13TH INTERNATIONAL DEPENDENCY AND STRUCTURE MODELLING CONFERENCE, DSM'11
CAMBRIDGE, MASSACHUSETTS, USA, SEPTEMBER 14 – 15, 2011

MODELING ARCHITECTURAL DEPENDENCIES TO SUPPORT SOFTWARE RELEASE PLANNING

Robert L. Nord[1], Ipek Ozkaya[1], Nanette Brown[1] and Raghvinder S. Sangwan[2]

[1] Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA
[2] Pennsylvania State University, Malvern, PA, USA

ABSTRACT
Organizations building products or systems that rely on software continue to demand increasingly rapid innovation and development processes that enable them to adjust products and systems to emerging needs. Release planning is a key activity in managing these processes. An essential aspect of release planning is balancing the development of new capabilities to address user needs against investment in system infrastructure and architecture to enable flexibility and maintainability. Providing quantifiable insight and visibility into both the delivered capabilities as well as the emerging quality of the software architecture is essential to product success. In this paper, we describe our use of dependency structure and domain mapping matrices to model architectural dependencies. These dependencies provide insight into the value of the capabilities being delivered over total effort to better guide the process of release planning.

Keywords: Software architecture, iterative release planning, dependency analysis, Design Structure Matrix (DSM), Domain Mapping Matrix (DMM), software economics

1 INTRODUCTION

The possible decisions over the course of release planning within any incremental and iterative software development process involve the trade-off of delivering value early to the customer with reducing cost by investing in infrastructure that will avoid rework in subsequent releases (Larman and Basili, 2003). What is needed 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 present our approach to provide guidance using dependency mapping and analytic techniques that provide insight into the cost and value implications of specific iterative delivery strategies. The intent is not to create a new life cycle development process, but to provide visibility into the architectural elements on which the capabilities (also known as features) that provide customer value depend, so that the stakeholders can make informed decisions based on cost and value. We take dependencies into consideration, including dependencies between:

1. capabilities, including functional and quality attribute (i.e., non-functional) requirements
2. capabilities and architectural elements,
3. architectural elements.

Understanding these dependencies allows for optimization of development activities across the releases of an iterative and incremental approach in support of the delivery of customer value (Denne and Cleland-Huang, 2003).

2 APPROACH TO MODELING AND MANAGING DEPENDENCIES

Balancing short-term and long-term needs requires a viable economic strategy that provides guidance on when to spend time designing versus delivering capabilities (Brown et al., 2010; Nord et al., 2011). Dependency management has been studied at the level of code artifacts and in the context of system engineering (Browning, 2001). It has also been studied in the context of requirements and design traceability in support of managing iterations (Kortler et al., 2010). Applying dependency management at the architecture level is beginning to show promising results due to increasing tool support for using the design structure matrix (DSM) for architectural analysis (Hinsman et al., 2009).

To support balancing short-term versus long-term needs, we use a release planning dashboard shown to the left and matrix-based modeling shown to the right in Figure 1.

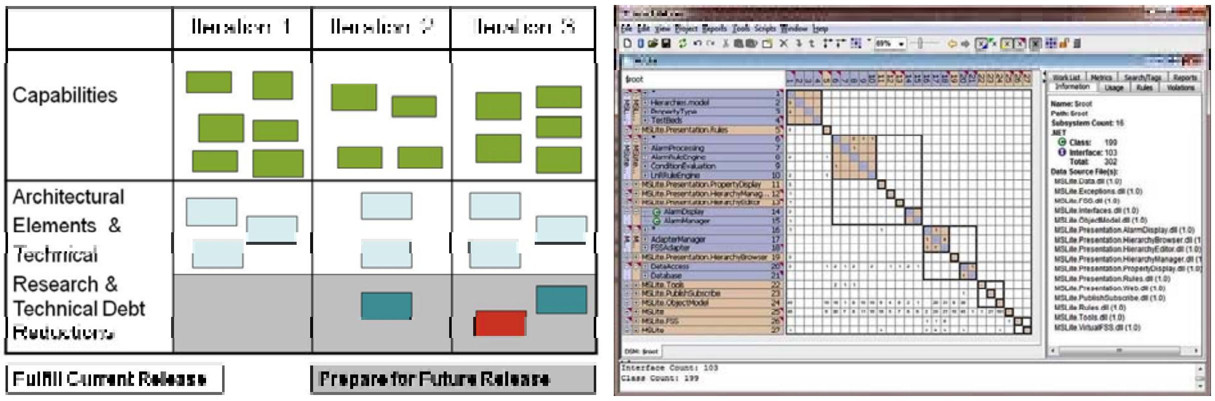

Iteration 1
Iteration 2
Iteration 3
Capabilities
Architectural Elements & Technical Research & Technical Debt Reductions
Fulfill Current Release
Prepare for Future Release

Figure 1. Release planning dashboard and supporting matrix-based modeling

In addition to reasoning about capabilities in the current release (shown in the first row of the release planning dashboard), we also consider modeling architectural elements (shown in the second row of the release planning dashboard) - incorporating investment within the current release to prepare for future releases. The quality of the architecture is assessed in the form of technical debt, that is, short-cuts taken in design that may need to be reworked in the future (Brown et al., 2010).

Support for this form of release planning is provided through the use of design structure matrix-based representations and analysis such as propagation cost, as illustrated to the right in Figure 1. Propagation cost measures the percentage of system elements that can be affected, on average, when a change is made to a randomly chosen element (MacCormack et al., 2008).

