Understanding the Role of Constraints on Architecturally Significant Requirements

Neil A. Ernst, Ipek Ozkaya, Robert L. Nord, Julien Delange, Stephany Bellomo, Ian Gorton
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, USA
{nernst,ozkaya,rn,jdelange,sbellomo,igorton}@sei.cmu.edu

***Abstract*—A key constraint on software development is reliance on tools, which we define as COTS products, software services, languages, frameworks and platforms. These tools may have significant architectural impacts that are not obvious when the requirements are elicited, tools selected, and architecture sketched out. In this paper, we report on a case study we conducted to identify architecturally significant requirements (ASRs) that were impacted by tool selection. We identified ASRs in an existing health IT project, CONNECT, and also identified the constraints on the project that were tool-related. We produce a mapping showing how the architectural risks identified in the initial architectural analysis were impacted by the tool choices made. We produce metrics showing how much time has been consumed when implementing ASRs that involve working around/with these constraints and the risks associated with them.**

***Index Terms*—work items, architecturally significant requirements, architecture analysis, constraints, CONNECT.**

I. Introduction

Delivering software in a manner that is both rapid and stable requires some focus on constraints that affect architecture decisions, as our previous work [1] revealed. These constraints are often embedded in tool choices such as commercial off-the-shelf (COTS) products, software services, languages, frameworks, and platforms. Furthermore, a specific finding of that research was that a key constraint on rapid fielding was COTS products which could sometime inhibit rapid delivery of value. In this paper, we examine the nature of these constraints in more depth, with specific reference to a case study we conducted on a large-scale government IT system, CONNECT. A note on 'constraints': we use this term in the sense of a decision which *guides* future decisions. That is, the presence of a constraint in a software project, such as the need to use Java 7, both enables and inhibits certain project-specific decisions. It will be more difficult to work with Microsoft systems, perhaps, but offers a rich set of standard library APIs.

As an example, consider the following situation. A software development organization has been given the mandate to ensure that a system will correctly integrate with external entities. We call this an *architecturally significant requirement* (ASR) [2]. In order to verify that this ASR is implemented, the team will engage in integration testing. However, the target system runs on Windows, while the developers are testing the changes on Linux. This is the *constraint*—a pre-existing infrastructure decision that impacts the ASR. In order to proceed with satisfying the ASR, the team must change its testing practice.

In this paper we:

- identify the presence of constraints by examining the development artifacts of the team (including issue tracking, source code, and commit logs);
- define the construct *Architecturally Significant Work Item* as a way of measuring architecture tasks using CONNECT;
- explore how different constraints have different impact on work item completion times.

II. Methodology

We followed an exploratory case study protocol as described in Yin [3]. Our research questions are:

1) To what extent is architecture impacted by pre-existing constraints?
2) What form do these constraints take?
3) How do the constraints impact project success?
4) How easy is it to work with/around the constraints?

We narrowed the research questions to study propositions:

SP 1. Can we identify what constraints are present in the CONNECT project?

SP 2. Given these constraints, what impact have they had on project effort?

SP 3. What metrics or measures are useful to derive that impact?

Our causal claim is that technology choices have significant impact (either negative or positive) on project architectural work, which in turn impacts project success. This exploratory case study is intended to give partial answers to this claim by deriving metrics which are useful for understanding the construct of *impact*. We leave the identification of impact on *success* to future work. The case study is conducted with one software project, and our unit of analysis is the individual architectural risks as identified in a systematic architectural analysis using the Architecture Tradeoff Analysis Method [4] undertaken in December 2011, by a team which included the second author [5]. We use a narrative analytical approach to the case study, relying primarily on the artifacts produced by the project

and as output of the architectural analysis, then analyzing these quantitatively. An outline of our methodology is as follows:

1. Select the risk themes identified in the ATAM for analysis.
2. From the CONNECT source code and repository, extract constraints for this project. These constraints are independent of the risk themes identified in the ATAM (step 1).
3. Examine the CONNECT issue tracker, and other artifacts, such as Excel spreadsheets, to derive work items relevant to our architecture risks, called Architecturally Significant Work Items (ASWIs).
4. Manually code those work items which are somehow relevant to the constraints identified in step 2.
5. Derive measures for estimating constraint impact on ASRs.

