Empirical Analysis of Software Refactoring Motivation and Effects

By

Sean M. Gilliland

B.S. Computer Science, University of Nevada, Reno, 2004

Submitted to the System Design and Management Program
In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the
Massachusetts Institute of Technology
February 2015

© 2015 Sean M. Gilliland. All Rights Reserved
The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature of Author_____
System Design and Management Program
January 30, 2015

Certified by_____
Dr. Michael A. M. Davies
Thesis Supervisor
Senior Lecturer, MIT Engineering Systems Division

Certified by_____
Dr. Carliss Y. Baldwin
Thesis Supervisor
William L. White Professor of Business Administration, Harvard Business School

Certified by_____
Dr. Alan D. MacCormack
Thesis Supervisor
MBA Class of 1949 Adjunct Professor of Business Administration, Harvard Business School

Certified by_____
Dr. Daniel Sturtevant
Thesis Supervisor
Research Associate, Harvard Business School

Accepted by_____
Patrick Hale
Director
System Design and Management Program

(Intentionally Left Blank)

Empirical Analysis of Software Refactoring Motivation and Effects

By
Sean M. Gilliland

Submitted to the System Design and Management Program on January 30, 2015
in Partial Fulfillment of the Requirements for the Degree of Master of
Science in Engineering and Management

ABSTRACT

As complexity and levels of technical debt within software systems increase over time the incentive of an organization to refactor legacy software likewise increases. However, the opportunity cost of such refactoring in terms of engineering time and monetary investment have proven difficult to effectively trade against the long term benefits of such refactoring. The research investigates the empirical effects of a multi-year refactoring effort performed at a world-leading software development organization. DSM architectural representations of software pre- and post-refactoring were compared using core-periphery analysis, and various quantitative metrics were identified and compared to identify leading indicators of refactoring. The research finds several uniquely identifying properties of the area of the software system identified for refactor, and performs a comparison of these properties against the architectural complexity of those modules. The paper concludes with suggestions for additional areas of research.

Thesis Supervisor: Dr. Michael A. M. Davies
Title: Senior Lecturer, MIT Engineering Systems Division

Thesis Supervisor: Dr. Carliss Y. Baldwin
Title: William L. White Professor of Business Administration, Harvard Business School

Thesis Supervisor: Dr. Alan D. MacCormack
Title: MBA Class of 1949 Adjunct Professor of Business Administration, Harvard Business School

Thesis Supervisor: Dr. Daniel Sturtevant
Title: Research Associate, Harvard Business School

ACKNOWLEDGEMENTS

This thesis, and indeed the entirety of my time at MIT, could not have been possible without the support of many individuals. First, thank you to my wife Angel, whose unwavering support and love kept me focused and gave me the courage to achieve one of my lifelong dreams. Without you I would not be where and who I am today.

Thank you to my family, friends, and co-workers, both at home and in Boston, whose consistent support kept me focused and motivated to overcome any and all obstacles.

Thank you to my thesis advisor, Dr. Michael Davies and Dr. Daniel Sturtevant for giving me the opportunity to perform meaningful research in an area I find personally and professionally compelling.

Thank you to Drs. Allan MacCormack and Carliss Baldwin for their guidance and insight while I worked through this research.

And finally, thank you to the SDM program and MIT staff and professors for the opportunity to study with and learn from some of the most talented and engaging people in the world.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 Background and Motivation

As software ages, is maintained, corrected, and expanded upon by developers both experienced and new to its intent, it is natural for the relationships between the various software modules to increase in complexity. As function is added, new form is likewise added, and relationships between previously separate modules are established, adding to the overall tangle of the system, unless strictly managed. This strict management of architecture is difficult to value – after all the architecture of a software system is not what is experienced by the user, only the function that the architecture facilitates – so over time emphasis is naturally placed on creating new function at the cost of the long term health of the system's architecture. This trade is referred to in software development industries as "technical debt" [1], and functions as any debt does. New function implemented today comes at a cost in terms of maintenance effort, engineering time, flexibility, and an increase in the complexity of the software system in the future. Unless this debt is continually serviced the software system eventually becomes rigid, difficult to extend and maintain, with modifications in one area of the system inexplicably affecting other, unrelated areas. Other less tangible costs include decreases in engineer morale, and increases in stress and turnover [2] making complex software systems both physically and mentally taxing.