In order to reason about the cost and value of the alternative development paths, we need to represent the release planning path in terms of the following properties:

- The release order of the increments in the path,
- Customer requirements delivered,
- Architectural elements delivered,
- Dependencies between elements scheduled for the current release,
- Dependencies between elements for current and previous releases, and between elements for current and projected future releases.

The use of design structure matrices (DSMs) in software engineering has focused on understanding design rules and has been increasingly incorporated into reverse engineering and architecting tools (Lindemann, 2009). In our approach, DSMs represent dependencies between customer requirements, and between architectural elements within the context of an iterative release planning exercise.

The implemented capabilities become the basis for computing the value delivered to the customer. The value of a capability is computed as the weighted sum of the benefit to the end user when implemented and the penalty incurred if postponed, where benefits and penalties are determined by the customer (Wiegers 1999). Dependency analysis is used to determine the precedence in the implementation of the features.

The cost of the implementation is a combination of the cost to implement the architecture elements selected to be added in the release and the cost to rework pre-existing elements. Rework cost is incurred when new elements are added to the system during a 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 (Bachmann et al., 2007). Dependency analysis is used to determine the precedence in the implementation of the architectural elements and to analyse the cost of rework.

Not only do we need to look at dependency analysis within a single domain of requirements or architectural elements, we need to represent dependencies between customer requirements and architectural elements within the context of an iterative release planning exercise. Understanding the dependencies between capabilities and architectural elements enables staged implementation of technical infrastructure in support of achieving stakeholder value. Domain mapping matrices (DMMs) represent this inter-domain mapping (Danilovic and Browning, 2007; Danilovic and Sandkull, 2005). DSM and DMM analysis can be combined to reach deeper conclusions about inter and intra-domain dependencies in a dual-domain context (Bartolomei et al., 2007). In this case, we are looking at return on investment as the value of capabilities delivered over total effort.

When we contemplate a particular step along a release path, we start with an initial selection of requirements and/or architectural elements. To understand the cost and value implications of the potential choice, we need to navigate dependencies.

We need to navigate within and among the domains in either direction, from requirements to architectural elements in a value-driven approach or from architectural elements to requirements in a cost-driven approach.

- Taking a value-driven approach, given a selection of requirements, dependency analysis within the requirements domain allows us to determine requirements precedence and grouping. Dependency analysis from the requirements domain to the architecture domain allows us to determine which architectural elements need to be implemented in support of the requirements.
- Taking a cost-driven approach, dependency analysis within the architectural domain allows us to determine precedence and grouping with respect to architectural elements. Dependency analysis from the architecture domain to the requirements domain allows us to determine which requirements are supported by the implemented architectural elements.

We also need to manage changing dependencies over time, comparing dependencies in the current release to those in previous releases to calculate cost of rework, and comparing dependencies in the current release to those projected in future releases to calculate technical debt. Any rework undertaken in the current release would go towards paying off the technical debt accumulated in the past.

3 MODEL STUDY

We conducted an exploratory analysis of a model problem to quantify the technical debt outcomes of alternate release strategies. We picked the Management Station Lite (MSLite) (Sangwan et al., 2008) system for the study because we have experience with the system 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. 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 to reset these values.
- Trigger alarms notifying appropriate users of life-critical situations.

We used this system to establish metrics for quantifying architecture quality in an earlier study where we used architecture structure metrics based on dependency analysis with propagation cost (Brown et al., 2011).

Metrics alone do not give guidance about how to optimize system development over time. Here, we used the propagation cost metrics from the earlier work, and used the analysis to model the impact of technical debt and pay-back.

We considered the development of the model problem according to three software development paths that characterize the dependencies associated with choosing a singular value-focused or cost-focused strategy, or a blend of both strategies with a focus on integrated return on investment.

- Path 1: *value focused*. Development focused on delivering the high value capabilities as soon as possible, making expedient decisions and deferring implementation of architectural elements. Releases were planned to occur evenly over the course of development (every two iterations). Architectural elements were implemented only when they could no longer be delayed, in this case, when the acceptance test cases, representing the quality attribute requirements, required them.

- Path 2: *cost focused*. Development focused on implementing architectural elements in such a way as to minimize rework, and added capabilities as a by-product of when the supporting elements were in place. By carefully considering dependencies, the amount of rework is zero.
- Path 3: *integrated return on investment*. Development focused on delivering the high value capabilities and pulled in the needed architectural elements on demand to support the capabilities.

Figure 2 shows the value of capabilities delivered over total effort for each of the three paths over five releases.

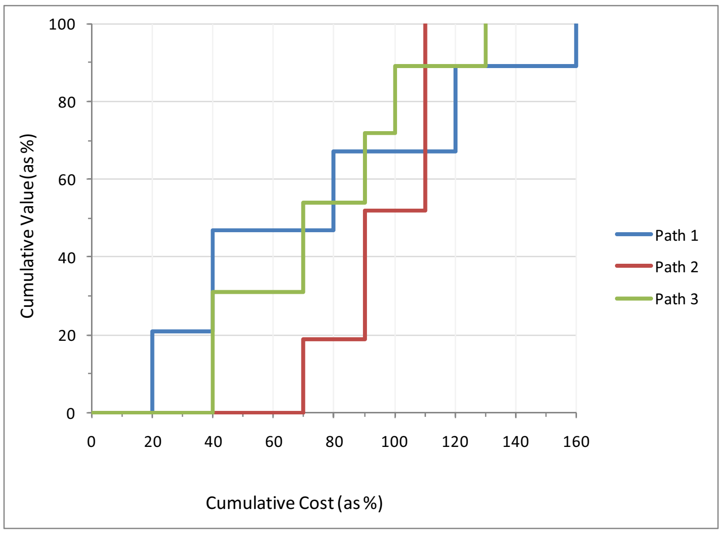


Figure 2. Value of capabilities delivered over total effort

The total implementation effort of the system independent of rework is depicted as 100% cost on the x-axis of the figure. The additional cost over 100% reflects the rework or expense to deal with the technical debt.

