyze the intra-domain dependencies. Then create a DMM mapping the inter-domain dependencies. This approach requires an integrated analysis of data from all resulting three matrices in order to arrive at meaningful conclusions. The associated benefits derive mostly from the discussions among involved stakeholders and clarification of the impact of intra-domain dependencies on inter-domain relations, as suggested by Danilovic in a complex multi-project development with a high level of uncertainty [6].

The second approach to analyzing dependencies across multiple domains combines DSM and DMM analysis within a Multiple-Domain Matrix (MDM). In a MDM, all items from both domains appear within a single square matrix [7] both as rows and columns (with the same ordering along both axes). In theory, this approach allows representation of mutual dependencies between items in different domains and possibly the application of the sequencing algorithm used for time-based DSMs. However, there seem to be no case studies available yet to draw upon for empirical confirmation. The environmental parameters extension, EDSM, applied by Sullivan is one attempt to include multiple domains within software analysis; however, that has not been applied to iteration planning [8] [9].

Table I summarized the use of DSM and DMMs within the context of an iterative release planning exercise.

TABLE I. DSM/DMM Identification

	Customer Requirements (CR)	Architectural Elements (AE)
Customer Requirements (CR)	DSM_{CR}	DMM
Architectural Elements (AE)	--	DSM_{AE}