The activity of refactoring software is one way to pay down technical debt and to control complexity. By investing time and engineering resources in the activity of software maintenance flexibility may be maintained, tightly-coupled components serving independent needs may be safely divided, and new features and defects are effectively isolated from disrupting other areas of the system. All of these are desirable attributes however their return on investment is elusive to gauge. Traditional methods of measurement – cost and time savings – tend to fail as a means

of measuring gains in software quality due to refactoring. Unlike physical products which may be repeatedly replicated to prove benefits of improvement all software projects are unique, making metrics that apply to one system foreign to another. This leads to a challenge and the crux of this research – given a multi-year software refactoring effort at a world-leading software development organization this research explores the application of Core-Periphery analysis to isolate areas of refactoring, and explore benefits along the edges of functional division.

1.2 Research Scope

The primary objective of this research is to apply and evaluate Core-Periphery analysis of a mature software system as an effective means to quantify the value of increased modularity through refactoring. The software under analysis is a primarily C++-based system intended to function in a highly regulated industry, and is comprised of various modules with varying degrees of coupling. For the purposes of this study the software system can be understood as being comprised of a highly structured module intended primarily to meet strict operating regulations – Module A – and a highly flexible module intended primarily to provide a variety of front-end experiences to the user – Module B. Figure 1 illustrates this relationship.

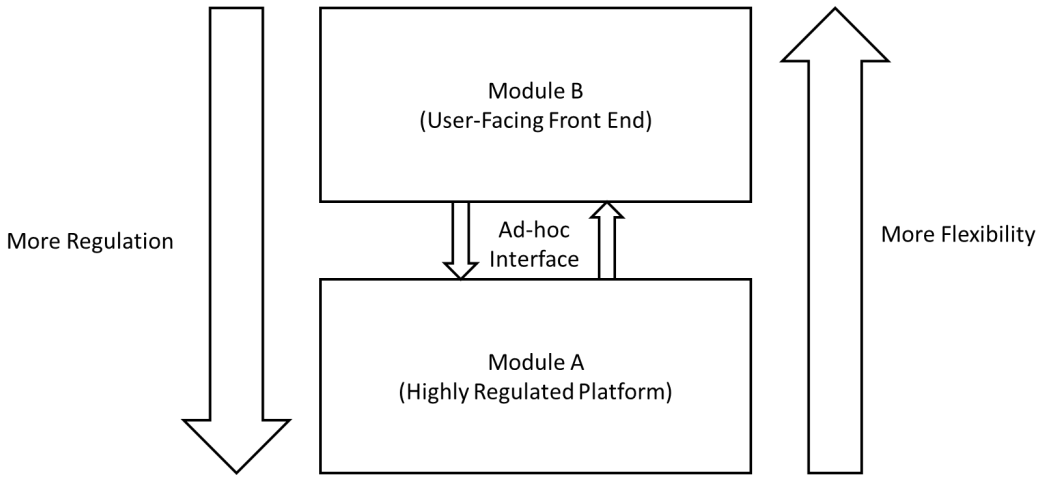


Figure 1 High-level Module Relationship of System under Analysis

Over the years from 2011-2014 this system has been split into two product branches, the first representing a high degree of direct coupling between Module A and Module B, and the second representing a decoupling of the primary modules with a well-defined, controlled interface. These branches have evolved independently of one another with function implemented in both using separate methods. This symmetry provides a unique opportunity to analyze the benefits and drawbacks of the additional modularity following refactoring. Metrics for each file in each system will be analyzed, including a Core-Periphery dependency analysis as well as a comparison of critical attributes such as date and time of modification, number of files required to add new features or correct defects, and number of changes within a file necessary for the same.

The final aspect of the research is an exploration of methods to augment Core-Periphery analysis in order to capture dependencies between files via message passing. The system under analysis makes extensive use of message passing between its component processes, however classic static analysis is unlikely to capture these dependencies accurately. Instead, such analysis is likely to indicate significant dependence on files that facilitate this communication rather than the communicating files themselves. Figure 2 illustrates this behavior.