Iterations are uniform duration and reflect cadences of development effort. In this case, there are ten iterations spanning development, each iteration representing 10% of the release cost. The product owner asks for the high priority capabilities and the developers say what is possible given the allotted resources. Developers plan at the granularity of tasks to develop elements that are needed to implement a capability.

Each path releases five increments of the product over the course of development according to their own timeline. Releases reflect stakeholder value and so are not uniform in duration.

- Path 1 shows high value during the first two releases, but the delivery of value tapers off as subsequent releases take longer, an indication of the rework needed to deal with the growing complexity of dependencies.
- Path 2 shows there is no value delivered to end users early on, as the team focuses on the architecture. Once the architecture is in place, the team settles into a rhythm of releasing high value capabilities every two iterations. In the ideal case, this path would have 100% value at 100% cost since there is zero rework. The extra 10% cost is a reflection of the granularity of the capabilities and elements and how they are allocated to iterations and releases.
- Path 3 shows that the combined emphasis on high value capabilities and architecture to manage dependencies makes delivery more consistent over time.

The first few releases of path 1 have an advantage compared to path 3 since we are getting more value. However, the expedient choices that defer architecture decisions accumulate so we have more debt to deal with at the end of path 1 (160% vs. 130%).

Path 3 outperforms path 2 for most of the development cycle. Path 2 pulls ahead at the end and has less debt to deal with (110% versus 130%). However, factoring in the cost of delaying the capabilities could make path 3 preferable at the end. And if additional money could not be allocated beyond that projected at the 100% mark, path 3 would have delivered 89% value whereas path 2 would have been capped at 52%.

Figure 3 shows the cadence of iterations per release over time. Each path releases five increments at differing intervals dependent on when there are sufficient capabilities to deliver value to the stakeholders (whether value to customers in the marketplace or value to downstream developers getting releases early for them to do their work).

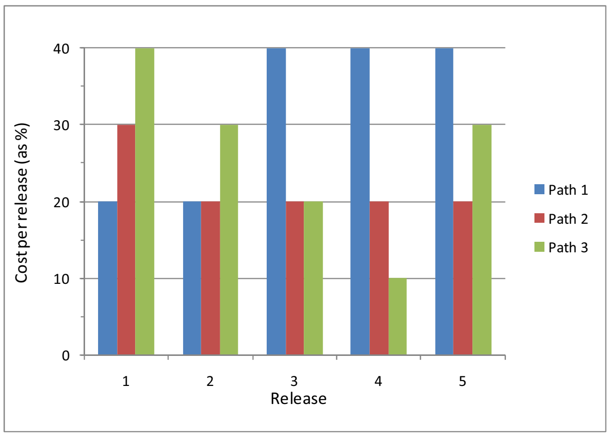


Figure 3. Release cadence

The first two releases of path 1 take two iterations, or 20% of the cost. Subsequent releases take longer at four iterations, an indication of the rework needed to re-architect as quality attribute requirements are considered. Path 2 is development focused, and just as iterations can be time-boxed, it is reasonable to expect releases in this context to have more uniform duration since they are not subject to market forces. For path 3, given the variability of the granularity of packaging marketable capabilities and the architectural elements they depend on, it can be expected that releases do not follow a regular cadence.

Another way of looking at the data in Figure 2 is to see what value the customer is accruing for a given cost. At 40% of the cost, path 1 has released two increments with 47% of the value, and path 3 has released one increment with 31% of the value. Path 2, with the focus on architecture has yet to produce any customer-oriented capabilities, and thus no value. At 100% of the cost, path 3 has achieved 89% value and is projected to take 3 more iterations to release the final increment. Path 2 has achieved 52% value and is projected to take one additional iteration to complete. Path 1, has achieved 67% value, and is projected to take 6 more iterations (spanning two releases) to complete.

The choices to make along the path between the competing interests of cost and value to meet the needs of a specific customer are situational. In certain contexts, a focus on value and 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, delayed release in the interest of reducing later rework cost might be the choice that better aligns with project and organizational drivers and concerns. This analysis provides visibility into the dependencies to support making informed choices.

4 SUMMARY AND FUTURE WORK

Our exploratory analysis demonstrates that dependency metrics can be extracted from the architecture and represented in the form of a DSM. DMM analysis can augment DSM analyses and can be used to represent the dependencies between capabilities and architectural elements to support iterative release planning where the ability to adjust courses of action is essential as the project progresses.

Based on these initial results, our research will continue with the following goals:

- Scaling techniques to model larger systems.
- Determining how much modeling is enough?
- Dealing with uncertainty and adjusting course over time.
- Tool support to transition these techniques into practice.