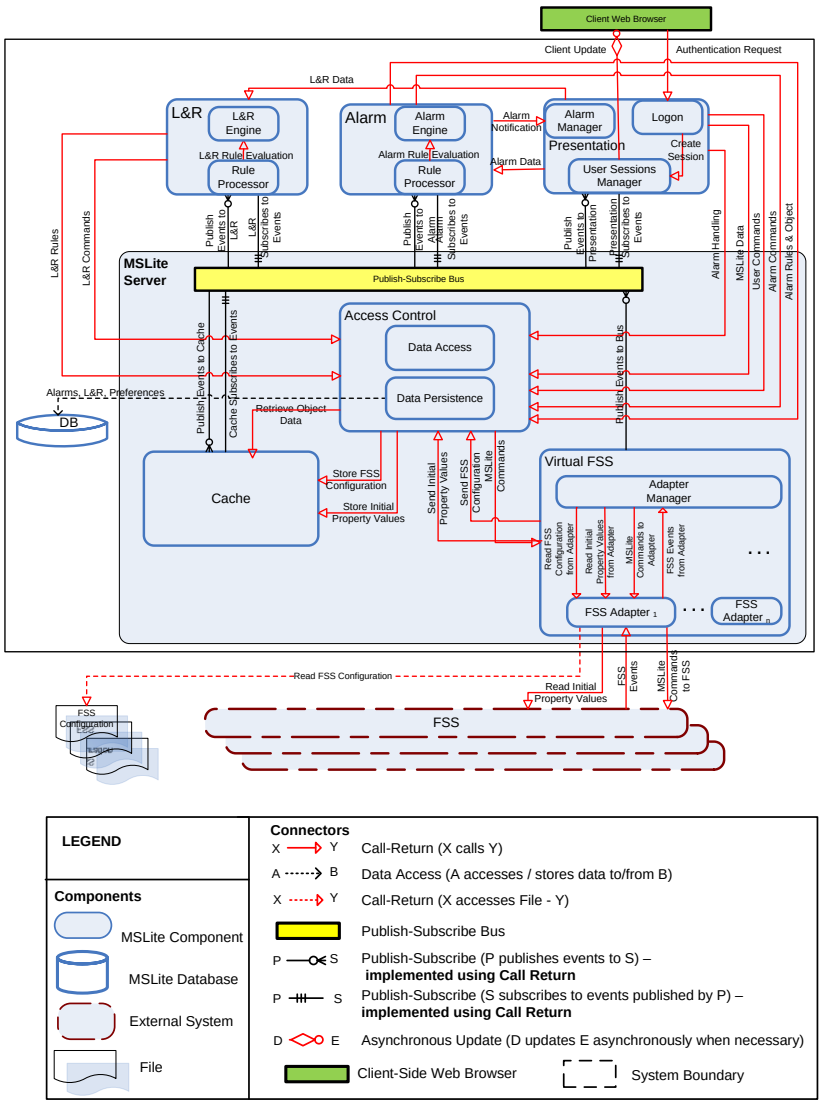

Figure 1. MSLite top-level runtime [10] architecture view

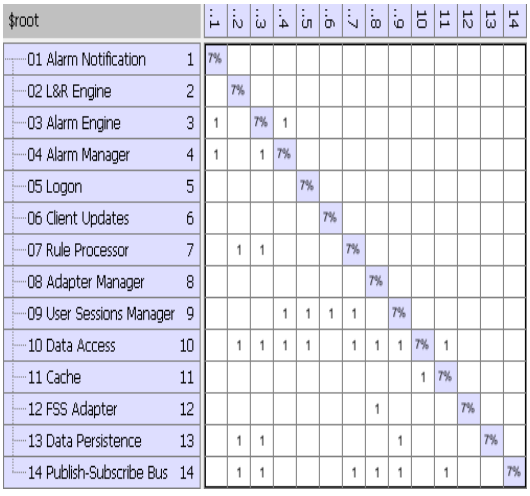

Figure 2. DSM representation of MSLite top-level runtime view

D. *Propagation cost*

Architecture quality and visibility are closely related. Reasoning about quality with a quantifiable model requires certain architectural properties to be represented objectively and in a repeatable manner across systems for the model to work. It is for this reason that we look more closely into using DSM and DMM analysis to provide a representation with support for objective metrics generation. We discuss the propagation cost metric in this context.

The propagation cost measures the percentage of system elements that can be affected, on average, when a change is made to a randomly chosen element [11]. Some existing approaches of using DSM analysis examine “direct” dependencies and provide metrics measuring complexity and [12] decision volatility [9]. Other approaches use a propagation cost metric to take into account “indirect” dependencies and observe correlations between software coupling and the coordination requirements among developers [13]. These approaches take a “shapshot” of the

current system state using the DSM for improving visibility at the architecture level. For the calculation of cost of change, we use the system propagation cost metric that can be derived from the DSM of architecture elements, DSM_{AE}.

Propagation cost, P_C, is calculated as the density of the matrix as represented by the total number of filled cells due to direct or indirect dependencies to the entire matrix size:

$$\frac{}{n^2}$$

According to MacCormack [11], this metric captures the percentage of system elements that can be affected, on average, when a change is made to a randomly chosen element.

The module view of the architecture provides visibility as displayed in the DSM. Starting with the propagation cost calculation our approach uses module as the unit of analysis. Hence, the types of dependencies that we model are module dependencies. We characterize the structure of design by measuring the degree of coupling and propagation cost (examining direct and indirect chains of dependencies).

III. PATH ANALYSIS

Next we describe our approach to development path analysis for iterative release planning. Each release (where a release might be an internal engineering release or an external release to the customer) in the path is described by the following attributes:

- Sequence number (the release order in the path)
- Customer requirements delivered
- Architectural elements implemented
- Number of dependencies between architectural elements implemented and other elements that have been implemented in a previous release

We calculate end-user value and total implementation cost of each release, expressed as units. We measure end-user value at each release by adding the value of all customer requirements supported by the implementation of those architectural elements present in the release.

Total cost of release *n*, Tc_n, is the combination of the cost to implement the architectural elements selected to be added in this release, Ic_n, plus the cost to rework pre-existing elements, Rc_n.

$$Tc_n = Ic_n + Rc_n$$

Implementation cost is incurred when new elements are added to the system during this release. Implementation cost, Ic_n, for release *n* is computed as follows:

- Sum the implementation cost of all architectural elements, AE_j implemented in release *n* (and not present in an earlier release).
- The implementation cost is assumed to be given for all individual architectural elements (independent of dependencies).

Rework cost is incurred when new elements are added to the system during this release, and one or more of the pre-existing elements have to 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 propagation cost. Rework cost, Rc_n, for release *n* is computed as follows:

- Compute the rework cost associated with each new architectural element AE_k implemented in release *n*. For each pre-existing AE_j with dependencies on AE_k, multiply the number of dependencies that AE_j has on AE_k times the implementation cost of AE_j times the propagation cost of release n-1.
- Sum the change costs for all new architectural elements implemented in the release.