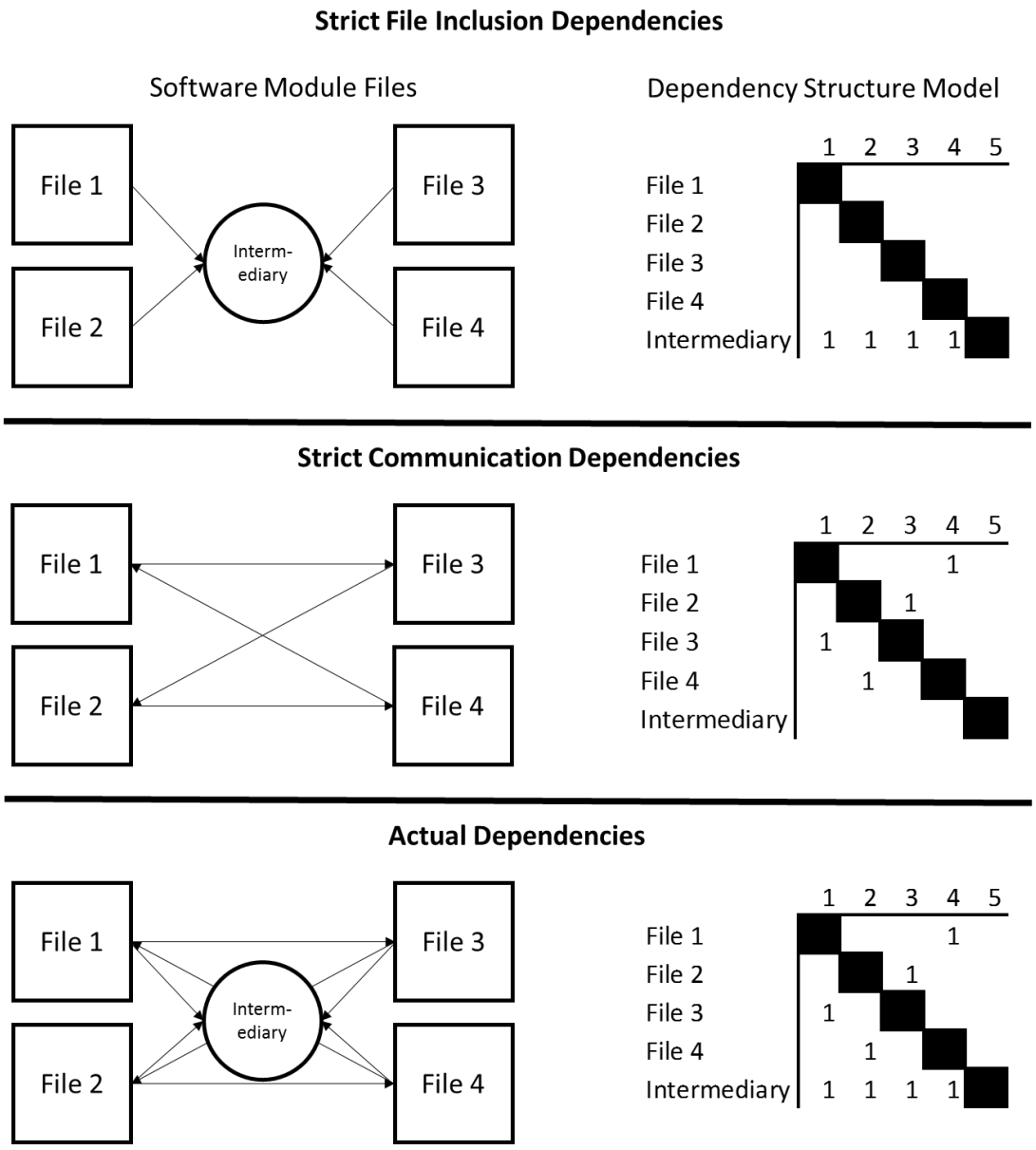


Figure 2 Software Dependency Structure using Message Passing

Finally, limitations and advantages of the research methodology are presented, including recommendations for future research.

1.3 Research Objectives

The primary objective of this research is to evaluate and quantify the benefits and drawbacks of software refactoring within a world-leading software organization. Core-Periphery analysis of

the system under study both pre- and post- refactoring will be used to isolate and analyze the primary area of refactoring – a formalized interface decoupling a highly stable and regulated software module from a highly fluid and less regulated module. Due to the multi-disciplinary nature of the software system all quantitative data is augmented using qualitative interviews with individuals experienced with pre- and post-refactoring to determine both measurable and perceived value of refactoring.

These data will be evaluated to confirm or refute the following research hypotheses. The first four hypotheses test metrics intended to identify motivating factors in the pre-refactoring software system. The second four hypotheses test these same metrics to measure the impact of refactoring in the post-refactoring software system.

- H1 – We investigate the relationship between the number of components that directly or indirectly depend on a module (Visibility Fan-In, VFI) [2], and the number of components that are directly or indirectly depended on by a module (Visibility Fan-Out, VFO) [2] within the system as it correlates to that module being targeted for refactoring; we expect the VFI characteristic of refactored modules to be elevated in the area of refactor.
- H2 – We investigate the relationship between the defect density within the system as it correlates to refactoring; we expect modules with increased defect density to be a factor for targeted refactoring within the system.
- H3 – We investigate the relationship between McCabe's Cyclomatic Complexity [22] within the system as it correlates to refactoring; we expect modules with increased cyclomatic complexity to be a statistically significant factor for targeted refactoring within the system.