ACKNOWLEDGMENT

The Software Engineering Institute is a federally funded research and development center sponsored by the US Department of Defense.

REFERENCES

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

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

Brown, N., Nord, R., and Ozkaya, I. (2010). Enabling Agility through Architecture, *Crosstalk Magazine*, Nov/Dec.

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

Brown, N., Nord, R., Ozkaya, I. and Pais, M. (2011). Analysis and Management of Architectural Dependencies in Iterative Release Planning. *9th Working IEEE/IFIP Conference on Software Architecture (WICSA 2011),* June.

Browning, T.R. (2001). Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions, *IEEE Transactions on Engineering Management*, 48(3).

Danilovic, M., and Browning, T.R. (2007). Managing complex product development projects with design structure matrices and domain mapping matrices, *International Journal of Project Management*, 25, 2007.

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

Denne, M., and Cleland-Huang, J. (2003). *Software by Numbers: Low-Risk, High-Return Development*, Prentice Hall, 2003.

Hinsman, C., Sangal, N., and Stafford, J. (2009). Achieving Agility Through Architecture Visibility, in LNCS 5581/2009, *Architectures for Adaptive Software Systems*, pp. 116-129.

Kortler, S., Helms, B., Berkovich, M., Lindemann, U., Shea, K., Leimeister, J.M., and Krcmar, H. (2010). Using MDM-Methods in Order to Improve Managing of Iterations in Design Processes, *12*th *International Dependency and Structure Modelling Conference, DSM'10*, July.

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

Lindemann, U. (2009). Technical DSM Tutorial, 2009. http://dsmweb.org.

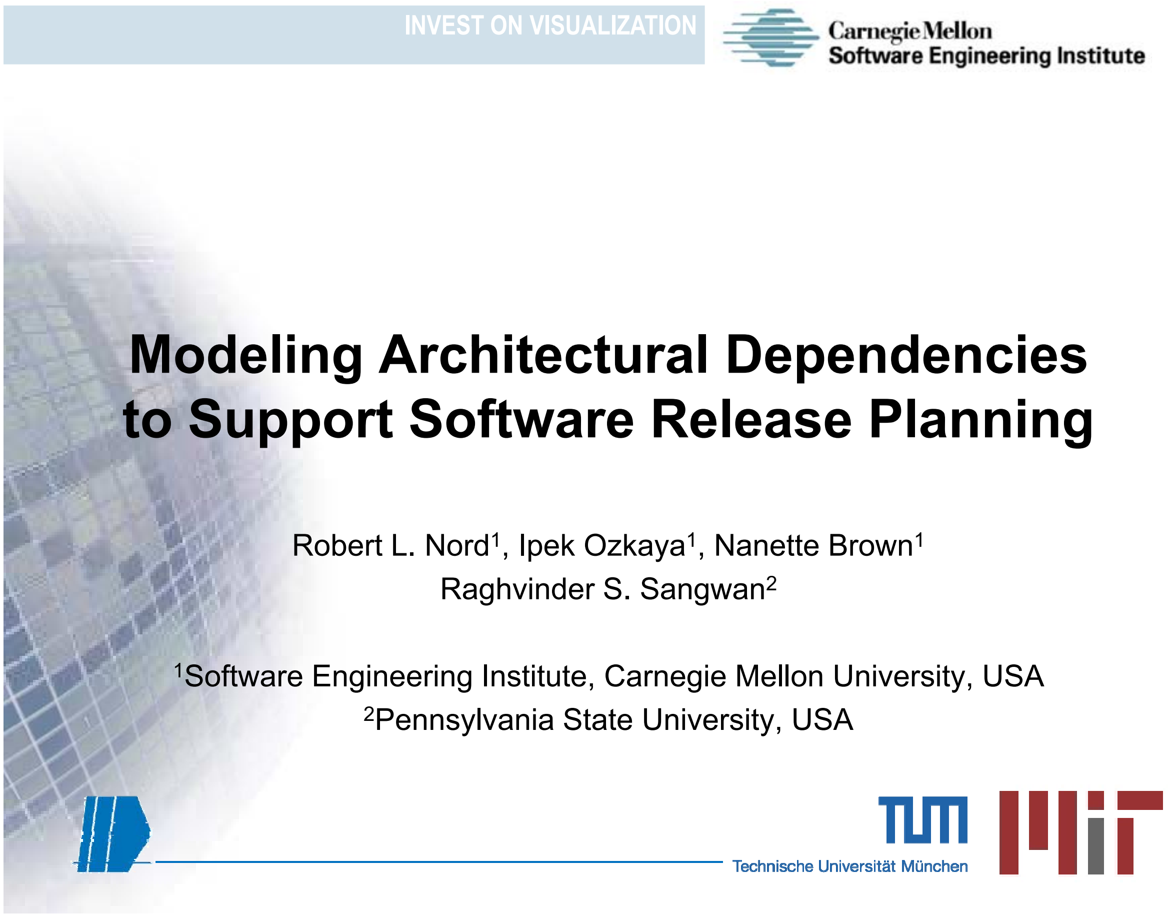
MacCormack, A., Rusnak, J., Baldwin, C. (2008). Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis, Harvard Business School, October 10 (Version 3.0).

Nord, R., Brown, N., Ozkaya, I. (2011). Architecting with Just Enough Information, *Sixth Workshop on SHAring and Reusing architectural Knowledge (SHARK 2011).* In: *Proceedings of the International Conference on Software Engineering (ICSE) 2011*. Honolulu, HI (USA), May 23, ACM.

