The Journal of Systems and Software 000 (2016) 1–13

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Technical debt and system architecture: The impact of coupling on defect-related activity

Alan MacCormack*, Daniel J. Sturtevant

Harvard Business School, Soldiers Field, Boston, MA 02163, United States

ARTICLE INFO

Article history:
Received 31 May 2015
Revised 28 May 2016
Accepted 4 June 2016
Available online xxx

Keywords:
Technical debt
Software architecture
Software maintenance
Modularity
Complexity

ABSTRACT

Technical Debt is created when design decisions that are expedient in the short term increase the costs of maintaining and adapting this system in future. An important component of technical debt relates to decisions about system architecture. As systems grow and evolve, their architectures can degrade, increasing maintenance costs and reducing developer productivity. This raises the question if and when it might be appropriate to redesign ("refactor") a system, to reduce what has been called "architectural debt". Unfortunately, we lack robust data by which to evaluate the relationship between architectural design choices and system maintenance costs, and hence to predict the value that might be released through such refactoring efforts.

We address this gap by analyzing the relationship between system architecture and maintenance costs for two software systems of similar size, but with very different structures; one has a "Hierarchical" design, the other has a "Core-Periphery" design. We measure the level of system coupling for the 20,000+ components in each system, and use these measures to predict maintenance efforts, or "defect-related activity." We show that in both systems, the tightly-coupled Core or Central components cost significantly more to maintain then loosely-coupled Peripheral components. In essence, a small number of components generate a large proportion of system costs. However, we find major differences in the potential benefits available from refactoring these systems, related to their differing designs. Our results generate insight into how architectural debt can be assessed by understanding patterns of coupling among components in a system.

1. Introduction

How do system design decisions affect the long-term costs of system maintenance? A wealth of studies has examined the topic of system design, developing insights into how decisions should be made during the development of new technological systems (Banker et al., 1993; Banker and Slaughter, 2000). This work reveals the critical impact of architectural choices in creating a design that can meet requirements along multiple, sometimes competing, dimensions of performance (e.g., functionality, speed, ease of use, reliability, upgradeability etc.). Fewer studies however, have explored how system design decisions affect performance in the *mature* stage of a system's life, where maintenance and adaptation costs are relatively more important. Given prior work argues that these costs can represent up to 90% of the total expenditures over a system's lifetime, this represents a significant gap in our knowledge (Brooks, 1975).

This topic is especially relevant to the software industry, given the dynamics of how software is developed. In particular, software systems rarely die. Instead, each new version forms a platform upon which subsequent versions are built. With this approach, today's developers bear the consequences of all design decisions made in the past (MacCormack et al., 2007). However, the early designers of a system may have different objectives from those that follow, especially if the system is successful and long lasting (something that may be uncertain at the time of its birth). For example, if early designers favor approaches that are expedient in the short term (say, to speed up time to market), later designers will bear the consequences of those decisions. Furthermore, as the external context for a system evolves over time, even design decisions that were made correctly may become obsolete and require revisiting (Kruchten et al., 2012a).

These dynamics raise an interesting question, in that for many mature systems, significant potential value might be released through design changes to reduce a system's complexity while maintaining its functionality (known as "refactoring"). Unfortunately, decision makers have little empirical data by which to evaluate the value that might be generated by such efforts

* Corresponding author. Tel.: +1 617-495-6856.
E-mail addresses: amaccormack@hbs.edu (A. MacCormack), dsturtevant@hbs.edu (D.J. Sturtevant).

http://dx.doi.org/10.1016/j.jss.2016.06.007
0164-1212/

(MacCormack et al., 2006). While a software architect might intuitively recognize the potential benefits of architectural change, senior managers typically require a robust assessment of the financial consequences of change, before funding such efforts. This need, to link software design decisions with their financial consequences, has given rise to a new metaphor, Technical Debt. It captures the *extent to which design decisions that are expedient in the short-term can lead to increased system costs in future* (Brown et al., 2010; Kruchten et al., 2012a).

In this paper, we attempt to bridge the worlds of software architecture and finance. In particular, we evaluate the relationship between system design decisions and the costs of maintenance for two software systems that represent different design "Archetypes" – one possesses a Core-Periphery design, the other possesses a Hierarchical design. We characterize system design using a network analysis technique called Design Structure Matrices (DSMs) (Steward, 1981; Eppinger et al., 1994). Our analysis allows us to calculate the level of coupling for components in each system, and thereby to identify which are more central to the design, and which are peripheral. We then analyze the extent to which components with different levels of network coupling generate different maintenance costs (i.e., in terms of the activity required to fix defects) in these systems. Our results allow us to speculate on the potential value that could be released by a refactoring effort, and to assess whether this differs between different system types.

The paper proceeds as follows. In the next section, we review the prior literature on Technical Debt and system design, focusing on work that explores how measures of system design predict the costs of maintenance. We then describe our methods, which make use of Design Structure Matrices (DSMs) to understand system structure, and measure the level of coupling between components. Next, we introduce the context for our study and describe the two systems that we analyze. Finally, we report our empirical results and discuss their potential implications for both practitioners and academia.

2. Literature review

2.1. Technical debt in software systems

In a software system, design decisions that systematically favor short-term gains over long-term costs create what is called "technical debt" (Cunningham, 1992; McConnell, 2007). These debts arise from, among other things, poor design practices, inadequate testing procedures, missing documentation, or excessively interdependent architectures (Brown et al., 2010; Seaman and Guo, 2011; Kruchten et al., 2012a, 2012b; Li et al., 2015). The interest on these debts comes in the form of increased costs for maintenance and adaptation in future. For smaller software systems, these costs may not be significant, hence not worth addressing. But as a system grows and evolves, these costs can become substantial and an increasing burden on development teams (Eick et al., 1999). Evolutions in the external context may also render past design choices outdated, creating a "technological gap" between an existing design and current requirements (Kruchten et al., 2012a). Where such technical debts exist, opportunities to create value through redesign may exist, assuming the value released exceeds the cost of taking action (Sarker et al., 2009; Schmid, 2013).

Early work in the field of technical debt focused on describing the phenomenon, and developing typologies for the different types of debt that can affect a system (Guo and Seaman, 2011; Kruchten et al., 2012a; Tom et al., 2013). For example, Kruchten et al., (2012a) propose a technical debt "landscape," which divides software improvements from a given state along two dimensions: whether they are visible or invisible; and whether they focus on maintainability or evolvability. Early empirical studies emphasized understanding the debts associated with lower-level decisions surrounding code complexity, coding style, code "smells" and poor system documentation (Brown et al., 2010). Tools based upon these approaches focus on the degree to which a software system follows, or departs from, common coding "rules" (e.g., sonarcube.org). To make the concept of Technical Debt operational, significant efforts have been made to define metrics that capture both the totality of the debt in a system, as well as the drivers of different components of this debt (Seaman and Guo, 2011; Nughoro et al., 2011).

More recently, attention has been given to how higher-level design decisions associated with a system's architecture impact technical debt (Nord et al., 2012; Kazman et al., 2015; Xiao et al., 2016). As Kruchten et al., (2012a) argue, "*more often than not*, technical debt isn't related to code and its intrinsic qualities, but to structural or architectural choices…." Of particular interest to this study, several authors have developed metrics to capture structural properties of software systems, to be used to evaluate architectural debt (MacCormack et al., 2006; 2012; Nord et al., 2012; Kruchten et al., 2012b; Kazman et al., 2015). While the specifics of these metrics differ by study, they retain a common theme in that they focus on measuring the *coupling* in a system. This is achieved by examining direct and indirect dependencies between system components.

2.2. The design of complex technological systems

A large number of studies have contributed to our understanding of the design of complex systems (Holland, 1992; Kaufman, 1993; Rivkin, 2000; Rivkin and Siggelkow, 2007). Many of these studies are situated in the field of technology management, exploring factors that influence the design of physical or information-based products (Braha et al., 2006). Products are complex systems in that they comprise a large number of components with many interactions between them. The scheme by which a product's functions are allocated to components is called its "architecture" (Ulrich, 1995; Whitney et al., 2004). Understanding how architectures are chosen, how they perform and how they can be changed are critical topics in the study of complex systems.

Modularity is a concept that helps us to characterize architecture. It refers to the way that a product's design is decomposed into parts. While there are many definitions of modularity, authors tend to agree on the concepts that lie at its heart: The notion of interdependence within modules and independence between modules (Ulrich, 1995). The latter concept is referred to as "loose-coupling." Modular architectures are loosely-coupled in that changes made to one component have little impact on others. The costs and benefits of modularity have been discussed in a stream of research that has examined its impact on complexity (Simon, 1962), production (Ulrich, 1995), platform design (Sanderson and Uzumeri, 1995), process design (MacCormack et al., 2001) process improvement (Spear and Bowen, 1999) and industry structure (Baldwin and Clark, 2000).