The algorithm for rework is directional in nature and represents an initial effort to formalize the impact of architectural dependencies upon rework effort. The cost of each architectural element, the number of dependencies impacted by each architectural change and the overall propagation cost of the system may all be seen as proxies for complexity, which is assumed to affect the cost of change. In most cases the number of dependencies impacted by architectural change is "1" but when there is more than 1 interface/dependency the cost will increase. The relative weighting and relationship between these factors, however, is a subject of 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.

IV. STUDY RESULTS

We conducted an exploratory analysis to quantify the cost and value outcomes of alternate release strategies using dependency analysis with propagation cost. For the purposes of our example, value reflects the priority points of the capabilities. We picked the Management Station Lite (MSLite) [3] [10] system for the study, a system with which we have previous experience and access to the architecture artifacts.

MSLite is a hardware-based field system for controlling a building's internal functions, such as heating, ventilation, air conditioning, access, and safety that automatically monitors and control the building's internal functions. Figure 1 shows the component and connecter (C&C) view of the MSLite system and the resulting DSM view (Figure 2). In our study we are assuming the module structure is similar to the C&C view and look at dependencies documented in the relationships among architectural elements: call-return, data access, events. The system users are facilities managers, and the system broadly performs the following functions:

- Manage a network of hardware-based field systems used for controlling building functions.
- Issue commands to configure the field systems. Define rules on the basis of property values of field systems that trigger reactions and issue commands to reset these property values.
- Trigger alarms notifying appropriate users of life-critical situations.

First we expressed the requirements from MSLite as user stories and acceptance test cases. Requirements were prioritized according to their relative benefit to the end user when implemented and the penalty incurred by the end user if postponed. Furthermore, we assigned a user value to each user story (US) and architecturally significant acceptance test case (ATC) as the weighted sum of benefits and penalties [14], as seen in Table II. The dependency analysis was used to determine precedence in the implementation of capabilities. In the absence of a "Product Owner" we made assumptions concerning the acceptability of splitting the delivery of functional and non-functional capabilities across releases. In future applications, we plan to expand the concept of "minimum marketable features" [2] to the broader construct of "minimum releasable capabilities" to ensure that inter-dependent functional and quality attribute (also known as non-functional) requirements are released in batches that deliver acceptable end-user value.

The components outside the MSLite Server implement the core user functionality, namely monitoring building facilities and issuing commands to change their properties (for example lower the building temperature). Detecting alarm conditions and automating rules for property changes are other important capabilities of the system.

The main purpose of the components of the MSLite Server is to provide support for the quality requirements. For instance one of the main system goals is to support multiple field systems. The Virtual FSS component handles all the field systems (FSS) descriptions and creates appropriate events for the other components in the system to listen. Other components such as Cache and Access Control explicitly support performance and security, respectively.

TABLE II. MSLITE FUNCTIONAL AND ARCHITECTURALY SIGNIFICANT REQUIREMENTS

	Feature Description	Value
US01	Visualize properties of field objects	27
US02	Change field object properties	25
US03	Add alarm condition	23
US04	Alarm notification & acknowledgement	21
US05	Ignore alarm notification	16
US06	Add logic condition and reaction	19
US07	Alarms and logic/reaction pairs persistence	17
US08	Secure access to system	9
ATC14	Connect similar field system	14
ATC15	Connect field system using different format and interfaces	9
ATC16	Connect field system providing new functionality	13
ATC17	Field object properties update speed	9
ATC18	Alarm notification speed	10
ATC19	No loss of alarm notifications	11
ATC20	Field object properties data access	9
ATC21	Field object properties updated	11
	Total feature value	**243**

The methodology used for the study consisted first of defining the system requirements and system structure using DSM and DMM analysis. Next we defined two alternate strategies or development paths by which to realize the requirements and system structure. The third and final step within our method was to analyze the patterns of value generation and cost incursion that resulted from each of the two development paths.