Sangwan, R., Neill, C., Bass, M., and El Houda, Z. (2008). Integrating a software architecture-centric method into object-oriented analysis and design, *Journal of Systems and Software*, 81, May, 727-746.

Wiegers, K. (1999). First Things First: Prioritizing Requriements, Sofwtare Development.

Contact: Robert L. Nord
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA, USA
Phone: +1 (412) 268-1705, Fax: +1 (412) 268-5758
e-mail: rn@sei.cmu.edu, www.sei.cmu.edu



INVEST ON VISUALIZATION

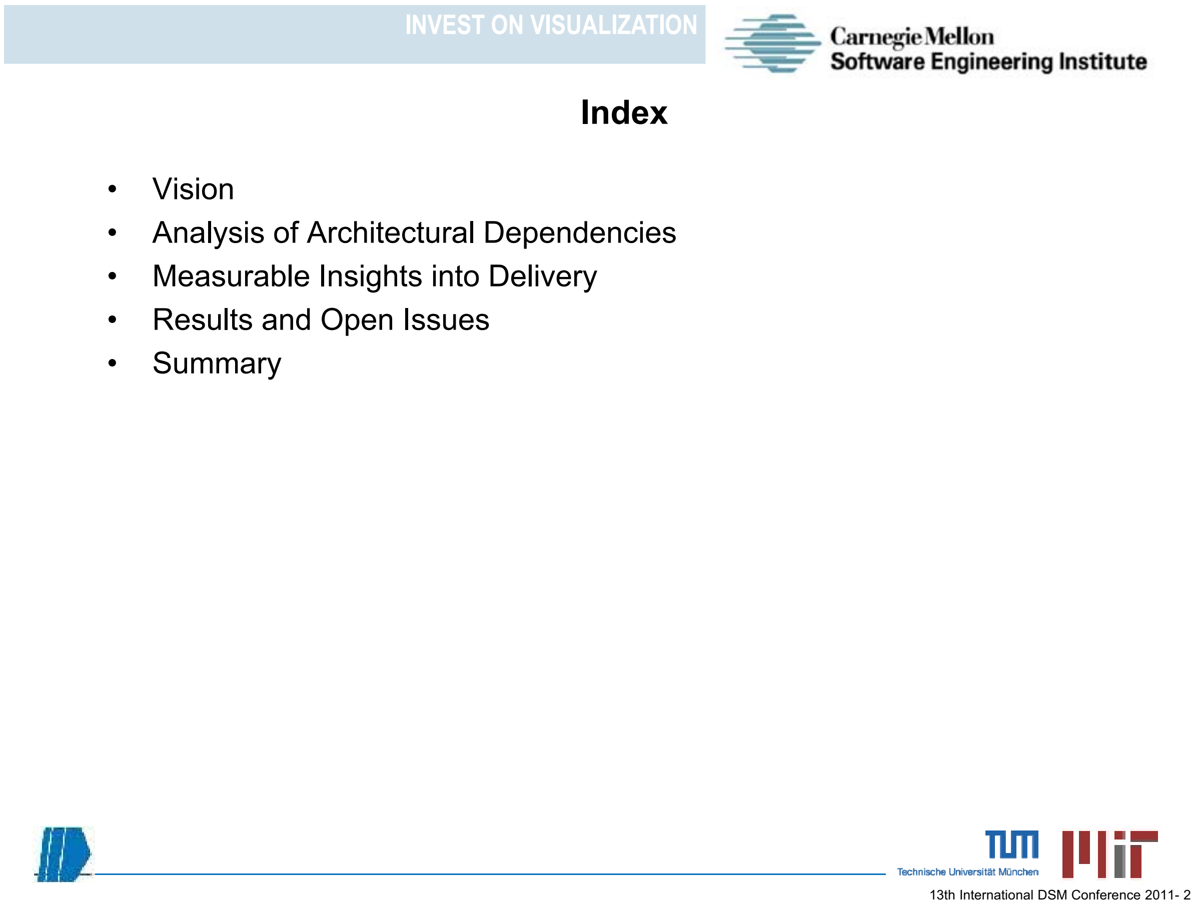
Carnegie Mellon
Software Engineering Institute

Modeling Architectural Dependencies to Support Software Release Planning

Robert L. Nord[1], Ipek Ozkaya[1], Nanette Brown[1]
Raghvinder S. Sangwan[2]

[1]Software Engineering Institute, Carnegie Mellon University, USA
[2]Pennsylvania State University, USA