Studies that seek to measure the modularity of technical systems typically focus on capturing the level of coupling that exists between different parts of a design. In this respect, the most prominent technique comes from the field of engineering, in the form of the Design Structure Matrix (DSM). A DSM highlights the inherent structure of a design by examining the dependencies that exist between its constituent elements in a square matrix (Steward, 1981; Eppinger et al., 1994). These elements can represent design tasks, design parameters or the components that comprise the system. DSMs have also been used to explore the degree of alignment between task dependencies and project team communications (Sosa et al., 2004). Recent work has extended this methodology to show how dependencies can be extracted automatically from software source code and used to understand system design

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

(Baldwin et al., 2014; MacCormack et al., 2006). Metrics that capture the level of coupling for a system's components can be calculated from a DSM, and used to analyze designs (MacCormack et al., 2012). We adopt this approach in our work.

2.3. Software design, modularity and maintenance costs

The formal study of software modularity began with Parnas (1972) who proposed the concept of "information hiding" as a mechanism for dividing code into modular units. This required designers to separate a module's internal details from its external interfaces, reducing the coordination costs involved in system development and facilitating changes to modules without affecting other parts of the design. Subsequent authors built on this work, proposing metrics to capture the level of *coupling* between modules and *cohesion* within modules (e.g., Selby and Basili, 1988; Dhama, 1995). Modular designs have low coupling and high cohesion. This work complemented studies that sought to measure the complexity of a design for the purposes of predicting developer productivity (e.g., McCabe 1976; Halstead, 1977). Whereas measures of complexity focus on individual components, measures of software modularity focus on the relationships *between* components. These two concepts are complementary.

Studies seeking to link measures of system design with maintenance costs tend to focus on predicting the cost and frequency of changes across systems. For example, Banker et al. (1993) examined 65 maintenance projects across 17 systems and found costs increased with complexity, as measured by the average "procedure" size and number of "non-local" branching statements. Kemerer and Slaughter (1997) examined modification histories for 621 software modules and found enhancement and repair frequency increased with module complexity, as measured by the number of decision paths normalized by size (McCabe, 1976). Banker and Slaughter (2000) examined three years of modification data from 61 applications and found total modification costs increased with application complexity, as measured by the number of input/output data elements per unit of functionality. And Barry et al. (2006) examined 23 applications over 20 years and found an increase in the use of standard components (a proxy for modularity) was associated with a decline in the frequency and magnitude of changes.

2.4. Intended contribution of this study

While the studies above have made major contributions to our understanding of the impact of design decisions on system costs, they do not consistently address several issues that are key to assessing the technical debt due to system architecture, and hence the potential value of refactoring. First, most focus on the *complexity* of the components in a system, but do not capture the *coupling* between components, the primary driver of modularity. Second, studies that do measure coupling tend to capture only *direct* linkages between components (e.g., Chidambur and Kemerer, 1994) so do not account for the potential for changes to propagate via chains of *indirect* dependencies. Third, those that recognize the role of indirect dependencies often use metrics from social network theory (Dreyfus, 2009), which do not capture the *asymmetric* nature of coupling in a technical system (i.e., a component may "depend upon" many others, but be "depended upon" by no others). Finally, many use a cross-sectional research design in which the system is the primary unit of analysis. Hence we do not know if tightly coupled components *within* these systems incur greater costs than others – a necessary condition to create value through refactoring (i.e., by reducing the level of coupling for some components).

To address the first concern, we characterize design in terms of the *coupling* between components, while controlling for the internal complexity of components. To address the second concern, we capture data on both the direct and *indirect* coupling of components, to account for the potential propagation of changes through a system. To address the third concern, we capture data on the *direction* of dependencies between components, to tease apart the impact of in-degree versus out-degree coupling (Martin, 2002). To address the fourth concern, we explore the relationship between coupling and maintenance cost at the *file* level, allowing us to determine if there are systematic differences in cost across components that can be reduced by a refactoring effort. Finally, to increase robustness, we conduct analyses on two systems with very different designs, to increase our confidence in the ability to generalize the results.

We note several recent studies adopt similar techniques to ours to understand architectural debt, based upon measuring the coupling between components in a DSM, albeit with differing approaches to the extraction of file level interdependencies and sorting of files into categories or "design spaces"(Kazman et al., 2015; Xiao et al., 2014; 2016). We view this convergence on a dominant approach to architectural debt analysis as a positive development, and note these perspectives produce complementary insights.

3. Research methodology

3.1. Analyzing software systems using design structure matrices

We build on prior work that describes how to apply Design Structure Matrix techniques to the analysis of software system architecture (Baldwin et al., 2014; MacCormack et al., 2006, 2012; Sosa et al., 2013). These methods rely upon capturing the dependencies that exist between source files (i.e., components) and calculating measures of coupling between these files, both for each component and for the system as a whole.

There are many types of dependency between source files in a software system (Shaw and Garlan, 1996; Dellarocas, 1996). To capture important dependencies, we use a commercial tool called a "Call Graph Extractor" (Murphy et al., 1998), which takes source code as input, and outputs dependencies, including those that exist between functions, classes and global data.[1] [2] Rather than develop a call-graph extractor, we tested several commercial products that could process source code written in different languages, capture indirect dependencies (i.e., those that flow through intermediate files), run in an automated fashion and output data in a format that could be input to a DSM. A product called Understand, distributed by Scientific Toolworks, was chosen for use.[3]

The dependency information output from a call-graph extractor can be aggregated at the source file level and displayed in a DSM. For example, if FunctionA in SourceFile1 "calls" FunctionB in SourceFile2, then SourceFile1 depends upon (or "uses") SourceFile2; hence a binary dependency is marked in location (1, 2) in the DSM. This highlights the fact that a change to SourceFile2 might impact SouceFile1. Critically however, this does not imply

[1] Many types of dependencies can be explored in software. The systems that we analyze are written in C/C++. We looked at important relationship types in C/C++ (i.e., those which might cause changes to propagate between files) including function calls, method calls, class inheritance, global data modification, parameter setting, use of a typedef, use of a class instance, header file inclusion, and overrides.

[2] Dependencies can be extracted statically (from the source code) or dynamically (when the code is run). We use a static call extractor because it uses source code as input, does not rely on program state (i.e., what the system is doing at a point in time) and captures the system structure from the designer's perspective.

[3] See https://scitools.com. Note that the company updates its products periodically, and there may be minor differences in the results obtained when using different versions of this tool on the same code base.

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

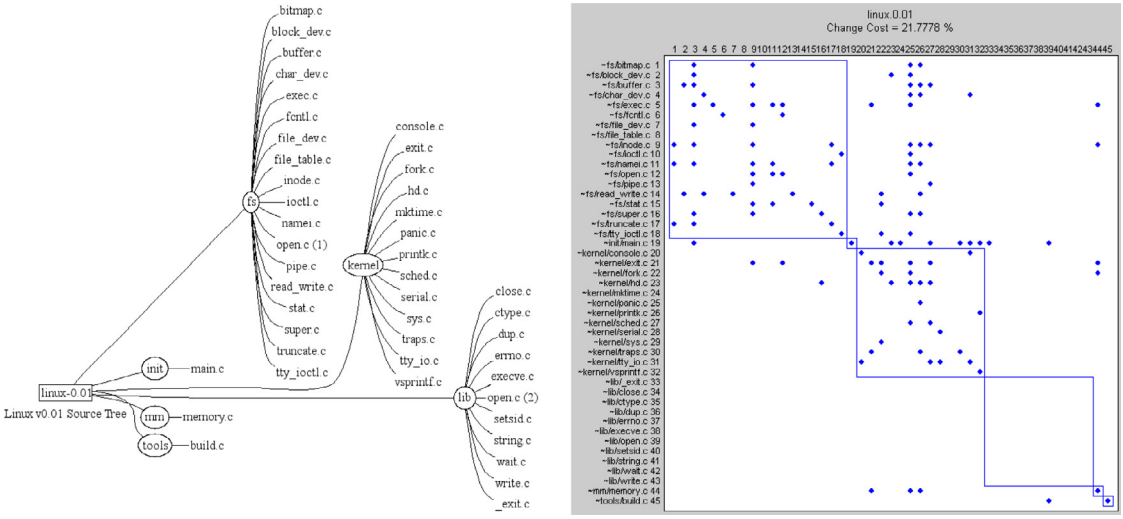

Fig. 1. Directory structure and architectural view of linux v0.01.