- H4 – We investigate the relationship between new feature development within the system as it correlates to refactoring; we expect modules with lower rate of new feature development to be a factor for targeted refactoring within the system.
- H5a – We investigate the impact of refactoring on VFI and VFO within the system; we expect refactoring to reduce these values, moving refactored modules toward the Periphery from the Core.
- H6 – We investigate the impact of refactoring on defect density values within refactored modules as compared to the remainder of the system; we expect refactoring to decrease this metric in the area of refactor.
- H7 – We investigate the impact of refactoring on McCabe's cyclomatic complexity values within refactored modules as compared to the remainder of the system; we expect refactoring to decrease this metric in the area of refactor.
- H8 – We investigate the impact of refactoring on new feature development within refactored modules as compared to the remainder of the system; we expect refactoring to increase this metric in the area of refactor.

1.4 Organization of Thesis

Chapter 2 of this thesis reviews the existing literature of modularity and its effects on software systems, software refactoring costs and benefits, design structure matrices as tools to represent and analyze software structure, and Core-Periphery analysis of software. This review is followed by a discussion of the DSM framework as it has been previously applied to Core-Periphery analysis at Ironbridge Technology. Focus is placed on the application of the Core-Periphery method to analyze the tangible and intangible impact of complex software architecture, as well as its potential to isolate and compare specific software modules between systems.

Chapter 3 discusses the software development organization and software system under analysis, providing a general overview of the software and organizational architecture. This architecture is then broken down along lines of increasing regulatory requirements, and overlaid with the software organization's structure to identify ideal locations of increased modularity across both product and organizational boundaries. This analysis is then performed using the organization's refactored software architecture and re-compared.

Chapter 4 introduces Core-Periphery analysis of the software system both pre- and post-refactoring, expressing the software's internal dependencies in DSM form. Due to its extensive application, the software concept of message passing and its drawbacks to generalized static dependence analysis is then discussed. A solution to this drawback is proposed, implemented, and then utilized to augment the Core-Periphery analysis with additional software dependencies that may have previously been overlooked. This augmented view of both systems serves as the basis for further analysis, with the area identified in Chapter 3 as the primary focus.

Chapter 5 provides an overview and analysis of gathered meta-data for each file in both views of the system. This data is gathered directly from the software organization's version control systems, and provide views of various qualities of different areas of the software systems. These metrics include code rate of change, activity of developers at all experience levels, and number of file modifications necessary to implement new features and correct defects. This information is overlaid with Core-Periphery analysis from Chapter 4 to visualize these data across different module categories.

Chapter 6 provides a final discussion of research findings, beyond answering the primary research questions. Advantages and disadvantages of the framework are discussed, and areas of future work are identified that could strengthen the analysis.

1.5 Data of Thesis

Data for the background literature research is obtained from publically available published works by various experts in their respective fields. Data used to analyze the software system pre- and post-refactoring has been directly queried from the software organization's version control systems, defect tracking systems, and project tracking systems; these data reflect the entire recorded history of each file in the system. Exact measurements are not shared within this research, however conclusions and derived data visualizations, tables, and charts are included. High-level software and organizational architecture are derived from software design documentation and organizational charts shared by the software organization, and have been anonymized in accordance with an agreement with the organization. Detailed software architecture of both system views is derived using Core-Periphery tools developed by Dr. Dan Sturtevant and Scientific Tool's *Understand* software package.

Chapter 2 Literature Review

2.1 What is Modularity?

Before a discussion of the impact of modularity on complex systems, it is important to establish what constitutes modularity, and how it can be applied to the software under analysis. It is well established that modularity is a desirable attribute, a useful tool for managing complexity, and is actively sought by organizations and engineers both as good design. Clark and Baldwin [3] describe a system as "modular" when it exhibits the attributes of interdependence within and

independence across structurally independent units. These independent units, and the relationships between them together form the larger system. Sanchez and Mahoney [4] describe modularity as a special form of design creating high degrees of independence between modules through the use of standardized component interface specifications, with these interfaces specifying the inputs and outputs linking the components of the design. Gauthier and Ponto [5] similarly describe a module as a separate and distinct system component whose inputs and outputs are well-defined, and whose independence may be used to isolate errors and deficiencies from the remainder of the system. For the purposes of this study a system is considered to be modular if it exhibits the qualities of high internal cohesion and low external coupling, with well-defined interfaces that establish expected inputs and outputs with other modules.