Technische Universität München



INVEST ON VISUALIZATION

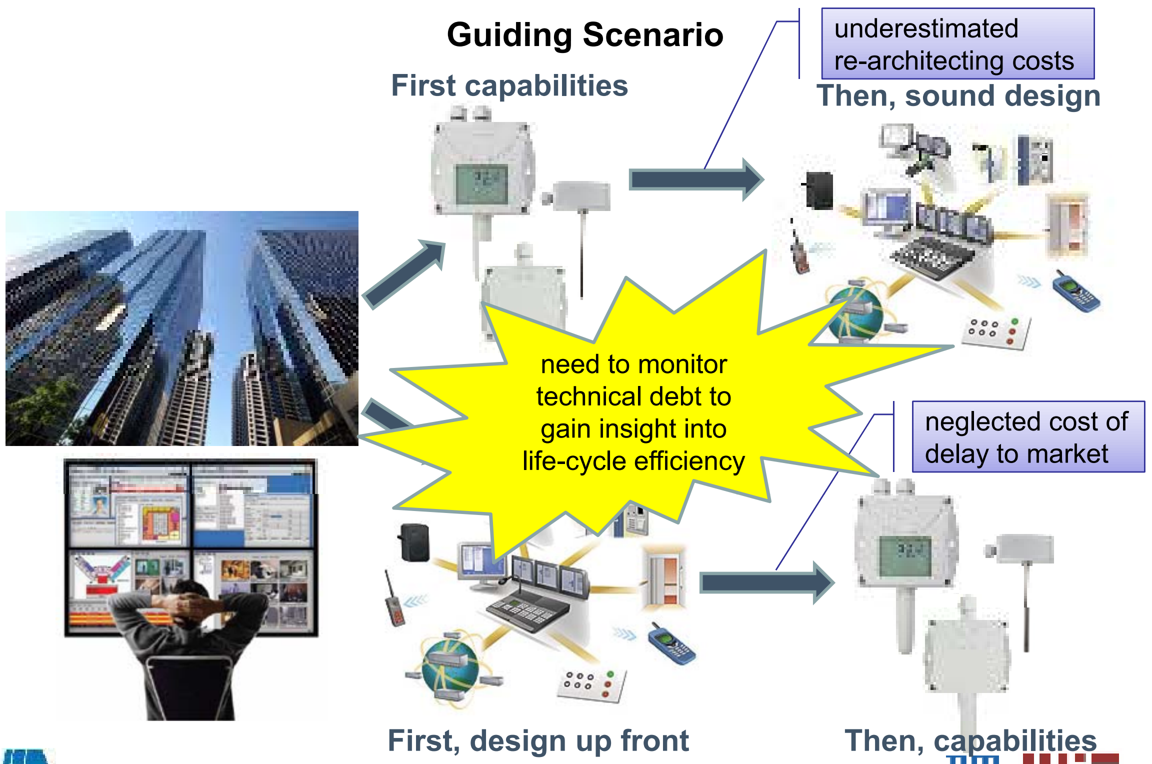
Carnegie Mellon
Software Engineering Institute

Index

- Vision
- Analysis of Architectural Dependencies
- Measurable Insights into Delivery
- Results and Open Issues
- Summary

Technische Universität München

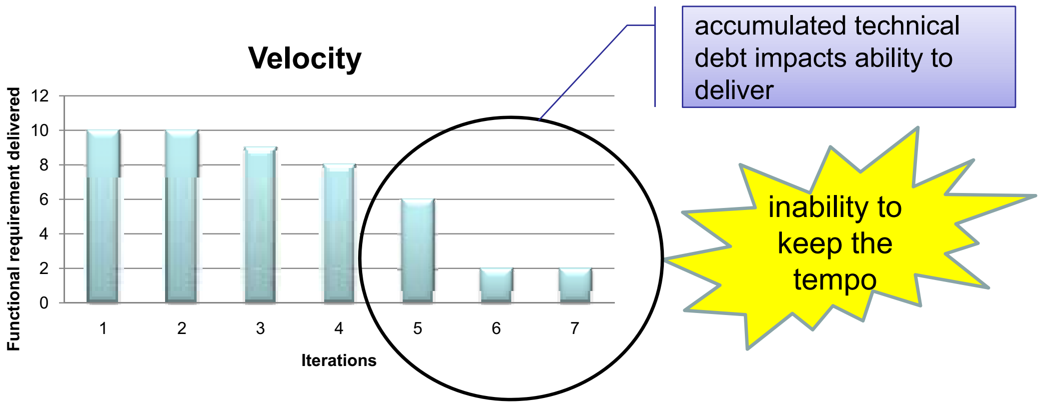
Guiding Scenario


First capabilities
underestimated re-architecting costs
Then, sound design
need to monitor technical debt to gain insight into life-cycle efficiency
neglected cost of delay to market
First, design up front
Then, capabilities

Focus on Value

Standard iteration management in agile development processes

➔ functional, high-priority stories allocated first.



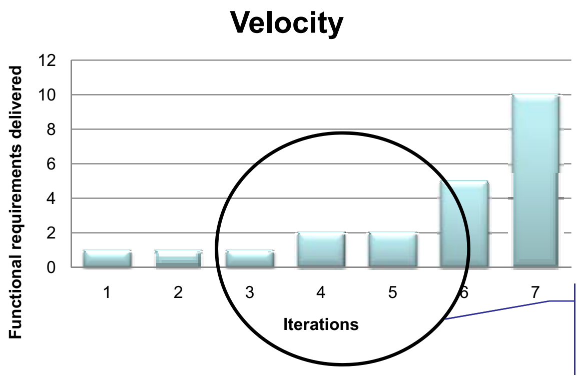
Tracking and monitoring mechanism is solely based on customer capabilities delivered.

Carnegie Mellon Software Engineering Institute

Focus on Cost

Standard iteration management in phase-based development processes

➔ up-front requirements and design tasks allocated first.



delayed customer delivery

taking on some debt can increase tempo

No explicit and early tracking and monitoring mechanisms that is development artifact specific.