A. System Definition

We first identified the set of DSM/DMM matrices that contain all the intra and inter-domain dependencies relevant for our system analysis:

- Customer requirements DSM (DSM_{CR})
- Architectural elements DSM (DSM_{AE})
- DMM mapping customer requirements to architectural elements

Each DSM_{CR} matrix cell *c(ij)* denotes that:

- if *i* is an acceptance test case then *i* directly depends on user story *j*, if the test case elaborates on the environment or criteria under which j takes place
- if *i* is a user story, then i directly depends on user story *j*, if the functionality provided by story j is required for the functionality in story i to be delivered

Each DSM_{AE} matrix cell *a(ij)* denotes:

- element j directly depends on element *i*, if some output provided by element *i* is an input to element *j* or if element *j* uses/calls some functionality in element *i* to perform its work

Each DMM element *ca(ij)* denotes that:

- if *i* is an acceptance test case, then *i* directly depends on element *j*, if *j* supports the quality scenario described by *i*
- if *i* is a user story, then *i* directly depends on element *j* if the functionality required for story *i* execution is (partially or fully) implemented in element *j*

Figure 3 shows the resulting DMM of MSLite architecture elements and features.

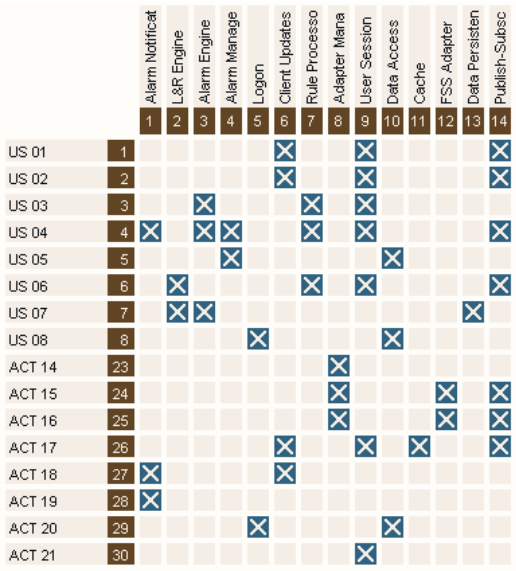

Figure 3. MSLite DMM view

We produced a catalog of relevant architectural elements. Some of the high-level components in Figure 1 were decomposed into sub-components to be able to separate fine-grained architectural elements to include in the development path. For instance Access Control component was sub-divided into Data Access and Data Persistence sub-components. This allowed us to consider each of them separately and assign their implementation to different releases according to the desired business goal (i.e., maximizing early delivery of end user value or minimizing cost due to rework).

The DMM in Figure 3 was created after identifying the dependencies from user stories to architectural elements (via the functional features they support) and from system-level acceptance test cases to architectural elements (via the quality requirements they promote) according to the following two threads of logic:

- User story (requires) → (implemented in) architectural element(s)
- System-level acceptance test case (verifies) → quality requirements → (promoted by) architectural element(s)

B. Definition of Development Paths

We created two development paths with two different goals, i) maximizing early delivery of value for the end user; ii) minimizing implementation cost. We used the value of the features from Table II and the cost of each element from Table III as our basis. Our goal was to see how the two paths would differ when using propagation cost as the basis for understanding the incurred rework costs. Each path consists of a set of software releases where each release implements new architectural elements. In this study both paths had the same defined end point and used the same catalog of architectural elements as building blocks to control for the differences in implementation and to highlight the influence of the orderings in the paths.

TABLE III. IMPLEMENTATION COST OF EACH ARCHITECTURAL ELEMENT

Architectural Element	Implementation Cost
Alarm Notification	4
L&R Engine	3
Alarm Engine	3
Alarm Manager	6
Logon	2
Client Updates	5
Rule Processor	8
Adapter Manager	5
User Sessions Manager	5
Data Access	9
Cache	3
FSS Adapter	6
Data Persistence	6
Publish-Subscribe	9

The use of dependency management techniques such as DSM and DMM provides strong support in the path definition process, allowing us to identify the architectural elements to be implemented and the customer requirements to be delivered in each release for both development paths. Once defined, these two paths served as input for performing path analysis and deciding on the best one for the project given specific customer needs. In other words, within a specific project context how willing is the customer to accept additional rework cost in exchange for the early delivery of valued functionality.

1) Path #1: Maximizing value for the end user.

End-user value optimization is directly derived from the value associated with user stories and system-level acceptance test cases. The ones providing higher value, as summarized in Table II, would be delivered first and therefore the development path is directly determined by the elements supporting the functionality that the user stories require or supporting the system qualities defined in acceptance test cases.