A. Background

CONNECT[1] is an initiative of the U.S. federal government to interconnect health-care information, beginning with governmental agencies, but ultimately other private organizations. It is licensed under the BSD license, with the intent that other vendors adopt the code and use it in proprietary software and support offerings. CONNECT is developed using a modified agile approach. They follow Scrum as their agile software development methodology. They hold bi-weekly sprints (83 as of late 2012) and make fairly frequent releases. There have been 45 separate committers.

The project collects requirements from the key stakeholders, some of whom include the Department of Defense (DoD), Federal Housing Authority, and the Center for Medicare and Medicaid Services (CMS). It therefore has multiple important customers, and since it concerns health information, must abide by many governmental regulations. The development is contracted to private consultants and developed in several different offices. Periodic code sprints bring all developers together to sync. A change control board (CCB) made up of the cooperating agencies governs the process of prioritizing development.

All agencies have developed a list of requirements beforehand, which are then amalgamated through the CCB. However, because the development is iterative, certain stories are left for future sprints, and the requirement is not fully described until it becomes part of an active sprint, as needed. JIRA, the project's issue tracker, uses a particular ontology of software development, anchored in the Epic/User Story/Task/Bug hierarchy, but this is not the same as the breakdown CONNECT uses, which includes separate requirements labels (e.g., REQ-097, EST009 are both names for the logging requirement). Thus the JIRA fields are a mélange of JIRA labels and CONNECT language. For example, in the Description field, CONNECT developers had added:

QA: No new tests required, Validation Suite should suffice.
CC: Codereview, checkin, validation suite passes.

It should be noted that JIRA supports full customization of the field labels and organization, which has been done to add the "Epic" issue type, but nothing more.

B. Architecture Tradeoff Analysis

The second author was involved in a multi-day workshop that conducted architecture risk analysis using the Software Engineering Institute's Architecture Tradeoff Analysis Method (ATAM)[2]. Broadly speaking, an ATAM involves a team of experts and stakeholders meeting in person to construct system scenarios. These scenarios capture both functional requirements and quality attributes the system should satisfy, such as performance or usability. The existing system architecture is evaluated with respect to a subset of these scenarios, which allows for the identification of *risks, non-risks, sensitivity points*, and *tradeoffs*. Risks are grouped into *risk themes* that are traced back to their impact on the architecture and business goals. The ATAM approach is shown in Fig. 1.

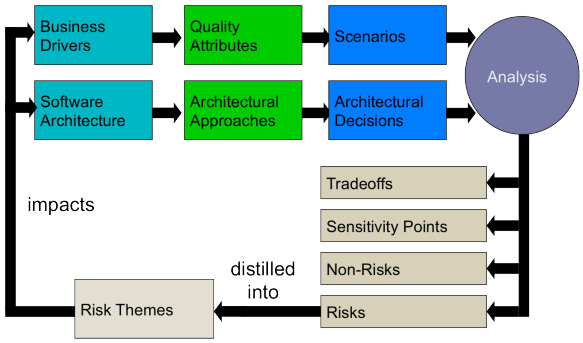

Figure 1 - Conceptual Flow of ATAM [4]

Key risk themes for CONNECT were identified during the ATAM. The ATAM evaluation was conducted for release 3.x and the risks identified for this version included (with our code in parentheses):

- *(NA) Missing and incomplete analysis* – this refers to the lack of testing and modeling of essential properties like performance. One key requirement the stakeholders identified was the need to handle large image files, but there was no easy way to assess whether this requirement was satisfied.
- *(CC) Configuration and integration complexity* – since CONNECT is middleware, configuration of CONNECT for the variety of systems to which it integrates is highly complex.
- *(CDM) CONNECT/DIRECT goal mismatch* – the DIRECT project is a related health integration approach that has a focus on a lightweight data exchange format using SMTP messaging.
- *(AGS) Adapter/gateway separation* – CONNECT initially separated the handling of messages from the integration with other systems. However, these roles have become confused over time, and it is not clear how the roles should be separated.

[1] http://www.connectopensource.org
[2] http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm

[2] http://www.sei.cmu.edu/architecture/tools/evaluate/atam.cfm

- *(AO) Architecture and requirements omissions* – The ATAM revealed some architectural decisions had not yet been made or understood.
- *(MG) Glassfish/Metro dependency* – CONNECT used Glassfish for its Java Enterprise Edition implementation, with Metro as a framework for various web service specifications. However, Metro is not the choice of many potential partners.
- *(DOC) Documentation* – Missing, out-of-date, and incomplete documentation for the architecture and CONNECT developers, impacting future open-source growth, upgrades and maintainability.

