



Visualizing and Measuring Software Portfolio Architecture: A Power Utility Case

Citation

Lagerström, Robert, Carliss Y. Baldwin, and Alan MacCormack. "Visualizing and Measuring Software Portfolio Architecture: A Power Utility Case." Special Issue on DSM Conference 2014. Journal of Modern Project Management 3, no. 2 (September–December 2015): 114–121.

Published Version

<http://www.journalmodernpm.com/index.php/jmpm/article/view/146>

Permanent link

<http://nrs.harvard.edu/urn-3:HUL.InstRepos:34403524>

Terms of Use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Open Access Policy Articles, as set forth at <http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#OAP>

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. [Submit a story](#).

[Accessibility](#)

Visualizing and Measuring Software Portfolio Architecture: A Power Utility Case

Robert Lagerström¹, Carliss Baldwin², and Alan MacCormack²

¹KTH Royal Institute of Technology

²Harvard Business School

Abstract: In this paper, we test a Design Structure Matrix (DSM) based method for visualizing and measuring software portfolio architectures. Our data is drawn from a power utility company, comprising 192 software applications with 614 dependencies between them. We show that the architecture of this system can be classified as a “core-periphery” system, meaning it contains a single large dominant cluster of interconnected components (the “Core”) representing 40% of the system. The system has a propagation cost of 44% and architecture flow through of 93%. This case and these findings add another piece of the puzzle suggesting that the method could be effective in uncovering the hidden structure in software portfolio architectures.

Keywords: Design structure matrices, Software architecture, and Software application portfolio

1 Introduction

Business environments are constantly evolving, requiring continual changes to the software applications that support a business. Moreover, during recent decades the sheer number of applications has grown significantly, and they have become increasingly interdependent. As a result, the management of software applications has become a complex task; many companies find that implementing changes to their application portfolio architecture is increasingly difficult and expensive. To help manage this complexity, firms need a way to visualize and analyze the modularity of their software portfolio architectures as well as the degree of coupling between components.

Baldwin et al. (2014) present a method to visualize the hidden structure of software architectures based on Design Structure Matrices (DSMs) and classic coupling measures. This method has been demonstrated on *individual* software systems (such as Linux, Mozilla and Apache) but not (to the same extent) on software *portfolio* architectures in which a large number of applications have dependencies to each other (Lagerström et al., 2013) and possibly relationships with other types of components such as business groups and/or infrastructure elements (Lagerström et al., 2014a). In this paper, we apply Baldwin et. al.’s architectural visualization and measurement methods to *enterprise* wide applications, using data collected by Cheraghi (2014) at a Nordic power utility company. This data comprises 192 software applications and 614 dependencies between these applications.

We show that the Power Utility's enterprise architecture can be classified as core-periphery. This means that 1) there is one cyclic group (the "Core") of components that is substantially larger than others, and 2) this group comprises a substantial portion of the entire architecture. We find that the Core contains 76 components, representing 40% of the architecture.

The remainder of this paper is structured as follows: Section 2 presents related work; Section 3 describes the hidden structure method; Section 4 presents the power utility case used for the analysis; Section 5 discusses the approach and outlines future work; and Section 6 concludes the paper.

2 Related Work

Many software applications have grown into large systems containing thousands of interdependent components, making it difficult for a designer to understand the complexity of the design. As a result, much recent work on the visualization and measurement of complex software systems has focused on the use of network methods to characterize system structure (Barabási, 2009). Specifically, these methods emphasize identifying the linkages (dependencies) that exist between different elements (nodes) in the system (Simon, 1962). A key concept here is modularity, which refers to the way in which a system's architecture can be decomposed into different parts. Although there are many definitions of modularity, authors tend to agree on the fundamental features: the interdependence of decisions within modules, the independence of decisions between modules, and the hierarchical dependence of modules on components that embody standards and design rules (Baldwin and Clark, 2000).

Studies that use network methods to measure modularity typically focus on capturing the level of coupling that exists between different parts of a system. The use of graph theory and network measures to analyze software systems extends back to the 1980s (Hall and Preiser, 1984). More recently, a number of studies have used social network measures to analyze software systems and software development organizations (Dreyfus and Wyner, 2011). Other studies make use of Design Structure Matrices (DSMs), which highlight the network structure of a complex system using a square matrix (Sosa et al., 2007). DSMs have been used widely to visualize the architecture of and measure the coupling between the components of individual software systems (MacCormack et al, 2012).