First we identify the top 3 most valuable user stories and/or system acceptance test cases. We then look into the customer requirements DSM_{CR} to retrieve the stories upon which the high value stories depend.

After the complete subset of user stories and/or system level acceptance test cases to deliver is defined, we look at the DMM to identify the architectural elements they depend on. Then we use DSM_{AE} to retrieve any additional architectural elements upon which the initial set of architectural elements depends. The resulting subset of user stories, acceptance test cases, and architectural elements constitutes the first release. The process repeats by picking the next top 3 customer requirements in terms of end-user value and following the same steps described until the next release is obtained. The final step of this iterative process will deal with the 3 (or less) lowest value customer requirements to implement.

2) Path #2: Minimizing implementation cost.

Cost is defined both in terms of the cost to implement an architectural element as well as in terms of the number of architectural elements that will be affected by a change to a single element in the system (rework cost). Therefore to minimize rework cost the architectural elements with fewer dependencies on other elements would be delivered first as they are least likely to require modifications when new elements are introduced. Associated with the different goals of these two paths were different heuristics for release definition. For path #1, each release was conceptualized as an external deliverable to an end user. Release definition was therefore based upon the attainment of a cohesive set of end-user value. For path #2, on the other hand, each release was conceptualized as a verifiable executable deliverable either internally for validation or externally for user adoption.

Table IV lists the features to be implemented at each release of the two paths based on these two different approaches.

C. Analysis of the Paths

Table VI summarizes the outcomes of the two paths. In path #1, the increase in value occurs more rapidly. For instance at release 2 the system provides already more than 50% of the total value to the user. In contrast, in path #2 at release 2 there is still no value accrued.

TABLE IV. ALLOCATION OF STORIES TO RELEASES IN EACH PATH

	Path 1	Path 2
Release 1	US01: Visualize field object properties US02: Change field object properties US03 - Add alarm condition	none
Release 2	US04:Alarm notification & acknowledgement US06:Add logic condition and reaction US07:Alarms and logic/reaction pairs persistence	none
Release 3	US05: Ignore alarm notification ATC14: Connect similar field system ATC 16: Connect field system providing new functionality	ATC14:Connect similar field system ATC 15: Connect field system using different format and interfaces ATC 16: Connect field system providing new functionality ATC 21: Field object properties updated
Release 4	ATC 18: Alarm notification speed ATC 19: No loss of alarm notifications ATC 21: Field object properties updated US 08: Secure access to system	US01: Visualize field object properties US02: Change field object properties US 08 : Secure access to system ATC 17: Field object properties update speed ATC 20: Field object properties data access
Release 5	ATC 15: Connect field system using different format and interfaces ATC 17: Field object properties update speed ATC 20: Field object properties data access	US03: Add alarm condition US04 : Alarm notification & acknowledgement US05: Ignore alarm notification US06 : Add logic condition and reaction US07: Alarms and logic/reaction pairs persistence ATC 18: Alarm notification speed ATC 19: No loss of alarm notifications

This is due to the fact that path #2 leads to building first structural elements that provide the foundation for those quality requirements that cut across the entire system:

- Publish-Subscribe Bus for performance and modifiability in general
- Data Persistence for keeping the state of the system and user preferences
- FSS Adapter for modifiability
- Data Access for security
- Cache for performance

The reason for this variation is that the release identification is based on requirements value for path #1 and on the number of dependencies for path #2. TABLE V shows the summary of the cost of each release for both paths.

TABLE V. COMPARISON OF THE COSTS OF THE TWO PATHS

Path #1	Implementation Cost	Propagation Cost	Rework Cost
Release 5	12	0.35	9.92
Release 4	2	0.31	0
Release 3	11	0.33	0
Release 2	19	0.46	5.72
Release 1	30	0.52	0
Cumulative Implementation Cost	74		
Path #2	**Implementation Cost**	**Propagation Cost**	**Rework Cost**
Release 5	16	0.35	0
Release 4	15	0.37	0
Release 3	10	0.39	0
Release 2	12	0.36	0
Release 1	21	0	0
Cumulative Implementation Cost	74		