2.2 Modularity in Software Systems and Organizations

Structure and its impact within both software systems and organizations has been the subject of discussion for many years and by many authors. Organizations and software systems are particularly well suited to modular structure due to their relatively similar logical compositions – organizational structure facilitates the completion of tasks by "quasi-independent divisions" just as software systems produce value through the coordination of modules of routines to create customized applications [4]. Developers of software systems benefit from and actively seek increased modularity in order to facilitate flexibility and adaptability, and achieve this through minimization of coupling between modules, effectively encapsulating information within each module; this encapsulation not only allows individual modules to be constructed without specific knowledge of the internal workings of other modules, but allows modules to be replaced, reinserted, or further decomposed without interrupting the function of the overall system [4].

By isolating and maintaining well-structured interfaces between modules the system is capable of transformation in several dimensions, each applicable to a different system need. Baldwin and Clark [3] identified six distinct operations which leverage the well-established interfaces and independent nature of modules –augmentation, exclusion, substitution, splitting, inversion, and porting. Augmentation and exclusion add and remove modules respectively, substitution replaces one module with another, splitting divides and replaces a single module with multiple modules, inversion re-encapsulates a portion of a module and moves it higher in the design hierarchy, and porting describes the act of modifying a module's interface to facilitate compatibility in more than one system. A modular system is thus capable of adaptation and reconfiguration as business and design needs change over time. From Design Rules: The Power of Modularity [3] p. 143, Figure 3 illustrates the application of these modular operators.

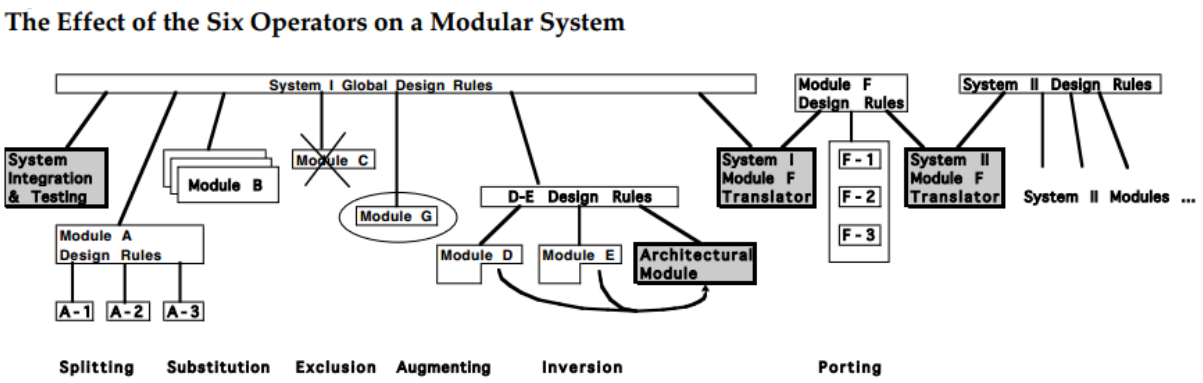


Figure 3 Application of the Modular Operators on a System

In an analysis of open source software, MacCormack et al. [6] establish a method of analysis utilizing design structure matrices to measure the relative degrees of modularity within a system by measuring the cost of dependence between elements. They establish two primary costs – propagation cost, which assumes the cost of dependence between elements is constant regardless of logical distance between modules, and clustered cost, which assumes the cost of dependence

differs depending on whether elements are contained within the same or different clusters. In this way it is possible to establish a level of relative modularity of a system, and empirically calculate where modules (intended and otherwise) exist within a system's structure.

2.3 Software Refactoring Costs and Benefits

Mens and Tourwe [7], describe the need for evolution and adaptation as an "intrinsic property of software." It is an activity that controls complexity of software as it ages, is modified, extended, and maintained by the myriad developers involved in even the simplest of software development. Because it is an accepted, natural activity, the primary question when facing refactoring is not **if** it is required, but **how** and **where** to refactor the software for greatest effect. While the activity of refactoring can be broken down into distinct activities, the primary goal of software refactoring remains as improving the internal structure of the code without altering its external behavior [8].

The goals of software refactoring immediately lend themselves well to the concept of modularity. Through encapsulation and the establishment of well-defined interfaces the external behavior of a software module can be verified as constant, while the internal components of the module may be effectively modified without knowledge of or impact to the internal workings of other modules. Because of this relationship, highly modular systems are not only adaptable as discussed in Section 2.2, but lend themselves well to internal evolution and iteration, thus effectively facilitating the natural software development activity of refactoring. Baldwin and Clark [3] describe system evolution as a value-seeking process in which architecture is modified in order to best fit system needs as they change.