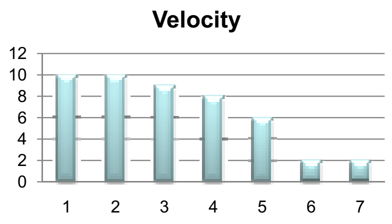
Technische Universität München

Carnegie Mellon Software Engineering Institute

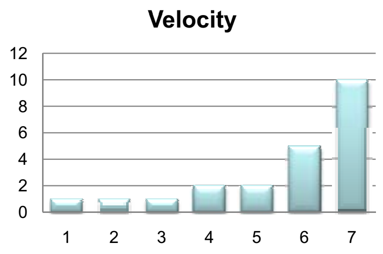
Vision

Manage architectural dependencies with dependency structure matrices (DSMs)

Focus on Value

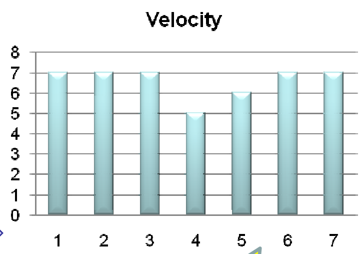


Focus on Cost



Use metrics to monitor and focus development tasks

Focus on Integrated ROI



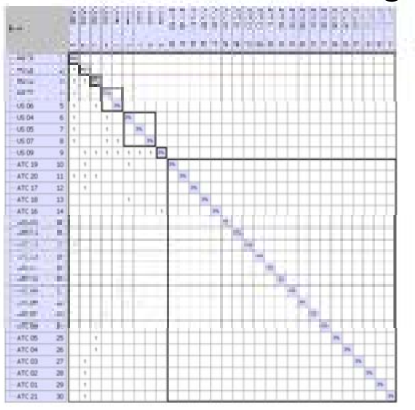
Ability to adjust course with empirical basis

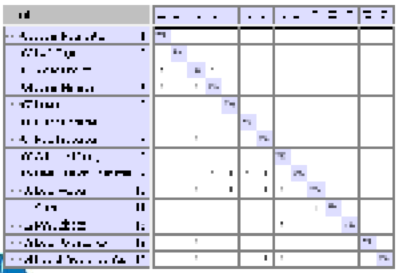
Technische Universität München

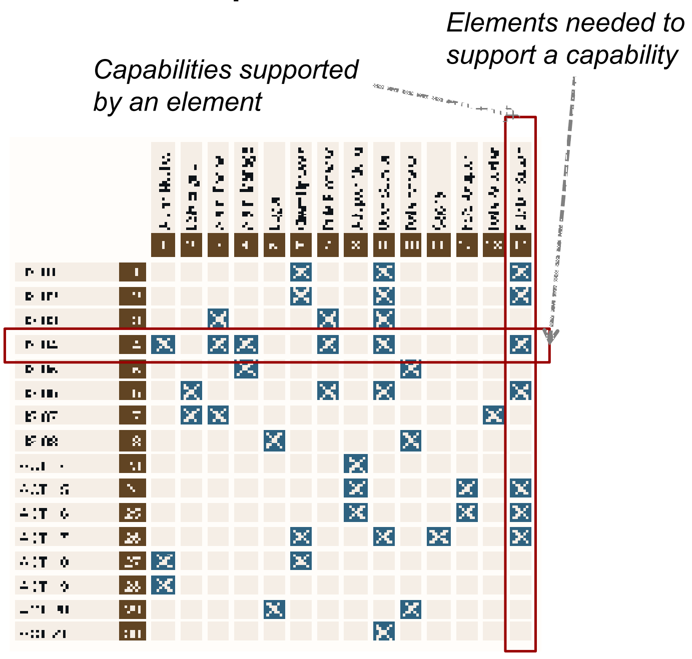
Dependency Management

Dependencies between capabilities & supporting architectural elements	Understanding the dependencies between capabilities and architectural elements enables staged implementation of technical infrastructure in support of achieving stakeholder value.
Dependencies between architectural elements	Dependency analysis within the architectural domain enables determination of precedence and grouping with respect to architectural elements.
Dependencies between capabilities	Dependency analysis within the requirements domain enables determination of requirements grouping and precedence.

Modeling Architectural Dependencies

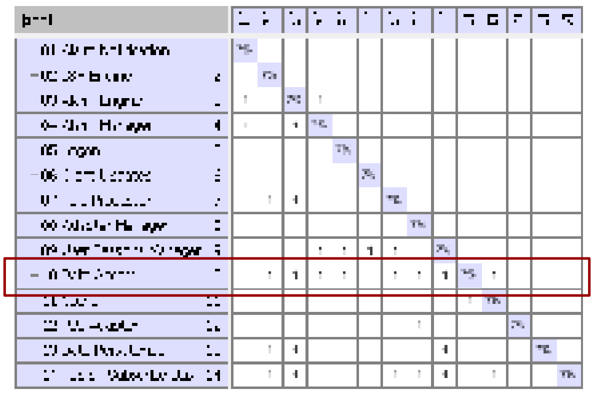

DSM – requirements


DSM – architectural elements


Capabilities supported by an element
Elements needed to support a capability
DMM – requirements to elements

Carnegie Mellon Software Engineering Institute

Analyzing Architectural Dependencies



Total cost = F(C_i, C_r), a function of implementation and rework cost.

Implementation cost is given for all individual architectural elements.