TABLE VI. COMPARISON OF THE OVERALL OUTCOME OF THE TWO DEVELOPMENT PATHS

		Release 1	Release 2	Release 3	Release 4	Release 5
Path #1	Cumulative value	75	132	175	216	243
	% of total value	30.86%	54.32%	72.02%	88.89%	100.00%
	Cumulative cost (Ic_n + Rc_n)	30	54.72	65.72	67.72	89.64
	% of total implementation cost	40.54%	71.72%	86.58%	89.28%	118.47%
Path #2	Cumulative value	0	0	47	126	243
	% of total value	0%	0%	19%	52%	100%
	Cumulative cost (Ic_n + Rc_n)	21	33	43	58	74
	% of total implementation cost	28.38%	44.60%	58.11%	78.38%	100%

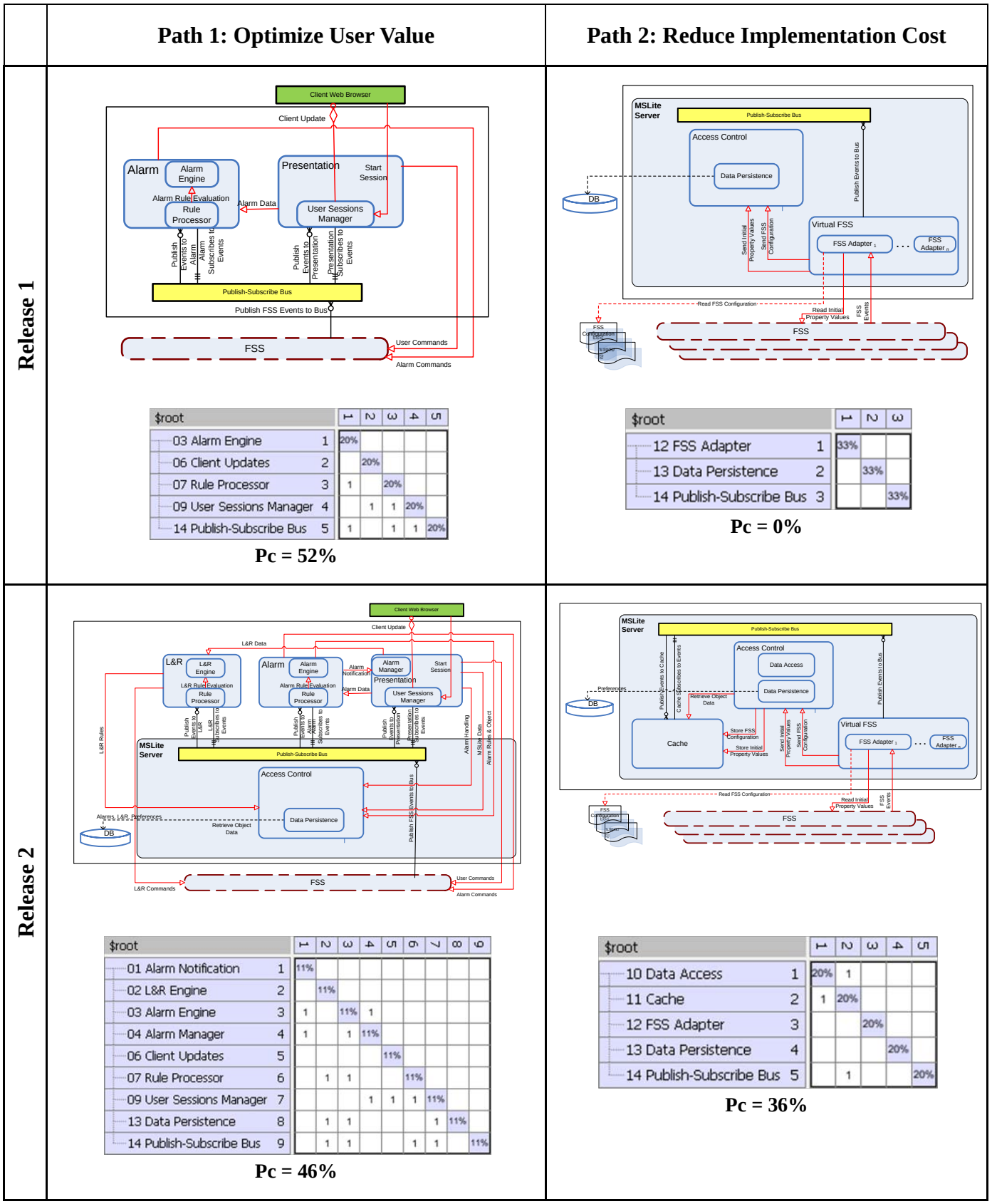

Figure 4. The module view of the emerging architecture at each release along with the DSM and propagation cost for the first two releases