3 Method Description

The method we use for network representation is based on and extends the classic notion of coupling. Specifically, after identifying the coupling (dependencies) between the elements in a complex architecture, we analyze the architecture in terms of hierarchical ordering and cyclic groups and classify elements in terms of their position in the resulting network (this method is described in Baldwin et al, 2014).

In a Design Structure Matrix (DSM), each diagonal cell represents an element (node), and the off-diagonal cells record the dependencies between the elements (links): If element i depends on element j , a mark is placed in the row of i and the column of j . The content of the matrix does not depend on the ordering of the rows and columns, but different orderings can reveal (or obscure) the underlying structure. Specifically, the elements in the DSM can be arranged in a way that reflects hierarchy, and, if this is done, dependencies that remain above the main diagonal will indicate the presence of cyclic interdependencies (A depends on B, and B depends on A). The rearranged DSM can thus reveal significant facts about the underlying structure of the architecture that cannot be inferred from standard measures of coupling. In the following subsections, a method that makes this “hidden structure” visible is presented.

3.1 Identify the direct dependencies and compute the visibility matrix

The architecture of a complex system can be represented as a directed network composed of N elements (nodes) and directed dependencies (links) between them. This DSM is called the “first-order” matrix. If the first-order matrix is raised to successive powers, the result will show the direct and indirect dependencies that exist for successive path lengths. Summing these matrices yields the visibility matrix V (Figure 1), which denotes the dependencies that exist for *all* possible path lengths. The values in the visibility matrix are constrained to be binary, capturing only whether a dependency exists and not the number of possible paths that the dependency can take (MacCormack et al., 2006). The matrix for $n=0$ (i.e., a path length of zero) is included when calculating the visibility matrix, implying that a change to an element will always affect itself.

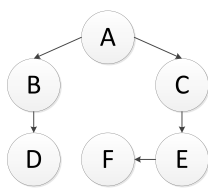
A Directed Graph	Design Structure Matrix	Visibility Matrix $V=\sum M^n; n=[0,4]$																																																																																																		
	<table><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr><tr><th>A</th><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><th>B</th><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><th>C</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr><tr><th>D</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><th>E</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr><tr><th>F</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>		A	B	C	D	E	F	A	0	1	1	0	0	0	B	0	0	0	1	0	0	C	0	0	0	0	1	0	D	0	0	0	0	0	0	E	0	0	0	0	0	1	F	0	0	0	0	0	0	<table><tr><th></th><th>A</th><th>B</th><th>C</th><th>D</th><th>E</th><th>F</th></tr><tr><th>A</th><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td></tr><tr><th>B</th><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><th>C</th><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr><tr><th>D</th><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr><tr><th>E</th><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr><tr><th>F</th><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>		A	B	C	D	E	F	A	1	1	1	1	1	1	B	0	1	0	1	0	0	C	0	0	1	0	1	1	D	0	0	0	1	0	0	E	0	0	0	0	1	1	F	0	0	0	0	0	1
	A	B	C	D	E	F																																																																																														
A	0	1	1	0	0	0																																																																																														
B	0	0	0	1	0	0																																																																																														
C	0	0	0	0	1	0																																																																																														
D	0	0	0	0	0	0																																																																																														
E	0	0	0	0	0	1																																																																																														
F	0	0	0	0	0	0																																																																																														
	A	B	C	D	E	F																																																																																														
A	1	1	1	1	1	1																																																																																														
B	0	1	0	1	0	0																																																																																														
C	0	0	1	0	1	1																																																																																														
D	0	0	0	1	0	0																																																																																														
E	0	0	0	0	1	1																																																																																														
F	0	0	0	0	0	1																																																																																														

Figure 1. A directed graph with the corresponding DSM and visibility matrix.

Several measures are constructed based on the visibility matrix V . First, for each element i in the architecture, the following are defined:

- VFI_i (Visibility Fan-In) is the number of elements that directly or indirectly depend on i . This is found by summing entries in the i^{th} column of V .

- VFO_i (Visibility Fan-Out) is the number of elements that i directly or indirectly depends on. This is found by summing entries in the i^{th} row of V .

In Figure 1, element A has VFI equal to 1, meaning that no other elements depend on it, and VFO equal to 6, meaning that it depends on all other elements in the architecture.