Fig. 2. Example in graphical and DSM form (direct and visibility DSMs).

that SourceFile2 depends upon SourceFile1; the dependency is not symmetric unless code in SourceFile2 calls a function in SourceFile1. Dependencies are directed, and DSMs provide a representation of the directed graph.

We display data in a DSM using the *Architectural View*. This view groups each source file into a series of nested clusters defined by the directory structure, with boxes drawn around each layer in the hierarchy. To illustrate, we show the Directory Structure (left) and Architectural View (right) for Linux v0.01 in Fig. 1. This system comprises six subsystems, three of which contain only one component and three of which contain between 11–18 components. In the Architectural view, each "dot" represents the existence of one or more dependencies between two components (i.e., source files).

3.2. Measuring component coupling in software systems

In order to assess system structure, we measure the degree to which components are coupled to each other. In particular, we capture all *direct and indirect* dependencies a component possesses with other components, a concept known as "Visibility" (Sharman and Yassine, 2004; Warfield, 1973). To account for the fact that software dependencies are asymmetric we calculate separate visibility measures for dependencies that flow into a component ("Fan-In") versus those that flow out from it ("Fan-Out").

To illustrate, consider the example depicted in Fig. 2 in graphical and DSM forms. Element A depends upon ("uses") elements B and C. In turn, element C depends upon element E, hence a change to element E may have a *direct* impact on C, and an *indirect* impact on A, with a "path length" of two. Similarly, a change to element F may have a direct impact on element E, and an indirect impact on elements C and A, with a path length of two and three, respectively. The Visibility DSM displays all of the direct and indirect dependencies between components in the system. It is found by computing the transitive closure of the direct dependency DSM. (By definition, each component depends on itself; hence we insert dependencies on the diagonal of the visibility DSM.)

Measures of component coupling are derived from the Visibility DSM. Visibility Fan-In (VFI) is obtained by summing down the columns for each component; Visibility Fan-Out (VFO) is obtained by summing along the rows for each component. The density of the Visibility DSM, defined as the system's Propagation Cost, can be used to compare the mean level of coupling across different systems (MacCormack et al., 2006). Intuitively, this measure captures the fraction of a system's elements that could potentially be affected, when a change is made to a single element chosen at random.

3.3. Dividing components into categories and classifying system type

The components of a software system can be divided into different categories based upon their levels of coupling, revealing structural patterns that may not be apparent from the Architectural view. In particular, Baldwin et al. (2014) describe how sorting components by their levels of fan-in and fan-out visibility helps to reveal both design *cycles* and design *hierarchy*.

Design cycles occur when a group of components are mutually interdependent, meaning that each depends on all others, either directly or indirectly. For example, if file A depends on file B, and file B depends on file A, these components are part of a cycle. Changing either one could potentially affect the performance of the other. The components in a cycle possess the same levels of both Fan-In and Fan-Out visibility, given that they are interconnected. Design cycles can therefore be detected by inspection of the visibility measures for all the components in a system

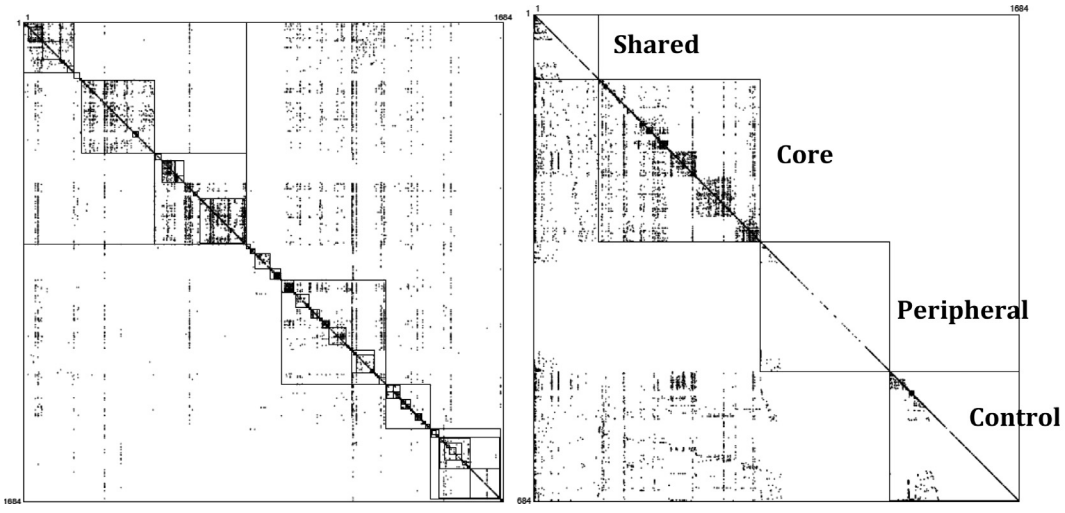

Fig. 3. Example system showing architectural (left) and "core-periphery" (right) views.

(Baldwin et al., 2014). The largest design cycle (i.e., largest "cyclic group") is referred to as the system's Core.

Design hierarchy refers to a specific ordering of components in a system, such that dependencies flow in a uniform direction. For example, if file A depends on file B, and file B depends on file C, there is a natural ordering of these components. Specifically, all components depend on C, either directly or indirectly. Hence file C belongs at the "top" of the hierarchy, given changes to C can potentially impact all other files. Similarly, file A is at the bottom of the hierarchy, given it can be changed with no impact on other files.

Prior work shows that software systems can be classified into different types, based upon the relative size of the largest cyclic group of components (Baldwin et al., 2014). *Core-Periphery* systems have a large, dominant, cyclic group of components (the "Core") comprising more than 5% of the system. Multi-core systems also possess a large cyclic group comprising more than 5% of the system, but in addition, possess other cyclic groups of comparable size. Finally, *Hierarchical* systems possess either no cyclic groups, or small cyclic groups, the largest of which do not exceed the 5% threshold.

The components in a system can be divided into groups, based upon their levels of coupling, and used to analyze system structure and performance. In Core-Periphery systems, components are divided into four groups based upon comparisons to the visibility of *Core* components (i.e., those in the largest cyclic group). *Shared* components have higher VFI, but lower VFO; *Control* components have higher VFO, but lower VFI; and *Peripheral* components have both lower VFI and lower VFO.

Fig. 3 shows how components of a system can be reordered in a DSM, based upon these four groupings, to reveal hidden structure (i.e., design cycles and design hierarchy). On the left, the Architectural view shows the nested directory structure of the system, in alphabetical order (the boxes indicate directories and subdirectories). Files are listed both down the vertical and across the horizontal axes. Each "dot" represents a dependency between two files (designated by the row and column). On the right, the Core-Periphery view re-arranges files into four categories – Shared, Core, Periphery, and Control – by comparing their visibility measures to the visibility measures of the largest cyclic group of components. The resulting DSM has a "lower-diagonal" form (Steward, 1981). That is, most dependencies are below the diagonal. Dependencies that remain above the diagonal denote the presence of cyclic dependencies (e.g., A calls B and B calls A).

In Core-Periphery systems, dependencies flow from Control components through Core components, to Shared components. Peripheral components lie outside the main "flow of control" (i.e., they may depend on Shared components but not on Core or Control components). Prior work suggests many systems have a Core-Periphery design, with the Core encompassing, on average, 16% of components (Baldwin et al., 2014).

In Hierarchical systems, the relatively small size of the Core means that component groupings based upon the four categories described above are unbalanced in terms of size, creating challenges for statistical analyses. Furthermore, if a system contains several smaller cycles of comparable size, the identity of the largest cyclic group may change from version to version, making the classification of components unstable as a system evolves. To address these issues, the components in hierarchical systems can be divided into four groups based upon comparisons to the *Median* levels of both Fan-In and Fan-Out Visibility (Baldwin et al., 2014). In such systems, components with high VFI and high VFO are known as *Central* components (to distinguish them from Core components). The remaining three groups follow the logic above, and are called *Shared-M, Control-M* and *Peripheral-M*. Fig. 4 displays DSMs for a hierarchical system, with the left side showing the Core-Periphery view (constructed as described above), and the right side showing the Median view (where components are classified according to the median level of fan-in and fan-out visibility, but ordered in the same way as before).