Many tools have been applied in the past to software refactoring in an effort to best identify points of improvement, support buy-versus-build decisions, determine sub-component quality analysis, test resource allocation, and identification of error-prone components. Coleman, et al. in a 1994 IEEE article [9] identified and evaluated five of these tools – all of which quantitatively measured software maintainability in relation to primary software metrics, and were applied by major software organizations as simple measures of maintainability consumed by "line" engineers to effect improvement. These tools effectively identified problematic areas of the code base that aligned with the expert's intuition of where difficulties could be located, lending credence to their opinions. Importantly, these tools could be applied at differing levels of system decomposition – component, sub-system, and whole-system – in order to provide system-level insight.

However, in spite of the tools and general support for the benefits of software refactoring relatively few empirical studies have been performed to evaluate the value and cost of software refactoring. The outcome of those studies that have been performed can at times be contradictory – for instance, Kim et al. [10] note that while Weissberger and Diehl [11] find that high levels of refactoring are often followed by high levels of introduced defects, Ratzinger et al [12] find that increased levels of refactoring actually have the opposite effect. In their analysis of the impact of refactoring at Microsoft, Kim et al. [10] identify several roadblocks that contribute to the cost of refactoring, including regression defects, code churn, merge conflicts, resource conflicts with other projects, additional overhead for code reviews and reviewers, and the risk of committing too much engineering time or over engineering the system. While engineers intuitively understand the benefits of refactoring, these contradictory findings serve to

complicate the already difficult task of identifying where and when to refactor for the most benefit.

2.4 DSM as a Tool for Software Analysis

The Design Structure Matrix (DSM) has proven a useful analysis tool for a variety of applications in both organizational and technical systems. DSMs represent modules and their relational data in a two-dimensional grid format, providing an effective method of categorizing the relationships between individual components as well as component groups. Components with high levels of coupling can be effectively clustered, and components on which a high number of other components are dependent (or vice versa) stand out clearly using this kind of analysis. Originally introduced by Steward in 1981 [13], it has been applied extensively in the intervening years by a variety of managerial and technical experts [14, 15, 16]. MacCormack et al [6] note that one of the primary benefits of DSM analysis is its ability to highlight system modularity based not only on the number of dependencies between components, but also based on the pattern of those dependencies – this provides insight into the actual architecture of the analyzed system.

2.5 Core-Periphery Analysis of Software

Dr. Dan Sturtevant's 2013 doctoral thesis, *System Design and the Cost of Architectural Complexity* [2], analyzes the physical and psychological cost incurred by architectural complexity finding that measuring architectural complexity is equivalent (and in some cases superior to) traditional methods of measuring software complexity. One of the primary analysis methods applied in this research is MacCormack and Baldwin's Core-Periphery analysis of architectural DSM diagrams [2, 17], which captures the level of coupling between any given file and the rest of the system. Sturtevant describes this method in the following five steps:

- Capture a network representation of a software product's source-code using dependency extraction tools
- Find all indirect paths in the network by computing the graph's transitive closure
- Assign two visibility scores to each file that represent its reachability from other files or ability to reach other files in the network
- Use these visibility scores to classify each file as one of four types: peripheral, utility, control, or core

This method derives the actual architecture of a software system by extracting and analyzing the following software dependencies:

- *From* the site of a function call *to* the site of the function's definition
- *From* the site of a class method invocation *to* the site of the class method's definition
- *From* the site of a class method invocation *to* the site of the class definition
- *From* the site of a subclass definition *to* the site of the parent class definition
- *From* the site of a non-trivial user-defined type instantiation *to* the site where that type is defined

These values are then used to categorize files into one of four types:

- *Peripheral* files share the characteristic of low incoming and outgoing dependencies. These are the most isolated files within the system.
- *Shared* files do not depend on many other files, but are depended on by a high number of files. These files provide functionality used throughout the system.
- *Control* files depend on many other files, but are not highly depended on. These files coordinate behavior provided elsewhere in the system.

- *Core* files share the characteristic of high incoming and outgoing dependencies. These files represent areas of high coupling within the system, and are thus the most difficult to develop and maintain.

Sturtevant's analysis found mathematically significant correlation between the classification of a file / module, and the productivity and general mental health of those engineers contributing to that portion of the software. Increased architectural complexity resulted in more modifications necessary to effect change, both in correcting defects and adding new features, as well as an increase in defect density, a decrease in average length of tenure, and a general decrease in engineer productivity. Each of these metrics directly impacts both quality and quantity of software produced, leading to significant motive for engineering organizations to control the levels of architectural complexity in their products, in addition to more traditional complexity metrics such as McCabe cyclomatic complexity [2, 22].