Rework cost for release *n* is computed:

- SUM($C_r(E_k)$) for all new elements E_k
- $C_r(E_k)$ = SUM ($C_r(E_j)$) for all pre-existing elements E_j
- $C_r(AE_j) = D(E_j, E_k) * C_i(E_j) * Pc(n-1)$ where D() is # dependencies and Pc is propagation cost.

$$Pc = \frac{\sum_{i=0}^{n} M^{i}}{n^2}$$

Making dependencies visible earlier in the development life cycle accompanied by metrics improves communication of architecture quality.

Metrics alone do not give guidance about how to optimize value over time.

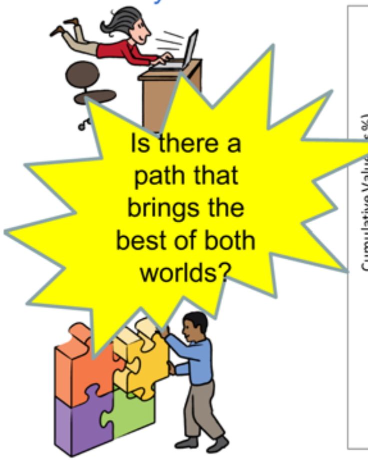
We can improve project monitoring by providing quantifiable quality models of the architecture during iteration planning.

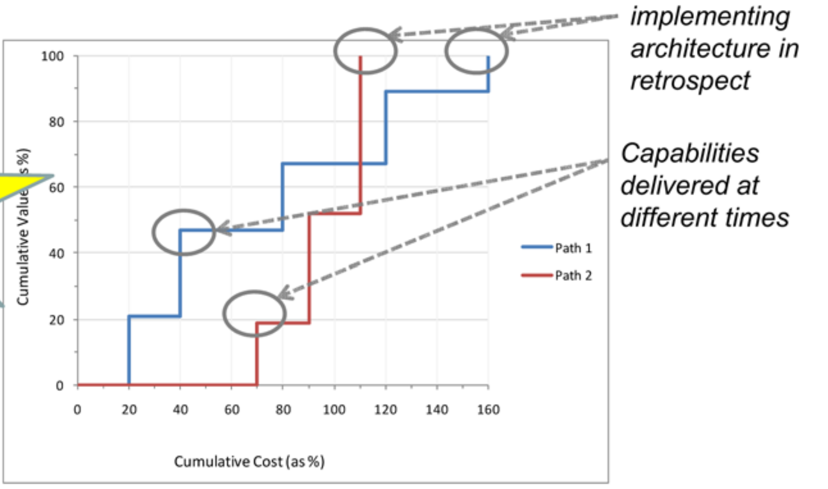
Technische Universität München

Carnegie Mellon Software Engineering Institute

Measurable Insights into Delivery -1

Path 1: *value focused; functionality first.*





Value of Capabilities Delivered over Total Effort

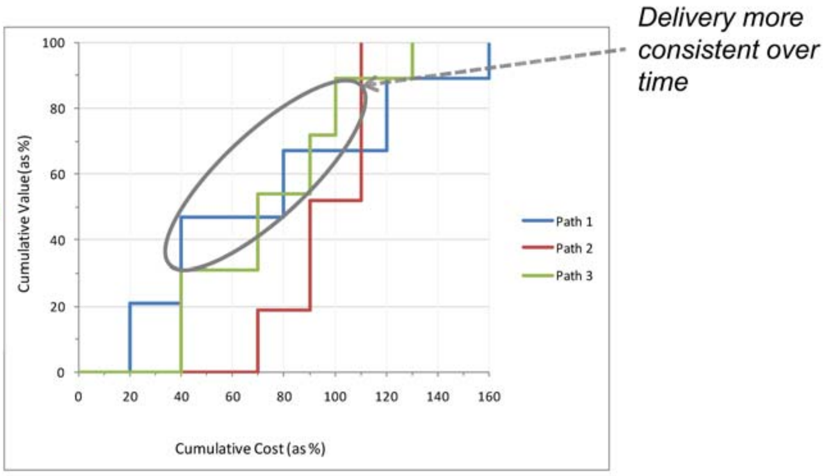
Path 2: *cost focused; architecture push.*

Technische Universität München

Carnegie Mellon
Software Engineering Institute

Measurable Insights into Delivery -2

Path 3: *integrated return on investment; functionality pulling architecture.*



Value of Capabilities Delivered over Total Effort

Carnegie Mellon
Software Engineering Institute

Results

The rework algorithm is directional and represents an effort to formalize the impact of architectural dependencies.

Rework cost is a relative value, used to compare alternative paths and to provide insight into architectural quality across releases within a given path.

A key aspect of managing strategic technical debt is the ability to quantify degrading architecture quality and the potential for future rework as each release is planned.

Open Issues

What are the appropriate proxies of complexity that affect cost of change?

How do we incorporate uncertainty and the forecast of rework in the model?

How do we characterize the economics of architectural violations across a long-term roadmap, rather than enforce compliance for each release?

Carnegie Mellon
Software Engineering Institute

Summary

Exploratory analysis demonstrates that dependency metrics, such as propagation cost, can be extracted from the architecture and represented in the form of a DSM.

DMM analysis can augment DSM analyses and can be used to represent the dependencies between capabilities and architectural elements to support iterative release planning where the ability to adjust courses of action is essential as the project progresses.

Based on these results, our research will continue with the following goals:

- Scaling techniques to model larger systems
- Determining how much modeling is enough
- Dealing with uncertainty and adjusting course over time
- Tool support to transition these techniques into practice