We use these as assumed architectural risk themes for the CONNECT project, with the exception of architectural omissions, as that theme was too broad to be useful. The question that this case study examines is the degree to which constraints impacted these risks.

C. Constraints

We defined constraints in the introduction as tool choices impacting the software project. Ideally identifying constraints would be automated, but for this study, the first author scanned the publically available source code[3] of the project, including the build instructions, and identified constraints using the following categories.

1) Commercial Off The Shelf Software (COTS)

This includes middleware, database products, and other internal but not product-specific dependencies. We automatically scanned all Java code for import statements and compared package names for these imports. We defined COTS use as a package that did not start with CONNECT packages. There were 17,800 import statements in total.

- 7154 were CONNECT internal dependencies, i.e., started with `gov.hhs.fha.nhinc.*`.
- 2340 were other health IT dependencies, most of which were to the HL7 messaging standard.
- 2411 were Java SDK dependencies.
- 2706 were test dependencies including JUnit.
- Remaining dependencies were numbered in the hundreds, including OASIS standards, log4j logging dependencies, webservices specific including Apache's CXF product and OpenSAML, Hibernate, an object-relational data persistence tool, and Spring, the dependency injection framework.

CONNECT's build and installation instructions also make note of dependency on MySQL for database server, and a number of MySQL SQL schema files are present.

2) Software services

This category includes external dependencies on web services and other external API calls. For CONNECT we searched the source for occurrences of the word URL, since Java's standard URL class is commonly used to encode external API calls. The majority of the results referred to CONNECT's own service implementation of URL endpoints, so we do not consider CONNECT to be directly using software as a service (SAAS).

3) Languages

A simple measure using the tool CLOC[4] identifies what programming languages were used. For the entire package, which includes build scripts, but no archives, CONNECT has 3.8 million lines of XML source code (without comments), 134 thousand lines of Java source code, and then much smaller amounts of Javascript, SQL, CSS, XSD, etc. The extensive amount of XML comes from web service test cases for CONNECT's service endpoints.

4) Frameworks

A framework is an internal dependency, much like COTS, that involves a significant architectural commitment, in terms of adopting the key architectural styles of those frameworks, such as Spring or Ruby on Rails. These dependencies were identified by examining the build information in the source download directory for version 4. The major framework choice for CONNECT has been its use of the Java Enterprise Edition (JEE) application server specification and a choice to use Web Services, and SOAP for implementation, initially with Glassfish as the application server and Metro as the Web Services stack. These choices bring significant benefits (such as Java Server Pages technology, among many others) but is difficult to disentangle from core business logic. Web services and SOAP have been criticized for undue complexity [6].

5) Platforms

Software platforms are deploy-time constraints. CONNECT is distributed using Apache's Maven build tool, which is responsible for downloading and configuring the software, running build scripts, and installing database schemas.

With these as our list of constraints, it should now be possible to use these to identify where they impact work on the architectural risks identified above.

D. Architecturally Significant Work Items

We define an Architecturally Significant Work Item (ASWI) as an element tracked in the CONNECT issue tracker (Jira), that has some relevance to the architectural risks identified above. All ASWIs comprise the set of units of analysis for this case study. That is, we use the ASWIs we identify as **signifiers** of possible constraint impact on ASRs. We identified all ASWI in the CONNECT issue tracker. Since we need to consider the impact of the ATAM, we selected only those issues opened since the ATAM was prepared—January 1, 2012. All decisions (architecture, tools) up to that point in time represent the constraints that future development is working with. We leave 'relevance' under-defined: it means that somewhere in the text of the issue some reference is made to the risks identified in Section II.B above. The CONNECT issue tracker is made up of several different fields, including comments by interested parties, including non-core members (such as members of the public or affected agencies), issue descriptions, links to related issues. In future the notion of relevance might be expanded to examine temporality (e.g., issues closed at the

[3] https://github.com/CONNECT-Solution, HEAD branch, as of April 1 2013.

[4] http://cloc.sourceforge.net/

same approximate time) or social measures such as churn (how many people/which people were involved in the issue).

CONNECT's issue tracker[5] has over three thousand issues, which is on the edge of tractability for manual analysis. We scanned through all issues that were not labeled as 'bugs' in order to identify only those issues that were significant feature work. CONNECT served as a good candidate for our goals as the CONNECT development process is not only open but also moderately rigorous, with useful standards in place to describe issues. A representative issue title is as follows:

"As a CONNECT product owner/adopter, I want to execute smoke tests(initiating) so that we can proceed with pilot testing (issue CONN-212)". We might code this as relevant to the Insufficient Analysis risk theme (NA).