The ability to not only effectively facilitate refactoring, but to also effectively identify areas of the code base expected to benefit the most from refactoring could prove to be valuable information for organizations pursuing refactoring. Kim et al. [10] performed interviews with more than 1000 engineers at Microsoft who participated in multiple refactoring efforts in the Windows Vista operating system, and while engineers were able to eloquently identify the benefits and risks of refactoring, the act of identifying where to perform refactoring was largely left to intuition and first-hand experience with highly complex areas of the system. In fact, while refactoring tools are actively included in Microsoft's Visual Studio software development environment, 71 percent of Kim's respondents indicated that while these tools were useful, they served as only part of a larger refactoring effort including a comprehensive architectural analysis.

The Core-Periphery method visualizes the VFI and VFO metrics of the software system in order to effectively analyze software architecture and identify areas in need of refactor.

Chapter 3 Software and Organization under Analysis

3.1 Introduction and History

The system under analysis constitutes a highly regulated entertainment product comprised of software, hardware, and firmware to create an end-to-end solution. While initially deployed in the 1990s and early 2000s as a single highly-coupled system the software sub-system has since been further decomposed along two primary functional requirements:

- A highly structured platform (Foundation) responsible for delivering the majority of regulatory requirements, accounting, money-handling, security, and operator configuration; and
- A flexible software component (game) responsible for delivering customized and highly variable player-facing content.

Very early releases of this system packaged the Foundation one-to-one with a game representing a unified product with both regulatory requirements and game content released together. As time progressed the naturally divergent purposes of the platform were identified and effectively separated from game development, revolutionizing the product and producing families of Foundation support upon which multiple games could be released. The software was effectively divided into two components, allowing all released games to benefit from new features and defects that were corrected in that Foundation family, and inversely ensuring backward compatibility would be maintained.

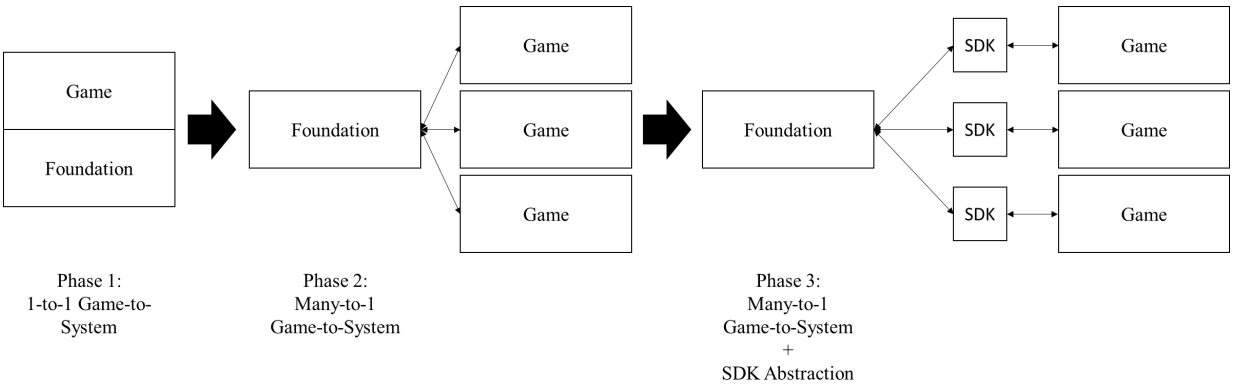


Figure 4 Evolution of the Foundation as a Platform

The internal structure of the organization itself underwent a similar transformation during this period of time, with the game development teams, once co-located with the Foundation developers, being separated into various "studios" each of which developed games targeting different regulatory jurisdictions and market segments. This separation of concerns allowed games and game studios to evolve at a rapid pace, experiment with new game formats and presentations, while the Foundation evolved at a slower, more deliberate pace dictated by its primary objective of maintaining regulatory compliance and stability.

However this separation of concerns did not come with an implementation of true modularity. The Foundation and game still bore high levels of coupling which fell into the following categories:

- Dependence via class inheritance, callbacks, and function invocation
- Dependence via message passing, and several categories of events representing changes in both the internal state of the system as well as the presentation of that internal state

This ad-hoc communication structure allowed the less-structured games to alter the internal state of the entire software system along multiple points of entry. Despite a robust body of documentation specifying expected use, this communication structure created tight coupling between these two components both serving opposing objectives, and would frequently lead to unexpected internal system states resulting in numerous defects without a clear owner.