4. Empirical data and measurement approach

4.1. Source of empirical data

We identified two firms that developed large software systems of similar size, but which possessed different system designs. One system possessed a "Hierarchical" design (system H), the other possessed a "Core-Periphery" design (system C). In order to analyze the relationship between design decisions and maintenance costs, we identified systems where we could track maintenance efforts to specific source files. Both the firms in our study used version control systems in which developers submit "patches" against files to implement changes to the software. Both firms also employed bug-tracking systems to store information about defects from the time they are found to the time they are fixed. Critically, both firms retained a consistent link between version control systems and bug-tracking systems, allowing us to track the relationship between a defect and the specific files changed to fix it. Finally, for both sys-

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

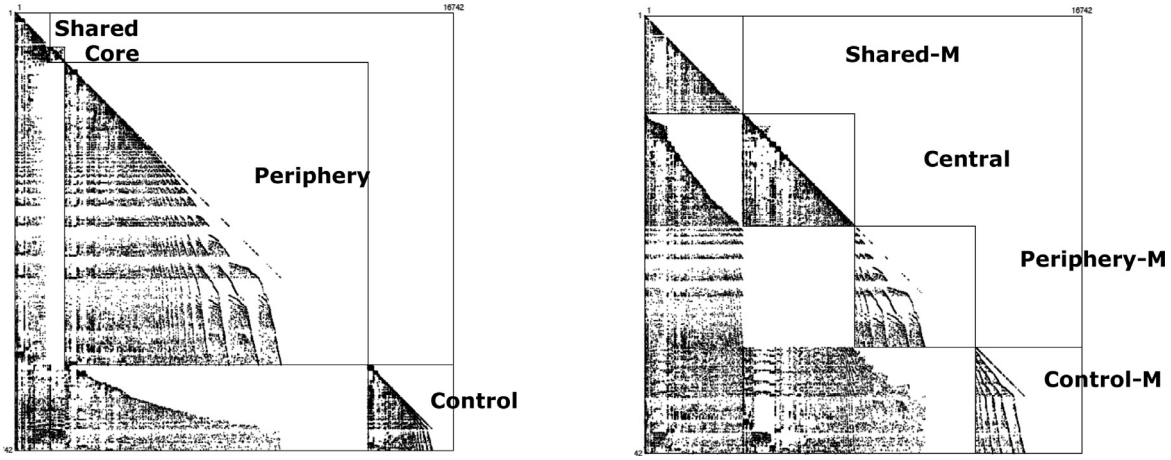

Fig. 4. Hierarchical system showing "core-periphery" (left) and "median" (right) views.

Fig. 5. Example of activity to fix a defect.

tems, we had access to the total number of hours spent performing maintenance activities.[4]

Both system H (the hierarchical system) and system C (the Core-Periphery system) had been in development for over 10 years and comprised around 20,000 files – primarily a mixture of C and C++ source files and header files. We examined maintenance logs for both systems for a three-year period after a major release in mid-2007. We captured only development activity that was *corrective* in nature.[5] Scripts to extract file-based metrics, and information from the bug-tracking and version control systems were developed and run against the source code, version control and defect-tracking systems. These scripts were developed to encrypt or anonymize sensitive information.

4.2. Measuring "defect-related activity"

A software defect goes through multiple stages in its life. It is introduced, possibly released into the field, discovered, corrected, and then that correction is validated. Development activity that arises to correct defects can be measured after-the fact by examining bug tracking and development logs. For both systems, the developers maintaining them used processes to ensure that corrective code changes made to each file were traceable to the entries in a bug tracking system. We used these logs to compute a metric called "defect related activity," for each file over the period. Fig. 5 shows an example of defect-related activity for a defect in system H.

In this example, a defect was corrected by modifying two source files (A and B). When the defect was discovered, "User 12" submitted identifying information into the bug tracking system. "User 15" then performed the work to repair the bug, submitting patches to two files (A and B) on two separate occasions to correct the issue. The new version of the software was then validated and merged into production by "User 17." To maintain consistency across both systems, we measure only the *repair* activity to fix a defect (i.e., we excluded the tasks of submission and validation). In this example, files A and B each experienced 2 pieces of 'defect related activity' to fix this defect. (Note the amount of defect related activity is always higher than the number of defects, given multiple files may be patched, on multiple occasions, to fix it). In our work, we correlate the defect-related activity for each file with measures of coupling for the file, as well as a variety of control variables shown to predict maintenance efforts in prior studies.

4.3. Descriptive statistics

Table 1 contains descriptive data for each system. While both systems contain a similar number of source files, they differ in the number of source lines of code. System H has around 5 million lines of code with a mean of 246 lines per file. System C has over 8 million lines of code, with a mean of 435 lines per file. Hence in our analyses, we note that it is important to control for performance differences driven by file size.

In system H, we identified 317 bugs that could be traced directly to files present in the 2007 release, accounting for 2909 pieces of defect-related activity. Of the 20,270 files in system H,

[4] See Appendix A for details on how data was extracted for each system.

[5] New feature development for system H was undertaken in a separate "branch" of the code base where development of the next major release proceeded independently. New feature development for system C was excluded by examining only defects that could be traced to the mid-2007 release of the software.

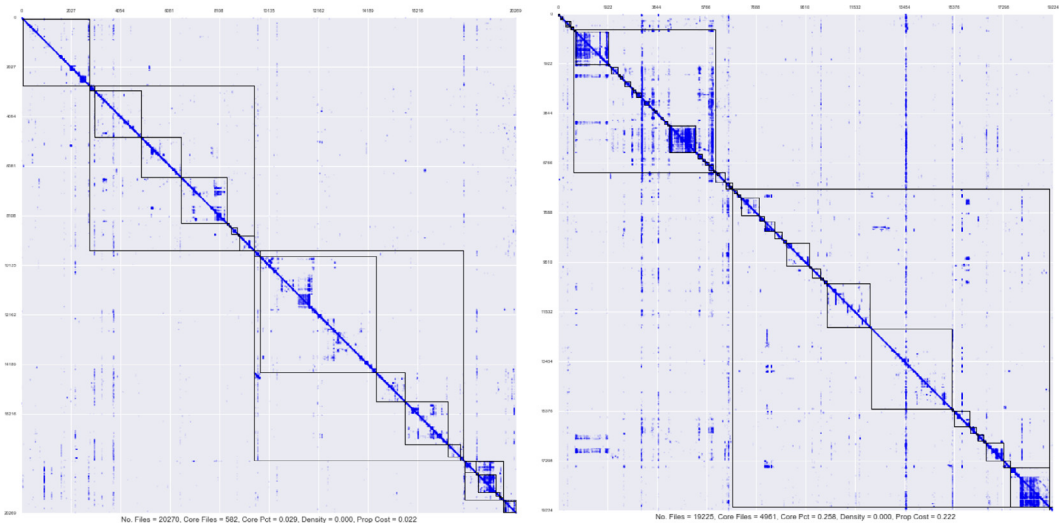

Fig. 6. The architectural views for system H (left) and system C (right).

Table 1
Descriptive statistics for systems.

	System H	System C
Files in System	20,270	19,225
Total Lines of Code	4,989,956	8,371,591
Defects in Period	317	3510
Files with Defects	1107	3249
Defect Related Activity	2909	11,228

Table 2
Distribution of files by core-periphery category.

	System H	System C
Shared	1319	558
Core	582	4961
Peripheral	11,579	1949
Control	3262	9840
Isolates	3528	1917
TOTAL FILES	20,270	19,225

1107 (5.6%) experienced some defect related activity in the period. In system C, we identified 3510 bugs that could be traced directly to files present in the 2007 release, accounting for 11,228 pieces of defect-related activity. Of the 19,225 files in system C, 3249 (16.9%) experienced some defect related activity in the period. Hence files in system C were three times more likely to experience defects as files in system H.

5. Empirical results

5.1. Characterizing system architecture

Fig. 6 displays the direct dependency DSM for both systems, displayed using the Architectural View. The density of the DSM for system H (0.02%) is lower than for system C (0.04%). However, it is difficult to know from this data alone whether these two systems possess different structural properties, in terms of the level and pattern of *indirect* dependencies between components. This question is answered by propagating the matrix of direct dependencies to identify the pattern of indirect linkages between files.