Our query consisted of all user stories, improvements, and feature requests that were closed or resolved, and marked fixed. We looked at 65 User Stories, 116 Improvements, and 82 Feature Requests. We used the ATAM release date of Jan 1, 2012 as a milestone for our analysis.

III. Discussion

A. Data Analysis

In order to account for possible effects from the ATAM, we looked at the difference between pre- and post-ATAM numbers.

- Pre-ATAM: There are 106 work items (i.e., user-stories, feature requests and improvements). 67 we coded as ASWI (associated with a risk theme) and 50 as constraints. 63% of work items are risk-related.
- Post-ATAM: There are 157 work items, 134 of which were coded as ASWI, and 126 as constraints. 85% of items are risk-related.

We can conclude that the ATAM had an impact on what was worked on, so the remainder of our analysis considers only items created after the ATAM was delivered.

From that subset, we went through the data to identify those which were impacted by or possibly enabled by the constraints identified previously. This was done by examining once again the issue itself to see where some of the constraints were mentioned or were possibly relevant. For example, some work items mentioned the need to do testing on Java 6 platforms as well as (standard) Java 7, in order to support partners who used Java 6. Here we have a risk theme, configuration complexity, an ASWI (the fact that testing needs to be done), and a constraint that impacts that theme (the fact that Java 7 is the default and not Java 6).

Table 1 – ASWIs per risk theme, post-ATAM.

Risk theme	Number of ASWIs
Lack of analysis (NA)	35
Config. Complexity (CC)	**40**
CONNECT-DIRECT (CDM)	19
Adapter-gateway (AGS)	5
Metro-Glassfish (MGS)	21
Documentation (DOC)	14

Table 1 shows the summary for Architecturally Significant Work Items found per risk theme. Notice the total is 134, which is the same as the number of post-ATAM items with a risk theme above. Table 2 shows the numbers for constraints.

Table 2 – ASWIs per constraint, post-ATAM.

Constraint	Number of ASWIs
Platform (PLT)	27
Framework (FMK)	43
Language (LAN)	5
Services	0
COTS	**51**

Configuration complexity and lack of analysis seem to have been given the most attention of the risk themes. Similarly, COTS and Framework constraints would be expected to play a large role in a project so highly dependent on integration with existing libraries and tools, not only for testing and analysis, but also for doing the complex lifting of web-services work. These results lend support to our study proposition 1, identifying constraints present in the project.

Our next analysis step was to estimate, given a work item, how impacted it might be because of the existing constraints. For this we have a number of possible measures. One is how long those issues were open, i.e., the time between issue creation and issue close date, as shown in Table 3.

Table 3 – Mean elapsed time to close an ASWI, by risk theme.

Risk Code	Mean Elapsed Time to Close (hrs)
DOC	133.8
CC	66.2
MG	100.1
AGS	233.2
NA	154.6
CDM	70.0

We can compare this to the mean elapsed time to close (in hours) for all closed, resolved user stories, improvements and feature requests, which is 134.5 hrs. For Configuration Complexity, Metro-Glassfish, and Connect-Direct, these issues were resolved significantly quicker than average.

Table 3 shows the average time for each issue tagged with that risk. An improvement to this calculation might look at how long each tuple of <risk,constraint> took to close. This is shown in Table 4. We omit language and services due to the low or non-existent numbers of tagged items.

What is the meaning of this table? One thing we can see is that for the "NA" or "Insufficient Analysis" risk theme, the longest time to close issues on average belongs to the COTS constraint. This makes sense, since testing in CONNECT uses tools like JUnit and SOAPUI. We can read this as saying that testing has a lot of dependency on these COTS libraries, or alternately, that these libraries are a problem point for the project.