3.2 Identify and rank cyclic groups

The next step is to find the cyclic groups in the architecture. By definition, each element within a cyclic group depends directly or indirectly on every other member of the group. First, the elements are sorted, first by VFI descending then by VFO ascending. Next one proceeds through the sorted list, comparing the VFI s and VFO s of adjacent elements. If the VFI and VFO for two successive elements are the same, they might be members of the same cyclic group. Elements that have different VFI s or VFO s cannot be members of the same cyclic group, and elements for which $n_i=1$ cannot be part of a cyclic group at all. However elements with the same VFI and VFO could be members of different cyclic groups. In other words, disjoint cyclic groups may, by coincidence, have the same visibility measures. To determine whether a group of elements with the same VFI and VFO is one cyclic group (and not several), we simply inspect the subset of the visibility matrix that includes the rows and columns of the group in question and no others. If this submatrix does not contain zeros, the group is one cyclic group. Cyclic groups found via this algorithm are referred to as the “cores” of the system. The largest cyclic group is defined as the “Core”. Once the Core is identified, the other components in the architecture can be classified into groups, as follows:

- “Core” elements are members of the largest cyclic group and have the same VFI and VFO , denoted by VFI_C and VFO_C , respectively.
- “Control” elements have $VFI < VFI_C$ and $VFO \geq VFO_C$.
- “Shared” elements have $VFI \geq VFI_C$ and $VFO < VFO_C$.
- “Periphery” elements have $VFI < VFI_C$ and $VFO < VFO_C$.

Using the above classification scheme, a reorganized DSM can be constructed that reveals the “hidden structure” of the architecture by placing elements in the order Shared, Core, Periphery, and Control down the main diagonal of the DSM, and then sorting within each group by VFI descending then VFO ascending (cf. Figure 4).

The method for classifying architectures into different types is discussed in empirical work by (Baldwin et al., 2014). Specifically, the authors find a large percentage of the architectures they analyzed contained a large cyclic group of components that was dominant in two senses: i) it was large relative to the number of elements in the system, and ii) it was substantially larger than any other cyclic group. This architectural type is classified as “core-periphery.” Where architectures have multiple cyclic groups of similar size, the architecture

is referred to as “Multi-Core”. Finally, if the Core is small, relative to the system as a whole, the architecture is referred to as “Hierarchical.”

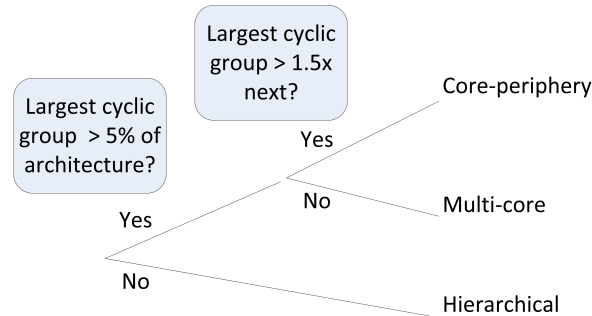


Figure 2. The architecture classification scheme.

4 Power Utility Case

We apply the method to a real-world example of a software portfolio architecture from a Nordic power utility company investigated by (Cheraghi, 2014). The data collected was stored in the company’s enterprise architecture tool/database. The subset we are investigating are applications related to what they refer to as their “smart grid” applications. These are thus mostly technical systems supporting the power distribution process, but also some administrative systems that have dependencies to and from the smart grid architecture.

4.1 Identifying the direct dependencies between the architecture components

The Power Utility dataset contains 192 software applications and 614 dependencies. The components in the architecture are applications supporting; meter reading, meter data management, contact and phone information, enterprise resource planning, enterprise asset management, work clearance and shift communication, trading, network information system, mobile workforce management, billing and service invoicing, customer self service, customer information, call center, BI data warehouse, distribution management, corporate compliance, access control, SCADA, document management, energy data management, project portfolio, sales, time and attendance capturing, et cetera. The dependencies between these components are information flows, e.g. Application A depends on information being sent from Application B.

We can represent this architecture as a directed network, with the architecture components as nodes and dependencies as links, and convert that network into a DSM.

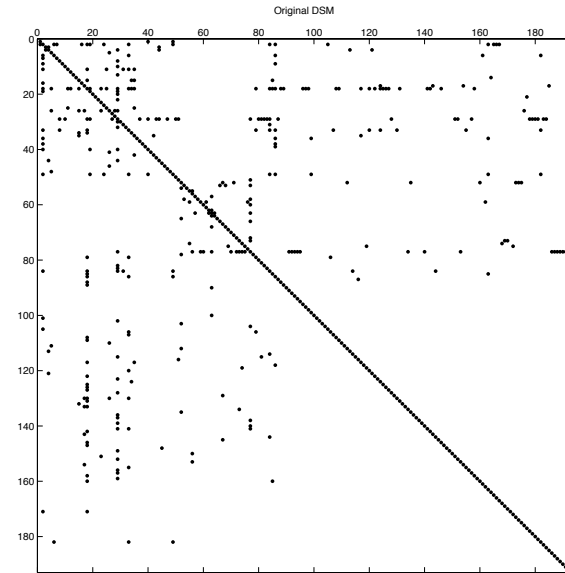


Figure 3. Input Design Structure Matrix (DSM).