The impact of the cost difference of the two paths is observed in the rework cost. The rework cost associated with path #1 release 2 is 5.72. This cost is incurred because the following elements needed to be reworked:

- Alarm Engine - reworked when Alarm Manager and Data Persistence are implemented.
- User Sessions Manager - reworked when Data Persistence is added.

Figure 4 shows how the different paths manifest themselves with different architectural elements and corresponding propagation costs. Due to space limitations, we show the emerging architecture elements and their DSMs and propagation costs for only the first two releases of both of the paths for comparison purposes. Figure 1 shows the final release which is the same for both paths.

In an agile project, this type of analysis can raise awareness of the cost associated with deciding on an implementation path solely focused on delivering value. If it is not acceptable for the customer, then the team and the customer can decide together on which value/cost trade-offs they are willing to accept.

Let's take as example the architectural elements implemented in release 5 of path #1: Data Access and Cache.

They increase the cost by nearly 20% while providing only about 11% of value. Both elements support quality requirements of medium importance, performance and security.

In the event that the customer is not happy with the overall increase of cost compared to path #2, we can envisage multiple scenarios for reducing the overall cost:

1. Given that Data Access has a high cost (implementation and rework cost) and that the Logon element already provides basic security the customer decides to drop implementation of Data Access.
2. Given that the customer is willing to incur some rework cost and the performance requirement (P1) is (partially) addressed by other architectural elements already in release 4 (Publish-Subscribe Bus, User Sessions Manager, Client Updates) the customer decides to drop implementation of Cache.
3. Given that the customer is not willing to drop any value but still wants to reduce cost, the Data Access element implementation is moved to an earlier release. Although there is an extra cost incurred because Data Access depends on Cache, the reduced cost due to avoiding rework of other elements that depend on Data Access would compensate.

V. DISCUSSION AND CONCLUSIONS

Dependency management has been studied extensively at the level of code artifacts and in the context of system engineering [15] Applying dependency management at the architecture level is beginning to show promising results due to increasing tool support for using DSMs for architectural analysis [16]. As our exploratory analysis demonstrates, metrics, such as propagation cost, can be extracted from the architecture, represented in the form of a DSM. DMM analysis can augment DSM analyses and be used to represent the dependencies between capabilities and architectural elements to further focus the goals of iterative release planning where courses of action may change as the project progresses.

Metrics associated with dependency also provide data for inferring the likely costs of change propagation, especially when dependencies between architectural elements are considered. One such example is discussed in Carriere et al. where the value of re-architecting decisions needed to be understood to determine if the expense to implement them is justified [17].

Making architectural dependencies visible earlier in the development life cycle accompanied by metrics improves communication of architecture quality similar to code quality metrics. Existing models are based on parameters such as the cost of modifying a single element, coupling, cohesion, and life-cycle time of modification [18]. Propagation cost provides insight into degrading architecture quality and identifying the "tipping point" to trigger re-architecting decisions, as it also provides insights for future rework costs.

Metrics alone do not give guidance about how to optimize value over time. We can improve project monitoring by providing quantifiable quality models of the architecture during iteration planning. We are investigating the use of the propagation cost metric to model the impact of degrading architectural quality in order to understand when to invest in improving the architecture as well as to inform tradeoff discussion involving architectural investment versus the delivery of end-user valued capabilities. Our goal is to provide an empirical basis on which to chart and adjust course.

Now that we have a baseline, we plan to investigate incorporating uncertainty in the economic framework and enhancing the approach to model runtime dependencies.

We accounted for rework in the current approach using a simple cash flow model where cost is incurred at the time of the rework. There are economic models that include rework cost that is predicted in future releases. These models become more complex since there are more choices for when to account for the future debt.