[5] http://issues.connectopensource.org

Table 4 -- Risks and Constraints elapsed time to close

Risk Code	Constraint Code	Mean ETC (hrs)	Number ASWIs
DOC	PLT	36.0	2
DOC	FMK	19.8	7
DOC	COTS	22.4	3
CC	PLT	32.7	13
CC	FMK	50.6	22
CC	COTS	44.4	20
MG	PLT	175.1	9
MG	FMK	85.8	10
MG	COTS	41.5	6
AGS	PLT	n/a	0
AGS	FMK	24	5
AGS	COTS	20.2	4
NA	PLT	50.3	4
NA	FMK	64.9	12
NA	COTS	137.5	24
CDM	PLT	30.6	5
CDM	FMK	65.2	7
CDM	COTS	33.2	8

Our analysis did not examine the sentiment behind the issues. We can also compare across constraints: here, Framework constraints caused the longest mean time to close for the Metro-Glassfish issue (85.8 hrs), possibly explained by the dependency on JEE, e.g., for the issue with FMK-MG codes, number GATEWAY-3412, is summarized in Jira as "integrate and deploy CONNECT to a new Open Source Application Server".

B. Study Propositions

We now return to our study propositions from Section I. They were:

SP 1. Can we identify what constraints are present in the CONNECT project?

We conclude that at a broad level we can estimate what external constraints existed, using automated dependency analysis and some knowledge of software frameworks.

SP 2. Given these constraints, what impact have they had on project effort?

SP 3. What metrics or measures are useful to derive that impact?

We defined a simple measure of Elapsed Time to Close for an ASWI in the project task management system. We used that to estimate how long these tasks stayed open for as a proxy for level of effort/pain that item required. We found that there were important distinctions between both risks and constraints. 85% of the issues in the 18 months since the ATAM evaluation are related to risk themes (ASWIs) and of these 94% had an associated constraint. Also, the number of ASWIs related to specific risk theme and constraint pairs is another simple measure to understand the dependencies.

It would be interesting in future research to see if we can better understand nature of constraint as enabler or inhibitor. To do that more dependency information is needed to understand what is being changed and the breakdown of the effort in terms of the implementation effort, work around effort (temporary fixes), and re-architecting (permanent fixes).

C. Rival theories:

While our metrics are largely cost-oriented, it is important to note that tool-selections are not inherently negative. They are enablers of significant value [7]. Like all artifact studies, this paper suffers from only considering the artifacts in the issue trackers, which may not tell the whole story [8]. For example, one issue marked "Fixed" was also commented on with "Closed. Unable to validate requirement." which would seem to imply that it wasn't fixed, but instead should have been marked "Won't Fix". Rival explanations for the results we observed might include that the issue tracker is a product of a particular approach to software development and that the benefits of the constraints are not being surfaced. Furthermore, it is to some extent irrelevant what constraints exist, since those choices may have been imposed from outside the project (i.e., that a full Web Services stack be used rather than a REST approach).

D. Threats to Validity

1) Internal validity

The ATAM was conducted 18 months prior to this study, and its guidance might not have been followed, or if it was, those threats might be already addressed or not a priority at this moment. We assume that the date an ASWI is closed is significant, but this might be a house-keeping close with the actual date of the fix much earlier (for example, someone goes through the Jira tool and gets rid of issues that are no longer relevant). Another problem is that we ignored low-level issues such as tasks or bugs, mainly in the interest of tractability. The large numbers of these issues suggest an automated approach is necessary, perhaps using a technique such as Latent Dirichlet Allocation [9]. Finally, there are artifact repositories that inform planning efforts—such as Word or Excel documents—which we were unable to access reliably.

2) External validity

CONNECT is a unique project: it is relatively open, yet also a large government software project. As such it may be unrepresentative of other projects. Furthermore, because CONNECT is a middleware application, a lot of effort has to be spent making it interconnect with other software and platforms. This means that architecture is of greater importance.

3) Construct validity

Our constructs, such as ASWI, were loosely defined in order to focus on exploring their potential. Not all architecturally significant requirements (ASRs) may be ASWIs, and in fact, one risk theme was omitted entirely due to its loose definition. A tighter definition, which for example states what **isn't** architecturally significant, would be important in further studies. The study only shows problems, not those things made possible by the technology, e.g. the lines of code saved by not having to do it that way.

4) Reliability

Our work is replicable to the extent that the CONNECT issues are all available, and we have shared our coding scheme. However, there is a high degree of subjectivity in how the codes were assigned to the work items. In particular, we used the notion of constraint loosely—in future, we think considering the specific technologies that are constraints, such as a web application server, would make more sense than these poorly defined categories.

IV. Related Work

Any technology selection, whether it be a development tool or a COTS technology, has positive and negative implications for the success of an agile project [10]. For complex technologies and projects, it is impractical to perform an a priori, full cost-benefit analysis of a particular tool selection, as research on COTS selection has suggested [11], and hence its implications, both positive and negative, are only discovered as the project iterations are undertaken.