From the DSM, we calculate the Direct Fan-In (*DFI*) and Direct Fan-Out (*DFO*) measures by summing the rows and columns for each element respectively. The next step is to derive the visibility matrix by raising the first-order matrix to successive powers and summing the results. Then, Visibility Fan-In (*VFI*) and Visibility Fan-Out (*VFO*) measures were calculated by summing the rows and columns for each element. See Table 1 for a sample of the Fan-In and Fan-Outs.

Table 1. A sample of Power Utility Fan-In and Fan-Outs.

Software application	<i>DFI</i>	<i>DFO</i>	<i>VFI</i>	<i>VFO</i>	<i>Class.</i>
SA1	2	1	122	1	Shared
SA2	2	1	8	1	Peripheral
SA3	3	2	121	133	Core
SA4	29	36	121	133	Core
SA5	4	4	121	133	Core
SA6	31	30	121	133	Core
SA7	2	2	3	134	Control
...	
SA103	1	2	1	2	Peripheral

To identify cyclic groups, we order the list of architectural components based on *VFI* descending and *VFO* ascending. This revealed a number of possible cyclic groups ($VFI=VFO$). By inspecting the visibility submatrices, we eliminated groups that had the same visibility measures by coincidence. After this procedure, we found the largest cyclic group (the “Core”) contained 76 components, while the second largest cyclic group contained only three. The architecture is thus defined as core-periphery (compare with the architecture classification scheme presented in Figure 2). The Core makes up 40% of the system, and is 25 times larger than the next largest cyclic group.

4.2 Classifying the components and visualizing the architecture

The next step was to classify the remainder of the components as Shared, Periphery, or Control using the definitions above. We found there were 57 Shared (30%), 76 Core (40%), 14 Peripheral (7%), and 23 Control (23%) components. Figure 4 shows the rearranged DSM, with the blocks labeled according to our classification.

5 Discussion and Research Outlook

As presented in (Baldwin et al., 2014), the hidden structure method was designed based on empirical regularity from cases investigating large complex software systems. All those cases were focused on one software system at a time, independent of its surrounding environment, analyzing the dependencies between its source files. In other words, that work considered the internal coupling of a system. In this paper, the same method is tested on the dependencies between software applications; i.e., the current work considers the external coupling between applications.

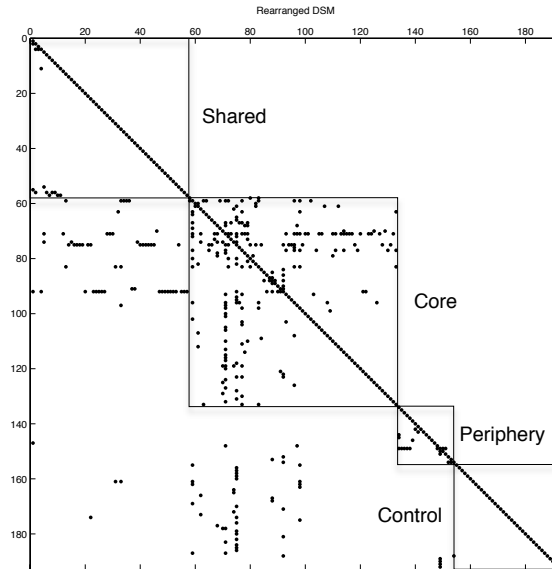


Figure 4. Power Utility rearranged DSM showing „first-order“ dependencies.

For the Power Utility case, the method revealed a hidden structure (thus presenting new facts) similar to those cases on software architecture investigated in previous studies. And the method also helped classify the architecture as core-periphery using the same rules and boundaries as in the previous cases. We have now collected a number of cases testing the method and they all seem to point in the same direction. For further cases see Lagerström et al. (2013; 2014a).