The ability to quantify degrading architecture quality and the potential for future rework cost during iterative release planning as each release is being planned is a key aspect of managing strategic technical debt [19]. Managing strategic shortcuts, as captured by the technical debt metaphor, requires better characterization of the economics of architectural violations across a long-term roadmap, rather than enforcing compliance for each release. Our approach facilitates reasoning about the economic implications and perhaps deliberately allowing the architecture quality to degrade in the short term to achieve some greater business goal (all the while continuing to monitor the quality of the architecture and looking for the opportune time to improve).

We accounted for module dependencies in the current approach to support analysis of modifiability type qualities. During this study we were able to account for runtime dependencies indirectly because our model system allowed us to map the component and connector view to the module structure view one-to-one. As an extension of our approach, we are looking at directly modeling runtime dependencies so

we can reason about the quality attributes (e.g., performance schedulability) that they support.

ACKNOWLEDGMENT

The Software Engineering Institute is a federally funded research and development center sponsored by the US Department of Defense. This research was part of an independent research and development project on Communicating the Value of Architecting in Agile Development conducted in collaboration with Philippe Kruchten and Erin Lim. We thank Raghu Sangwan for sharing the architectural artifacts of the MSLite System.

REFERENCES

[1] C. Larman and and V. R. Basili. Iterative and Incremental Development: A Brief History. IEEE Computer, 36(6), 2003: 47-56.

[2] M. Denne and J. Cleland-Huang, Software by Numbers: Low-Risk, High-Return Development, Prentice Hall, 2003.

[3] R. Sangwan, C. Neill, M. Bass, and Z. El Houda, "Integrating a software architecture-centric method into object-oriented analysis and design," Journal of Systems and Software, vol. 81, May. 2008, pp. 727–746.

[4] U. Lindemann, Technical DSM Tutorial, 2009. http://dsmweb.org

[5] M. Danilovic, T. Brown, Managing complex product development projects with design structure matrices and domain mapping matrices, International Journal of Project Management 25, 2007.

[6] M. Danilovic, B. Sandkull, The use of dependence structure matrix and domain mapping matrix in managing uncertainty in multiple project situations, International Journal of Project Management 23, 2005.

[7] J. Bartolomei, M. Cokus, J. Dahlgren, R. de Neufville, D. Maldonado, J. Wilds, Analysis and Application of Design Structure Matrix, Domain Mapping Matrix, and Engineering System Matrix Frameworks, Massachusetts Institute of Technology Engineering Systems Division, June, 2007.

[8] K.J. Sullivan, W.G. Griswold, Y. Cai, and B. Hallen, "The structure and value of modularity in software design," ACM SIGSOFT Software Engineering Notes, New York, NY, USA: ACM, 2001, pp. 99–108.

[9] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From Retrospect to Prospect: Assessing Modularity and Stability From Software Architecture. In Proceedings of the Joint 8th Working IEEE/IFIP Conference on Software Architecture and 3rd European Conference on Software Architecture (WICSA/ECSA), Working Session. September 2009.

[10] R. Sangwan, M. Bass, N. Mullick, D.J. Paulish, and J. Kazmeier, Global Software Development Handbook, Auerbach Publications, 2006.

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

[12] R. S. Sangwan, C. J. Neill: Characterizing essential and incidental complexity in software architectures. WICSA/ECSA 2009: 265-268.

[13] C. Amrit and J. van Hillegersberg. Coordination Implications of Software Coupling in Open Source Projects, P. Ågerfalk et al. (Eds.): OSS 2010, IFIP AICT 319, pp. 314–321, 2010.

[14] K. Wiegers, First Things First: Prioritizing Requriements, Sofwtare Development, 1999.

[15] T. Browning, Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions, IEEE Transactions on Engineering Management, Vol 48, No. 3, 2001.

[16] C. Hinsman, N. Sangal, J. Stafford, Achieving Agility Through Architecture Visibility, in LNCS 5581/2009, Architectures for Adaptive Software Systems, 2009 pp.116-129

[17] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, New York, NY, USA: ACM, 2010, pp. 149–157.

[18] F. Bachmann, L. Bass, and R. Nord, Modifiability Tactics, (CMU/SEI-2007-TR-002). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2007.

[19] 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," Proceedings of the FSE/SDP workshop on Future of software engineering research, New York, NY, USA: ACM, 2010, pp. 47–52.