Software ecosystems research has also identified that there are complex tradeoff and technology dependencies that drive architectural and organizational decisions, such as [12]. The Mining Software Repositories community has made a recent push to explore development analytics [13]. Related work has looked at extracting architecture dependencies [14] and quite a lot of attention paid to architecture metrics, e.g. [15].

V. Conclusion

Technology constraints on software development include framework decisions, platform choices, and available COTS products. All of these design choices have impact on the architecture and the ability of the system to meet its quality and business goals. This paper has shown a technique for analyzing the relationship between constraints and architectural risks themes identified in a prior architectural evaluation. The analysis approach leveraged available project data (e.g., Architecturally Significant Work Items) to provide greater insight into the impact of constraints. Architecturally significant requirements and architecture decisions are not always easy to tease apart. Technology constraints provide a concrete example where their influence on each other is not only specific to the business context, but also the stage of the project. Our exploratory case study suggests that risk themes that are identified, possibly through an architecture evaluation, can guide the identification and prioritization of architecturally significant work items.

Achieving quantification of "how much architecting is enough" or "when to start architecting" or "how to quantitatively relate requirements to architecting" has been difficult. While it is well-known that technology choices and constraints impact project success (negatively or positively) quantifying that impact needs to be part of the decision making process. In this study, we demonstrate one approach where such quantification can be achieved.

Acknowledgements

This paper does not necessarily reflect the opinions of the CONNECT project or its funders and developers. 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. This material has been approved for public release and unlimited distribution. Architecture Tradeoff Analysis Method®, ATAM® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University. DM-0000390

References

[1] S. Bellomo, R. L. Nord, and I. Ozkaya, "A Study of Enabling Factors for Rapid Fielding: Combined Practices to Balance Tension Between Speed and Stability," in *ICSE - SE in Practice Track*, San Francisco, 2013.

[2] P. Clements and L. Bass, "Relating Business Goals to Architecturally Significant Requirements for Software Systems," SEI, Technical Note CMU/SEI-2010-TN-018, 2010.

[3] R. K. Yin, *Case Study Research: Design and Methods*, vol. 5. Beverly Hills, CA: Sage Publications, Inc, 2003.

[4] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012.

[5] J. Klein and I. Ozkaya, "CONNECT Gateway Architecture Evaluation," Software Engineering Institute, Carnegie Mellon, Dec. 2011.

[6] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, "Developing web services choreography standards—the case of REST vs. SOAP," *Decision Support Systems*, vol. 40, no. 1, pp. 9–29, Jul. 2005.

[7] D. Atkins, T. Ball, T. Graves, and A. Mockus, "Using version control data to evaluate the impact of software tools," in *International Conference on Software Engineering*, 1999, pp. 324–333.

[8] J. Aranda and G. Venolia, "The secret life of bugs: Going past the errors and omissions in software repositories," in *International Conference on Software Engineering*, Vancouver, 2009, pp. 298–308.

[9] A. Hindle, N. A. Ernst, M. W. Godfrey, and J. Mylopoulos, "Automated topic naming to support cross-project analysis of software maintenance activities," in *MSR*, Honolulu, 2011.

[10] I. Gorton, "Cyber Dumpster Diving: Creating New Software Systems for Less," *IEEE Software*, vol. 30, no. 1, pp. 9–13, 2013.

[11] K. Smiley, Q. He, E. Kielczewski, and A. Dagnino, "Architectural requirements prioritization and analysis applied to software technology evaluation," in *Symposium on Applied Computing*, Honolulu, Hawaii, USA, 2009, pp. 397–398.

[12] J. D. McGregor, "Ecosystem modeling and analysis," in *International Software Product Line Conference - Volume 2*, Salvador, Brazil, 2012, pp. 268–268.

[13] T. Menzies and T. Zimmermann, "Goldfish bowl panel: Software development analytics," presented at the International Conference on Software Engineering, Zurich, 2012, pp. 1032–1033.

[14] W. Hu, D. Han, A. Hindle, and K. Wong, "The build dependency perspective of Android's concrete architecture," presented at the Working Conference on Mining Software Repositories, Zurich, 2012, pp. 128–131.

[15] E. Bouwers, J. P. Correia, A. van Deursen, and J. Visser, "Quantifying the Analyzability of Software Architectures," in *Working IEEE/IFIP Conference on Software Architecture*, Boulder, CO, 2011, pp. 83–92.