Compared to other complexity, coupling, and modularity measures, the hidden structure method considers not only the direct network structure of an architecture but also takes into consideration the indirect dependencies between applications. Both these features provide important input for management decisions. For instance, applications that are classified as Periphery or Control are probably easier (and less costly) to modify because of the lower probability of a change spreading and affecting other applications. In contrast, applications that are classified as Shared or Core are more difficult to modify because of the higher probability of changes spreading to other applications. A first study indicating this has been reported (see Lagerström et al., 2014b). This information can be used in change management, project planning, risk analysis, and so on.

From Table 1, we see that software applications 1, 2, 3, 5, and 7 all have low Direct Fan-In (DFI) and Direct Fan-Out (DFO) numbers. As such, those applications might be considered as low risk when implementing changes (compared to application 4 and 6 which have high DFI and DFO values). But if we also look at the Visibility Fan-In (VFI) and Visibility Fan-Out (VFO)

numbers, which measure indirect dependencies, we see that applications 3 and 5 both belong to the Core of the architecture. Thus any change to one of those might spread to many other applications (even though they have few direct dependencies). The same goes for application 1, which is classified as Shared. Therefore, we argue that the hidden structure method, which considers indirect dependencies, provides more valuable information for decision-making.

In our experience, we have found that many companies working with enterprise modeling have architecture blueprints that describe their application portfolio. Often, these are described using entity-relationship diagrams with boxes and arrows. When the entire application architecture is visualized using this type of model, the result is often a chaotic, messy picture that is difficult to interpret. Typically these models depict somewhat of a “spaghetti” architecture, with many applications and dependencies. This representation can be directly translated to the architect’s view DSM (cf. Figure 3). But this visualization does not really provide that much information either, other than that applications are depending on each other in a complex network. With this representation (and the entity-relationship model), we can trace a dependency between two applications, which then can be used for decision-making (compare with the discussion above on DFI/DFO versus VFI/VFO measures). However, if we instead use the hidden structure method and rearrange the DSM, as in Figure 4, we can actually see what applications are considered to be Core, Shared, Control, and Periphery. This gives us more insight about the structure of the architecture. We found that in the Power Utility Case the Core applications are spread out across the business processes and they vary between small, very specific tools to large, central ERP systems and data warehouses. Without the hidden structure method, an architect would have difficulty uncovering this type of complex architecture. The feedback from the case company was that they were surprised when presented with the results, both in terms of the Core size and architecture classification, and in terms of what systems that were found in the Core.

Measures such as the propagation cost, the architecture flow through, and the size of the core can be useful when trying to improve an architecture. Future scenarios can be compared in terms of these metrics.

A first step in future research is to test the hidden structure method with more enterprise application architectures. This will provide valuable input either supporting the method as currently constructed or with improvement suggestions for future versions. Another step would be to extend the application area. Future research could involve tests with more “complete” enterprise architecture models, considering many different types of elements such as business processes and roles, software applications and services, and databases and servers. One hypothesis is that business layer elements typically are classified as Control, infrastructure elements as Shared, and software elements as Core. This, however, remains to be tested (a first indication of this is reported in Lagerström et al., 2013). If the hidden structure method does enable the useful visualization and classification of complete enterprise architecture descriptions (including layer between different object groups), then it could be deployed to

analyze the quality of a particular architecture and possibly help improve that quality in terms of the removal or addition of elements and dependencies.

Both in the previous work by Baldwin et al. and in this case, it can be seen that many architectures have a single large Core. A limitation of the hidden structure method is that it only shows which elements (in this case, software applications) belong to the Core but does not help in describing the structure of that Core. Thus, future research might extend the hidden structure method with a sub-method for that purpose. That sub-method could help identify the elements within the Core that are most important in terms of dependencies and cluster growth. The hypothesis is that there are some elements in a Core that bind the group together or that make the group grow faster. As such, removing these elements or reducing their dependencies (either to or from them) may decrease the size of the Core and thus the complexity of the architecture. Identifying these elements also helps pinpoint where the Core is most sensitive to change.

We have also seen in previous work with enterprise application architectures that these often contain non-directed dependencies, thus forming symmetric matrices that have special properties and behave differently than those matrices containing directed dependencies. This could, for instance, be due to the nature of the link itself (as in social networks) or, as in most cases we have seen, due to imprecision in data (often because of the high costs of data collection). For companies, the primary concern is whether two applications are connected. The direction of the dependency is secondary. In one of our cases, the company had more than a thousand software applications but did not have an architecture model or application portfolio describing those applications. For that firm, collecting information about what applications it had and what those applications did was of primary importance. That process was costly enough, and consequently the directions of the dependencies between the applications were not a priority.