In Fig. 7, we show the Core-Periphery View for both systems. (We split out components that are not connected to *any* others in each system, which we call Isolates). In this view, the differences in system structure become clear. In the Core-Periphery view, above diagonal points denote cyclical dependencies (Sosa et al., 2007). System H has very few cyclical dependencies and a small Core (2.9% of the system). It is a Hierarchical system. System C has a large number of cyclical dependencies and a very large Core (25.8% of the system). It is a Core-Periphery system. We note also the difference in *Propagation Cost* between systems. System H has a propagation cost of 2.2%, meaning each component is connected, on average, to only a small number of other components. System C has a propagation cost of 22.2% meaning that each component is connected, on average, to many others. Compared to similar metrics reported in other studies of software systems, these figures lie at opposite ends of a continuum (Baldwin et al., 2014). The components in system H are very loosely-coupled, on average, compared to other systems. In contrast, the files in system C are very tightly-coupled, on average, compared to other systems.[6] These two systems represent a "matched pair" with which to explore the relationship between coupling and defect-related activity.

Table 2 shows the distribution of components across the five categories (Shared, Core, Periphery, Control and Isolates). As prior work suggests, the allocation of components to categories in system H is very unbalanced, given it is Hierarchical in nature, and has only a small Core. In fact, 75% of system components are classified as Peripheral or Isolates, which lie outside the main flow of control for the system.

Given this fact, we conduct further analysis for this system using *Median* levels of visibility, as recommended in prior work (Baldwin et al., 2014). For statistical purposes, the use of medians ensures a more equal distribution of files to different categories, which increases the robustness and generalizability of results. Fig. 8 shows a comparison of the Core-Periphery and Median views for system H.

5.2. Descriptive analysis

Table 3 shows the level of defect related activity (DRA) for each system, split by the four categories associated with different levels of coupling. Our primary hypothesis is that files with higher lev-

[6] Note that both systems exhibit a "small world" phenomenon common to many natural and engineered systems (Barabasi, 2009). Every pair of nodes that is connected via a chain of dependencies can be found in fewer than 7 steps (i.e., the longest path length between any two connected files is only 7 steps).

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

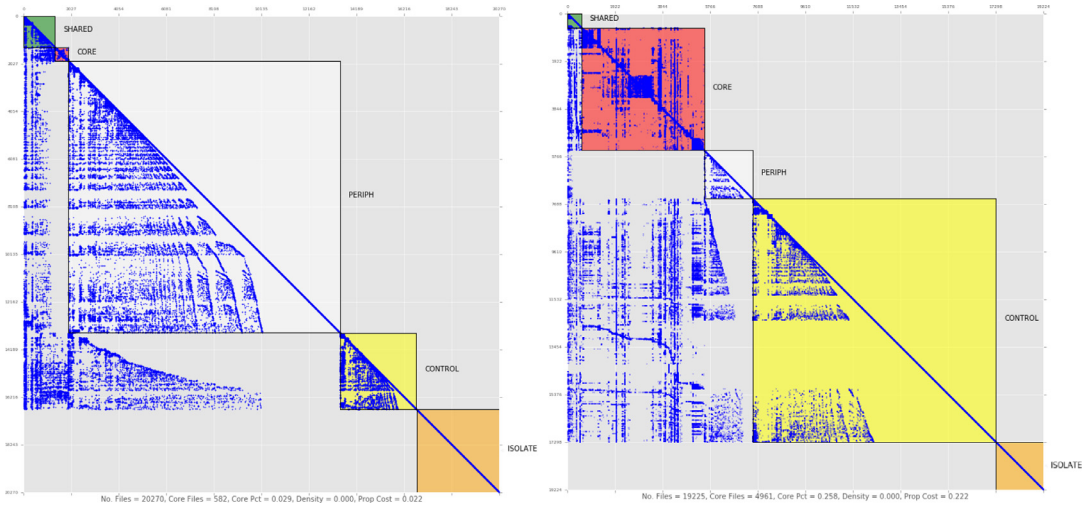

Fig. 7. The core-periphery views for system H (left) and system C (right).

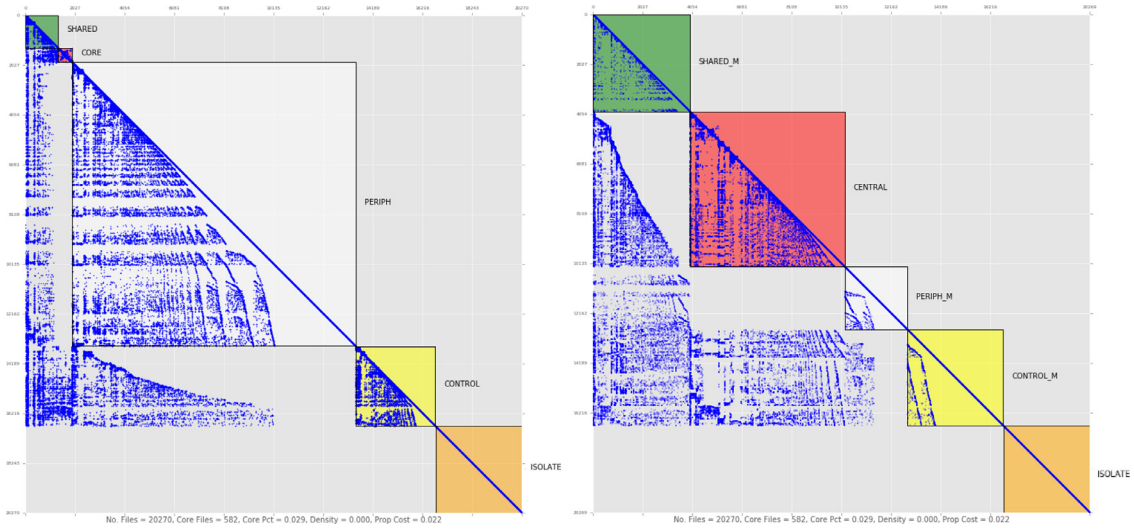

Fig. 8. The core-periphery (left) and median (right) views for system H.

Table 3
Defect-related activity by category for each system.

System H	Files	DRA	% DRA	Files with DRA	% Files with DRA
Isolate	3528	7	0.24%	7	0.2%
Periph-M	2555	25	0.86%	15	0.6%
Shared-M	3948	304	10.45%	126	3.2%
Control-M	3918	540	18.56%	189	4.8%
Central	6321	2033	69.89%	770	12.2%
All Files	20,270	2909	100.00%	1107	5.5%
System C	**Files**	**DRA**	**% DRA**	**Files with DRA**	**% Files with DRA**
Isolate	1917	210	1.87%	56	2.9%
Periph	1949	189	1.68%	113	5.8%
Shared	558	53	0.47%	36	6.5%
Control	9840	3811	33.94%	1556	15.8%
Core	4961	6965	62.03%	1488	30.0%
All Files	19,225	11,228	100.00%	3249	16.9%

els of coupling have a higher probability of experiencing defects, and generate a greater amount of defect-related activity, in both hierarchical and core-periphery systems. The descriptive data below support these hypotheses.

Consider first, the data for system H. Peripheral-M and Isolate files comprise 30% of the system, but experience only 32 of the 2909 pieces of defect related activity (i.e., 1.1%). Furthermore, defect-related activity is isolated to only 22 of the files in these two categories (0.4%), meaning over 6000 files experience *zero defects*. This contrasts with components in the 'Central' category, which generate 2033 pieces of defect related activity (70% of all such activity) despite comprising only 31% of files. Furthermore, over 12% of files in the Central category experience a defect during the study period.

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

Table 4
The *probability* of experiencing a defect (logit model).

Logit System H	1	2
Intercept	−3.26121***	−5.39875***
C-File	0.27725***	0.34005***
Lines of Code	0.00056***	0.00042***
Cyclomatic Complexity	0.00795***	0.00448***
Isolate	–	−1.06850***
Shared-M	–	1.88786***
Control-M	–	1.92061***
Central	–	2.98000***
Psuedo R-Squared	0.057	0.148
Logit System C	**1**	**2**
Intercept	−2.68624***	−3.28895***
C-File	1.01408***	0.88257***
Lines of Code	0.00028***	0.00022***
Cyclomatic Complexity	0.01958***	0.01481***
Isolate	–	−0.90617***
Shared	–	0.24153
Control	–	0.69669***
Core	–	1.34561***
Psuedo R-Squared	0.088	0.122

Table 5
The *amount* of defect related activity (OLS regression model).