A lack of tool support is one reason for the high costs associated with data collection. In prior the work of Baldwin et al. (2014), the analysis of internal coupling in a software system was supported by a tool that explored the source files and created a dependency graph automatically. In the enterprise architecture domain, however, such useful practical tools generally do not exist. Consequently, data collection requires considerable time. The most common methods are interviews and surveys of people (often managers) with already busy schedules. As such, future work needs to be directed towards data collection support in the enterprise architecture domain.

For the hidden structure method to be useful in practice, it needs to be incorporated into existing or future enterprise architecture tools. Most companies today already use modeling tools like Rational System Architect, BiZZdesign Architect, TrouxView, ARIS 9, and MooD Business Architect to describe their enterprise architecture. Thus, having a stand-alone tool that supports the hidden structure method is not feasible or very cost efficient. Moreover, if the method is integrated with current tools, companies can then

perform a hidden structure analysis by re-using their existing architecture descriptions.

Last, but not least, the most important future work is to test the VFI/VFO metrics and the element classification (Shared, Core, Periphery, and Core) with performance outcome metrics, such as change cost (Lagerström et al., 2014b) and incidents or defects. Doing so will help prove that the method is actually useful in architectural work. Currently, we can argue its benefits only with respect to other existing methods.

6 Conclusions

Although our method is used only in the one case presented in this paper and few other cases previously (Lagerström et al., 2013; Lagerström et al., 2014a), the results suggest that it can reveal new facts about the architecture structure on an enterprise application level, equal to past results in the initial cases of single software system (Baldwin et al., 2014). The analysis reveals that the hidden external structure of the software applications at the Power Utility can be classified as core-periphery with a propagation cost of 44%, architecture flow through of 93%, and core size of 40%. For the Power Utility, the architectural visualization and the computed coupling metrics provide valuable input when planning architectural change projects (in terms of, for example, risk analysis and resource planning).

References

- Baldwin, C. and Clark, K. 2000. *Design Rules, Volume 1: The Power of Modularity*. MIT Press.
- Baldwin, C., MacCormack, A., and Rusnack, J. 2014. Hidden structure: Using network methods to map system architecture. *Research Policy*, Article in Press. Accepted May 19 2014.
- Barabási, A. 2009. Scale-free networks: A decade and beyond. *Science* 325, 5939, 412-413.
- Brown, N., et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research (FoSeR'10)*, 47-52.
- Cheraghi, D. 2014. *Enterprise Application Architecture: How companies can benefit from using the Enterprise Architecture Analysis Tool*. Bachelor thesis, Degree Project in Computer Science, Communication and Industrial Management, KTH Royal Institute of Technology.
- Chidamber, S. R., and Kemerer, C. F. 1994. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 20, 6, 476-493.
- Dreyfus D. and Wyner, G. 2011. Digital cement: Software portfolio architecture, complexity, and flexibility. In *Proceedings of the Americas Conference on Information Systems (AMCIS)*, Association for Information Systems.
- Hall, N. R., and Preiser, S. 1984. Combined network complexity measures. *IBM journal of research and development* 28, 1, 15-27.

- Lagerström, R., Baldwin, C., MacCormack, A., and Dreyfus, D. 2013. Visualizing and Measuring Enterprise Architecture: An Exploratory BioPharma Case. In Proc. of the *6th IFIP WG 8.1 Working Conference on the Practice of Enterprise Modeling (PoEM)*. Springer.
- Lagerström, R., Baldwin, C., MacCormack, A., and Aier, S. 2014a. Visualizing and Measuring Enterprise Application Architecture: An Exploratory Telecom Case. In Proc. of the *Hawaii International Conference on System Sciences (HICSS-47)*, IEEE.
- Lagerström, R., Baldwin, C., MacCormack, A., & Dreyfus, D. 2014b. Visualizing and Measuring Software Portfolio Architecture: A Flexibility Analysis. Risk and change management in complex systems: Proceedings of the *16th International DSM Conference*.
- MacCormack, A., Baldwin, C., and Rusnak, J. 2012. Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis. *Research Policy* 41, 8, 1309-1324.
- Opsahl, T., Agneessens, F., and Skvoretz, J. 2010. Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks* 32, 3, 245-251.
- Simon, H. A. 1962. The architecture of complexity. *American Philosophical Society* 106, 6, 467-482.
- Sosa, M., Eppinger, S., and Rowles, C. 2007. A network approach to define modularity of components in complex products. *Transactions of the ASME* 129, 1118-1129.