OLS Regression System H	1	2
Intercept	0.02959***	−0.01078*
C-File	0.00463***	0.01181***
Lines of Code	0.00010***	0.00009***
Cyclomatic Complexity	0.00107***	0.00085***
Isolate	–	−0.00596
Shared-M	–	0.03319***
Control-M	–	0.02209***
Central	–	0.10078***
Adjusted R-Squared	0.059	0.080
OLS Regression System C	**1**	**2**
Intercept	0.05266***	−0.00565
C-File	0.13997***	0.11559***
Lines of Code	0.00004***	0.00004***
Cyclomatic Complexity	0.00561***	0.00488***
Isolate	–	−0.03523**
Shared	–	0.01835
Control	–	0.04167***
Core	–	0.24686***
Psuedo R-Squared	0.106	0.140

Similar results are seen in system C. Peripheral and Isolate files account for 20% of this system's components, yet only 3.6% of defect-related activity. In contrast, Core files account for 26% of system components, but 62% of defect-related activity. Over the period, 30% of Core files experience a defect, compared to 5.8% for peripheral files.

5.3. Predictive analysis

While the descriptive data above provide illustrations of the potential impact of coupling in these systems, they do not show that these measures are statistically significant. In particular, it might be the case that the overall pattern of activity we see in each category is driven by one or two outliers that experience large levels of defect-related activity. Furthermore, it is possible that the large differences in activity might be related instead to differences in the size or complexity of files in each category, and not their coupling. Hence, we develop statistical models for each system, to understand whether component coupling is a predictor of defect-related activity.

Table 4 shows a series of logit models predicting the *probability* of a file experiencing a defect over the study period in each system. Table 5 shows a series of OLS regression models predicting the *amount* of defect-related activity in each system. (We use the log of defect-related activity, given this measure is skewed, and truncated at zero). In these models, we control for the number of lines of code (LOC) in a file, as well as its cyclomatic complexity. With respect to the latter, we calcuate the cyclomatic complexity for all functions and methods within a file, and then use the *maximum* figure observed within a file. We also include a control for C/C++ files (i.e., those with the extension .c or .cpp) as opposed to header files (i.e., .h) to control for the fact that the latter play a different role in a software system, and are typically less complex. In each table, model 1 provides results for control variables, and model 2 adds predictor variables (i.e., Core/Central, Shared, Control, etc.). In model 2, the baseline is for a Peripheral file (i.e., dummy variables are included for Shared, Central/Core, Control and Isolate files).

The control variables are positive and significant for both models and both systems. Files with more lines of code, greater maximum cyclomatic complexity, and C/C++ files have a higher probability of defects, and experience a greater amount of defect-related activity. The models based upon controls alone explain 5.7% of the variation in the *likelihood* of experiencing a defect for system H, and 8.8% of the variation in the *likelihood* of experiencing a defect for system C. The models based upon controls alone explain 5.9% of the variation in the *amount* of defect-related activity for system H, and 10.6% of the variation in the *amount* of defect-related activity for system C.

In models that add predictor variables to the controls, we find a strong association between files that possess high levels of coupling and both i) the likelihood of experiencing a defect, as well as ii) the overall amount of defect-related activity. In system H, Shared-M, Control-M and Central files all have a higher probability of experiencing defects than Peripheral files, and all experience more defect related activity. Peripheral files with *zero* external coupling (Isolates) have a lower probability of experiencing defects than other peripheral files, but similar levels of defect-related activity. In system C, the results are broadly similar, with one exception. In this system, the dummy for *Shared* files is not significant in either model. Hence Shared files perform similarly to Peripheral files, with respect to the probability of experiencing a defect, and the overall amount of defect-related activity. This lends some support to the notion that the use of shared files represents good design practice, at least in Core-Periphery systems.

5.4. Financial analysis

We obtained resource-allocation data for the 3-year period between May 2007 and April 2010. We had access to the total hours devoted to corrective maintenance for both systems as a whole. We used a standard cost of $100/hour to estimate the fully loaded cost of this activity. We then distributed this cost across categories, based upon the level of defect-related activity observed in each (the assumption being that, on average, each piece of defect-related activity "costs" the same). We divided the total maintenance costs in each category by the total number of lines of code in each. The results, in Table 6, suggest a cost per line of code per year, by category, to maintain each system.[7]

For system H, we find each line of code in a Central file costs over 15 times as much to maintain as a line of code in a Peripheral-M file. For system C, we find each line of code in a Core file costs around three times as much as a Peripheral file. We note the Core-Periphery system costs on average, three times more per

[7] Note in this analysis, we have combined the activity in Isolate and Peripheral categories.

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

Table 6
Cost per line of code by component category.

System H	LOC	DRA	Cost of Activity	Cost/LOC	Cost/LOC/Year
Peripheral-M	617,621	32	$86,480.30	$0.14	$0.05
Shared-M	637,536	304	$821,562.87	$1.29	$0.45
Control-M	1,520,086	540	$1459,355.10	$0.96	$0.33
Central	2,214,097	2033	$5494,201.72	$2.48	$0.86
All Files	4,989,340	2909	$7861,600.00	$1.58	$0.55
System C	LOC	DRA	Cost of Activity	Cost/LOC	Cost/LOC/Year
Peripheral	700,616	399	$1432,916.71	$2.05	$0.71
Shared	82,475	53	$190,337.31	$2.31	$0.80
Control	3,825,163	3811	$13,686,329.78	$3.58	$1.25
Core	3,763,337	6965	$25,013,195.20	$6.65	$2.31
All Files	8,371,591	11,228	$40,322,779.00	$4.82	$1.68

line of code per year to maintain, compared to the Hierarchical system. Care must be taken in interpreting these results however, given these figures, unlike the statistical models presented earlier, do not control for differences in cyclomatic complexity across categories.

5.5. Projecting the potential cost savings from refactoring efforts

Our results suggest that system architecture decisions are associated with significant differences in maintenance costs. This gives rise to the question of whether refactoring efforts might be worthwhile, which have the aim of reducing the level of coupling for some components, and hence lowering costs. To speculate on the potential benefits of such actions, we draw on prior work that highlights the impact of system redesign efforts in terms of the ability to reduce component coupling. In particular, MacCormack et al. (2006) and Baldwin et al. (2014) show that a refactoring of the Mozilla web browser software led to a reduction of 50% in the number of highly coupled Core components.

With respect to system H, we noted earlier that the largest cyclic group of components in this system (i.e., the Core) comprised only 582 components. In such a system, the benefits from a refactoring effort that halved the number of such components would appear to be small. Alternatively, one might take the view that such an effort could reduce the coupling level of *Central* components, to the point where 50% of these components experience the same cost as Shared-M or Control-M components (i.e., a reduction from $0.86 per LOC per year to between $0.33-$0.45 per LOC per year). Assuming 1.1 million lines of code were affected (see Table 6), such a move might reduce annual maintenance costs by around $500,000 (i.e., 6.5% of total costs).

For system C, assuming a refactoring effort achieved similar results to that observed in Mozilla, 50% of the Core components could be moved to either the Shared or Control categories and hence experience a substantial decline in cost (i.e., from $2.31 per LOC per year to between $0.80-$1.25 per LOC per year). Assuming 1.9 million lines of code were affected (see Table 6), such a move might reduce annual maintenance costs by around $2400,000 (i.e., 6.0% of total costs). While these estimates are clearly speculative, the difference in magnitude between the two systems is notable. We conclude that in these two equally sized systems, refactoring appears to present a much larger opportunity for the Core-Periphery system than it does the Hierarchical system.

We note that our projections represent a broad, top down approach to assessing the potential value of refactoring, based upon the assumption that refactoring could achieve similar outcomes to prior observed efforts. We do this merely to demonstrate that techniques such as these can be used to identify places where large amounts of value creation might be possible via refactoring. However, for systems of the size we analyze, the *realization* of such value is unlikely to happen via a single major refactoring event (complete replacement is not a viable strategy for the systems in our study). Rather, this value is likely to be realized only via long-term efforts to modularize the system. Our methods provide guidance for where one might start in such a process – for example, gradually moving pieces of common functionality out of the Core, and structuring these as, say, Shared components. Indeed, such an effort was begun in system C prior to the end of the period we analyze. A senior developer managing this effort informed us:

> *"The system had grown in a somewhat uncontrolled fashion for two decades. During this time the organization had grown substantially too. After a while, it became increasingly apparent that we had architectural issues that were making it hard for teams to be productive and leading to quality problems. So we had to ask some hard questions about how to regain control. A complete rewrite was tempting but would have been impossible - too disruptive to our release schedule and our customers, and a huge investment with an uncertain outcome. So we had to refactor, but we could only do this in an incremental fashion."*

> *"We kicked off a "componentization" effort to refactor the architecture, with the explicit goal of breaking links, driving modularity into the codebase, and getting a handle on dependencies and APIs. We are using frameworks, assessment and visualization tools to give each developer team the information it needs to understand the system and how to break it apart. Developers can now take local action and coordinate with neighbors to clean up regions of the code. When every team finishes, we'll be left with a system that is much more architecturally sound. The effort is ongoing - it'll never really be complete because business needs and technologies change - but we've made good progress."*

> *"In terms of outcomes, a number of teams whose code has been broken out into modular components have reported that their productivity has more than tripled. Side effects happen less often, compile times are down, and testing and integration are much easier. Developers can focus on their actual jobs, without being as concerned about what's going on in the rest of the codebase."*

We note that in order to assess whether it is worth paying down architectural debt by refactoring a design, managers must also assess the costs of these efforts (i.e., the costs of either a single, large refactoring event, like was done for the Mozilla browser, or frequent, smaller, ongoing, componentization efforts, as is underway for system C). Without such data, we cannot know the true return on investment (ROI) from refactoring. We believe our methods can be extended to produce such estimates, which is a subject of ongoing research. However, we note recent research has made

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

progress in this field, evaluating the costs and benefits of refactoring based upon an assessment of the before and after ("desired") states of a design using network analysis techniques similar to our own (Kazman et al., 2015). We believe the ideal approach to understanding refactoring ROI likely combines data from both a top down approach (i.e., using empirical data on the structural impact of past refactoring efforts) as well as a bottom up approach (i.e., focusing on specific characteristics of the design that is to be refactored).

6. Discussion

This paper makes a distinct contribution to the emerging literature on Technical Debt in software systems, and in particular to the notion that significant debts may be related to system architecture. In particular, we show that components with higher levels of coupling are associated with higher costs of maintenance in two systems that possess very different structures; one being a Hierarchical design, and the other being a Core-Periphery design. These differences are highlighted through three separate analyses. First, a descriptive analysis, second a predictive analysis, and third, a financial analysis.

Our results have important implications for managers. They highlight the importance of design decisions made early in the life of a software system. Decisions about levels of component coupling are typically based upon challenges faced in the *current* version of a design. Yet our results reveal the long-lasting nature of these choices. Tightly coupled components cost significantly more to maintain many years after a system is introduced. The challenge for a developer is that these long-term costs are neither easy to calculate nor as salient as the near-term benefits that may stem from an approach to design that is more expedient in the short term, and which may involve greater levels of coupling.

Importantly, prior work has demonstrated that rather than being an explicit managerial choice in response to system requirements, the level of modularity in a system can be affected by factors outside the influence of individual managers. For example, MacCormack et al. (2012) compared software systems with the same size and function, but which had been developed through different organizational forms (i.e., open source communities versus commercial firms). They found the architecture of each system "mirrored" the design of the organization from which it came – loosely-coupled open source communities developed software with greater levels of modularity than collocated teams inside commercial firms. The implication is that even with the best of intentions, developers left to their own devices often create tightly-coupled, hard to maintain designs, unless organizational constraints dictate otherwise. However, the good news is that there ought to be ample opportunity to improve these designs, without affecting functionality. The question is whether it is cost effective to invest in such undertakings?

On that front, the results of our study suggest, "it depends." In projecting the financial benefits from refactoring (based upon outcomes of past refactoring efforts) we note substantial differences in value creation potential that appear to be related to the two distinctly different types of system we analyze. The value is much larger for the Core-Periphery system we study, given the Core of this system comprises almost 5000 mutually interdependent components (presenting a huge challenge to any developer trying to make a change to these components). But even with this large potential for value creation, the decision about refactoring ultimately comes down to the costs of such an effort, and the manner in which it could be executed. We provide qualitative evidence that in this scenario, refactoring might be achievable only via frequent, small, incremental componentization efforts, which aim to continuously improve system structure.

Several limitations of our study must be considered in assessing the generalizability of our results. We examine only two systems, albeit with very different designs and from two different firms. We cannot be sure that the findings would apply to other firms or systems. While we examine the costs of system design across 40,000 components, this sample and the results it generates may reflect idiosyncratic practices and design choices of these firms. Second, we have measured maintenance efforts by capturing the level of defect-related activity – in essence, the number of patches made to files to fix defects. However, the work involved to complete each patch is likely to vary according to the level of coupling for files. In this respect, we may be underestimating the true costs of high coupling (i.e., if patches to Core files consume more effort than patches to Peripheral files). Finally, while we speculate about the value that could be released via refactoring efforts, such actions have significant costs and other (often unintended) consequences on system performance. Decisions about whether to refactor a system require not only the detailed empirical data that we provide, but also a robust assessment of refactoring costs and risks before it could be known if such actions would be optimal.

Our study generates a number of avenues for future work. First, it provides a benchmark for future studies that seek to examine the relationship between measures of architecture and the costs associated with maintenance and adaptation. Second, it provides methods for evaluating the technical debt associated with software architecture, which could be verified via future empirical studies across a larger number of contexts and systems. Finally, while we focus only on the costs associated with corrective maintenance, a significant amount of the value from refactoring is likely to come from an increase in developer productivity when responding to *new* requirements. It is our hope that the methods we describe can provide a springboard for undertaking such enquiries.

7. Conclusion

Our work contributes to the emerging literature on technical debt, and in particular to studies which focus on those debts associated with system architecture. In particular, we show that measures of coupling, which capture a file's position within the network of system dependencies, are a strong predictor of subsequent file maintenance costs. We show this relationship is consistent across two software systems with very different designs; one has a "Hierarchical design", the other has a "Core-periphery" design. Our work is distinctive and departs from prior work in that we use a measure of coupling that captures the direct and *indirect* dependencies each component has in a system, as well as the *direction* of these dependencies. These data can be used to classify the role of each file in the system, and hence to identify how changes may propagate through the system.

The paper contributes to work that seeks to understand the potential value that can be released via architectural refactoring efforts that "pay down" architectural debt. Specifically, we combine financial data on the costs of maintaining components in different architectural categories, with empirical data on the outcomes of prior refactoring efforts, to project the value that could be released in each system. The results suggest greater value would come from refactoring the core-periphery system, which possesses a large core of almost 5000 mutually interdependent files. However, given the size and complexity of this system, the *realization* of this value would likely require a long-term commitment to modularize the system via frequent, small, incremental componentization efforts. Our methods provide guidance for which files one might target in such a process (i.e., Core files). And we provide anecdotal evidence from a senior developer in this system that this approach is proving useful in paying down architectural debts.

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

Appendix A. Further details of the data capture methods for the two systems

For system H, we wrote scripts to extract information from the version control system, relating to patches applied to fix defects (new feature development occurred in a separate branch of the code). Fig. 5 shows an example of the type of data available. Each patch to a file was associated with a unique bug ID and a user. We captured three years of data after a major release. A total of 518 bug IDs were identified, resulting in 12,040 pieces of defect related activity. Of these, 7124 (60%) could be traced to source files present in the major release. (The balance was associated with files where names had changed, which we could not track given the firm ran scripts remotely to maintain confidentiality; and new files added as a result of maintenance efforts post release). We removed activity related to "submit" and "validate" tasks, as these were administrative tasks associated with opening and closing bugs in the system (and we did not have comparable data for system H). For system H, the firm had a dedicated maintenance team, and thus could track person-hours. The firm provided aggregate data to us on the total number of hours the maintenance team spent on defect fixing efforts. We used a standard fully loaded cost for a developer to calculate the dollar cost of this activity. We were given access to one source code snapshot to analyze the system architecture. File names and directory path were anonymized to ensure confidentiality, but in a way that maintained the consistency of these names across source code and version control systems.

In system C, we had direct access to the version control system, so had more flexibility in how we gathered data. In a similar manner to above we mined the version control system to identify file patches to fix defects associated with the same bug ID. In this system, developers could write patches to fix defects or to add new features (as well as a number of other reasons). We captured data only on patches identified as defect fixes. And we limited this data only to defects that were present in the code that existed as of the start of the three-year period in 2007. (In system C, there was a data field that asked developers to identify the specific version of the code to which a defect related). This was the code snapshot that we analyzed to capture measures of the product architecture for prediction.

For system C, there was no distinction between developers who worked on new features versus those that maintained code. However, we had access to all commits made by all developers over the period, as well as the number of lines of source code committed in each patch. Hence we calculated the aggregate productivity (i.e., SLOC/month) for developers over the period, split by category (Core, Periphery etc.) and activity (i.e., defect fix versus new feature). We then *estimated* the total number of hours required for all commits related to fixing defects, using the total number of source lines of code added/deleted, combined with data on developer productivity when fixing defects.

References

Baldwin, CarlissY., Clark, KimB., 2000. Design Rules, Volume 1, The Power of Modularity. MIT Press, Cambridge MA.
Banker, R.D., Datar, S.M., Kemerer, C.F., Zweig, D., 1993. Software complexity and maintenance costs. Commun. ACM 36, 81–94.
Banker, RajivD., Slaughter, SandraA., 2000. The moderating effect of structure on volatility and complexity in software enhancement. Inform. Syst. Res. 11 (3), 219–240.
Barabasi, A., 2009. Scale-free networks: a decade and beyond. Science 325, 412–413.
Barry, EvelynJ., Kemerer, ChrisF., Slaughter, SandraA., 2006. Environmental volatility, development decisions, and software volatility: a longitudinal analysis. Manage. Sci. 52 (3), 448–464.
Braha, Dan, Minai, A.A., Bar-Yam, Y., 2006. Complex Engineered Systems: Science Meets Technology. Springer: New England Complex Systems Institute, Cambridge, MA.
Brooks, FrederickP., 1975. The Mythical Man-Month.
Brown, N.Y.Cai, Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., Nov 2010. Managing technical debt in software-reliant systems. In: FoSeR '10: Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research, 47-52.
Baldwin, C., MacCormack, A., Rusnak, J., 2014. Hidden structure: using network methods to map system architecture. Res. Policy 43 (8), 1381–1397.
Chidambur, S., Kemerer, C., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (9).
Cunningham, W., 1992. The wycash portfolio management system. Exp. Report, OOPSLA.
Dellarocas, C.D., 1996. A Coordination Perspective on Software Architecture: Towards a design Handbook for Integrating Software Components *Unpublished Doctoral Dissertation*. M.I.T.
Dhama, H., 1995. Quantitative models of cohesion and coupling in software. J. Systems Software 29, 65–74.
Dreyfus, D., 2009. Digital Cement: Information Systems Architecture, Complexity, and Flexibility Unpublished Doctoral Dissertation. Boston University.
Eick, StephenG., Graves, ToddL., Karr, AlanF., Marron, J.S., Audric Mockus, 1999. Does code decay? Assessing the evidence from change management data. IEEE Trans. Softw. Eng. 27 (1), 1–12.
Eppinger, S.D., Whitney, D.E., Smith, R.P., Gebala, D.A., 1994. A model-based method for organizing tasks in product development. Res. Eng. Des. 6 (1), 1–13.
Guo, Y., Seaman, C.A, 2011. Portfolio approach to technical debt management. 2nd Workshop on Managing Technical Debt. ACM.
Halstead, M.H., 1977. Elements of Software Science. Elsevier, New York.
Holland, JohnH., 1992. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence, 2nd Ed. MIT Press, Cambridge, MA.
Kauffman, StuartA., 1993. The Origins of Order. Oxford University Press, New York.
Kazman, Rick, Cai, Yuanfang, Mo, Ran, Feng, Qiong, Xiao, Lu, Haziyev, Serge, Fedak, Volodymer, Shapochka, Andriy, 2015. A case study in locating the architectural roots of technical debt. In: Proceedings of the 37th International Conference on Software Engineering. ACM.
Kemerer, C.F., Slaughter, S.A., 1997. Determinants of software maintenance profiles: an empirical investigation. J. Softw. Maintenance 9, 235–252.
Kruchten, P, Nord, R.L., Ozkaya, M., 2012a. Technical Debt: from metaphor to theory to practice. IEEE Softw 29 (6), 18–21.
Kruchten, P., Gonzalez, M., Ozkaya, I., Nord, R.L., Kruchten, N., 2012b. Change propagation as a measure of structural technical debt. WICSA/ECSA.
Li, Z., Avgeriou, P., Liang, P., 2015. A systematic mapping on technical debt and its management. J. Syst. Softw 101, 193–220.
MacCormack, A., Verganti, R., Iansiti, M., 2001. Developing products on 'internet time': The anatomy of a flexible development process. Manage Sci 47 (1), 133–150.
MacCormack, A., Baldwin, C., Rusnak, J., 2012. Exploring the duality between product and organizational architectures: a test of the 'mirroring' hypothesis. Res. Policy 41 (8), 1309–1324.
MacCormack, A., Rusnak, J., Baldwin, C., 2006. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. Manage. Sci. 52 (7), 1015–1030.
MacCormack, A., Rusnak, J., Baldwin, C., 2007. The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. Harvard Business School Working Paper, pp. 08–038.
Martin, R., 2002. Agile Software Development, Principles, Patterns, and Practices. Pearson Publishing.
McCabe, T.J., 1976. A complexity measure. Software Engineering, IEEE Transactions on 308–320.
McConnell, S. Technical Debt. 10x software development accessed 2016-05-27.
Murphy, G.C., Notkin, D., Griswold, W.G., Lan, E.S., 1998. An empirical study of static call graph extractors. ACM Trans. Softw. Eng. Method. 7 (2), 158–191.
Nord, R.L., Ozkaya, I, Kruchten, P, Gonzalez, M, 2012. In search of a metric for managing architectural technical debt. WICSA/ECSA 2012.
Nughoro, A., Visser, J., Kuipers, T., 2011. An Empirical model of technical debt and interest. In: Workshop on Managing Technical Debt. ACM, p. 2011.
Parnas, D.L., 1972. On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 1053–1058.
Rivkin, JanW., 2000. Imitation of complex strategies. Manage. Sci. 46, 824–844.
Rivkin, JanW., Siggelkow, Nicolaj, 2007. Patterned interactions in complex systems: implications for exploration. Manage. Sci. 53 (7), 1068–1085.
Sanderson, S., Uzumeri, M., 1995. Managing product families: the case of the sony walkman. Research Policy 24 (5), 761–782.
Sarker, S., Ramachandran, S., Sathish Kumar, G., Iyengar, M.K., Rangarajan, K., Sivagnanam, S., 2009. Modularization of a large-scale business application: a case study. IEEE Softw 26 (02), 28–35.
Schmid, K., 2013. On the limits of the technical debt metaphor: some guidance on going beyond. In: Proceedings of the 4th International Workshop on Managing Technical Debt, pp. 63–66.
Seaman, C, Guo, Y., 2011. Measuring and monitoring technical debt. Adv. Comput..
Selby, R.W., Basili, V.R., 1988. Error localization during software maintenance: generating hierarchical system descriptions from the source code alone. In: Software Maintenance, 1988., Proceedings of the Conference on, pp. 192–197.
Sharman, D., Yassine, A., 2004. Characterizing complex product architectures. Systems Engineering Journal 7 (1).
Shaw, Mary, Garlan, David, 1996. Software Architecture: An Emerging Discipline. Prentice-Hall, Upper Saddle River, NJ.

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007

Simon, H.A., 1962. The architecture of complexity. Proc. Am. Philosoph. Soc. 106, 467–482.
Sosa, M., Mihm, J., Browning, T., 2013. Linking Cyclicality and Product Quality. Manuf. Service Operations Manag. 15 (3), 473–491.
Sosa, ManuelE., Eppinger, StevenD., Rowles, CraigM., 2004. The misalignment of product architecture and organizational structure in complex product development. Manage. Sci. 50 (12), 1674–1689.
Sosa, Manuel, Eppinger, Steven, Rowles, Craig, 2007. A network approach to define modularity of components in complex products. Trans.ASME 129, 1118–1129.
Spear, S., Bowen, K.H., 1999. Decoding the DNA of the toyota production system. Harvard Bus. Rev.
Steward, DonaldV., 1981. The design structure system: a method for managing the design of complex systems. IEEE Trans. Eng. Manage. EM-28 (3), 71–74.
Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw 86 (6), 1498–1516.
Ulrich, Karl, 1995. The role of product architecture in the manufacturing firm. Res. Policy 24, 419–440.
Warfield, J.N., 1973. Binary matricies in system modeling. IEEE Trans. Syst. Manage. Cybernetics 3.
Whitney, D.E. (Chair) and the ESD Architecture Committee, 2004. The influence of architecture in engineering systems. Eng. Syst. Monograph.
Xiao, Lu, Cai, Yuanfang, Kazman, Rick, 2014. Design rule spaces: a new form of architecture insight. In: Proceedings of the 36th International Conference on Software Engineering. ACM.
Xiao, Lu, Cai, Yuanfang, Kazman, Rick, Mo, Ran, Feng, Qiong, 2016. Identifying and quantifying architectural debt. In: Proceedings of the 38th International Conference on Software Engineering. ACM.

Please cite this article as: A. MacCormack, D.J. Sturtevant, Technical debt and system architecture: The impact of coupling on defect-related activity, The Journal of Systems and Software (2016), http://dx.doi.org/10.1016/j.jss.2016.06.007