It was this level of coupling and opposing directives of the involved software components that served as the initial motivation to refactor the system. This effort finalized the modularity begun in the years prior establishing a distinct separation between the Foundation and game components with a well-defined interface. Two channels of communication were established using web sockets based on XML messaging between the components – the first channel managed reconciliation of the system's internal state between the Foundation and the game, while the second channel managed the presentation of the system's state to the player and operator.

Following this refactoring, the Foundation has since progressed as two independent products – the first representing the highly-coupled pre-refactoring architecture, the second representing the modularized post-refactoring architecture. For the purposes of this study the most recent revisions of both products is used as an exemplar for that paradigm.

3.2 Product versus Organizational Architecture

3.2.1 **Impact of Regulation**

The effects of regulation have been made apparent throughout not only the organization under analysis, but also the software system it produces. Conway's Law [18, 19] predicts that this is to be expected – the structure of an organization reflects the product created by that organization.

As the product matured through the 1990s and early 2000s this was made most apparent in the eventual separation between the Foundation and game. The directive of the Foundation to satisfy strict regulatory requirements demanded that its development cycles eventually include longer design and testing phases to consider the swiftly increasing interactions between jurisdictional needs and new features necessary for game development.

Games, on the other hand, experienced an opposing effect – mostly liberated of regulatory needs emphasis was instead placed on high levels of iteration, with dozens of games created per year. The interfaces between the Foundation and the Game modules rapidly began to experience strain, churn, and increased levels of maintenance due to these differences in necessary development velocity. This effect of architectural shear quickly outgrew the monolithic view of the system evident in its initial releases, driving the establishment of the first system platform and the separation of both game developers into separate teams, and games themselves into separate products.

3.2.2 **Mirroring of Organizational and Product Architectures**

MacCormack et al [20] explored the application of Conway's Law as it applied to tightly- and loosely-coupled organizations, finding that loosely-coupled organizations tended to create more modular software out of necessity – modules allowed disparate teams to modify and improve the software independently. The same pattern is observed in the organizational changes occurring during this period of architectural shear. As the product experienced tension between iteration and stability, so too did the teams contributing to its development. At this time game development "studios" were created, and located independently from the teams now responsible for the newly-created Foundation.

Communication between the Foundation and game teams was ad-hoc and informal, still in the initial stages of separation. This process was eventually formalized with the creation of an intermediary team through which communication was facilitated. This organizational change was soon followed by product refactoring that formalized the module interface between the software Foundation and games. Current organizational architecture overlaid with areas of jurisdictional responsibility closely match the refactored product architecture, including the now formalized communication channels required to finalize initial modularity first motivated by opposing levels of regulatory compliance.

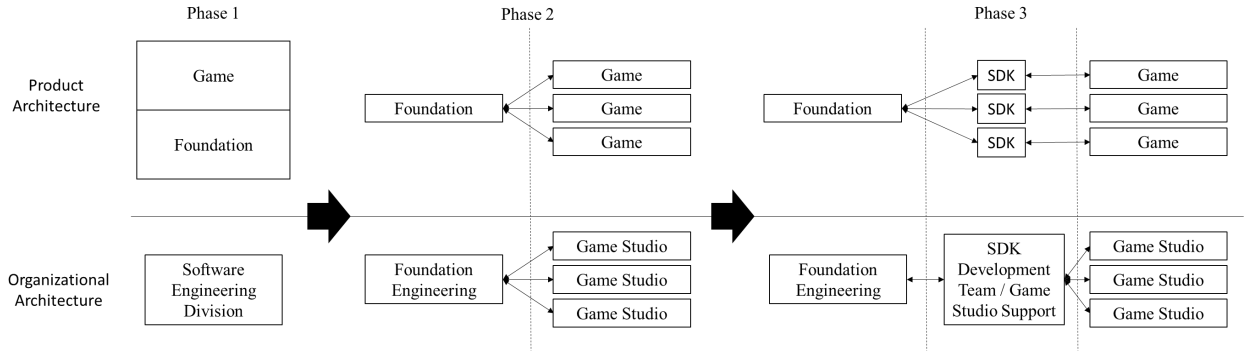


Figure 5 Product and Organizational Evolution over Time

3.2.3 **Strategic Motivation for Increased Modularization**

Over the course of the three phases of the product lifecycle described in Figure 5 the number of games capable of being deployed with any given Foundation increased rapidly. By altering the manner in which games were deployed the Phase 3 architecture experienced explosive growth in number of supported products as compared to previous iterations. In contrast to the organizational structure, the modularity in the software system that achieved this growth was not formalized – the interface between the Foundation and Game modules consisted of ad-hoc message passing, direct method invocation, and shared library dependencies resulting in lingering tight coupling at the software level.

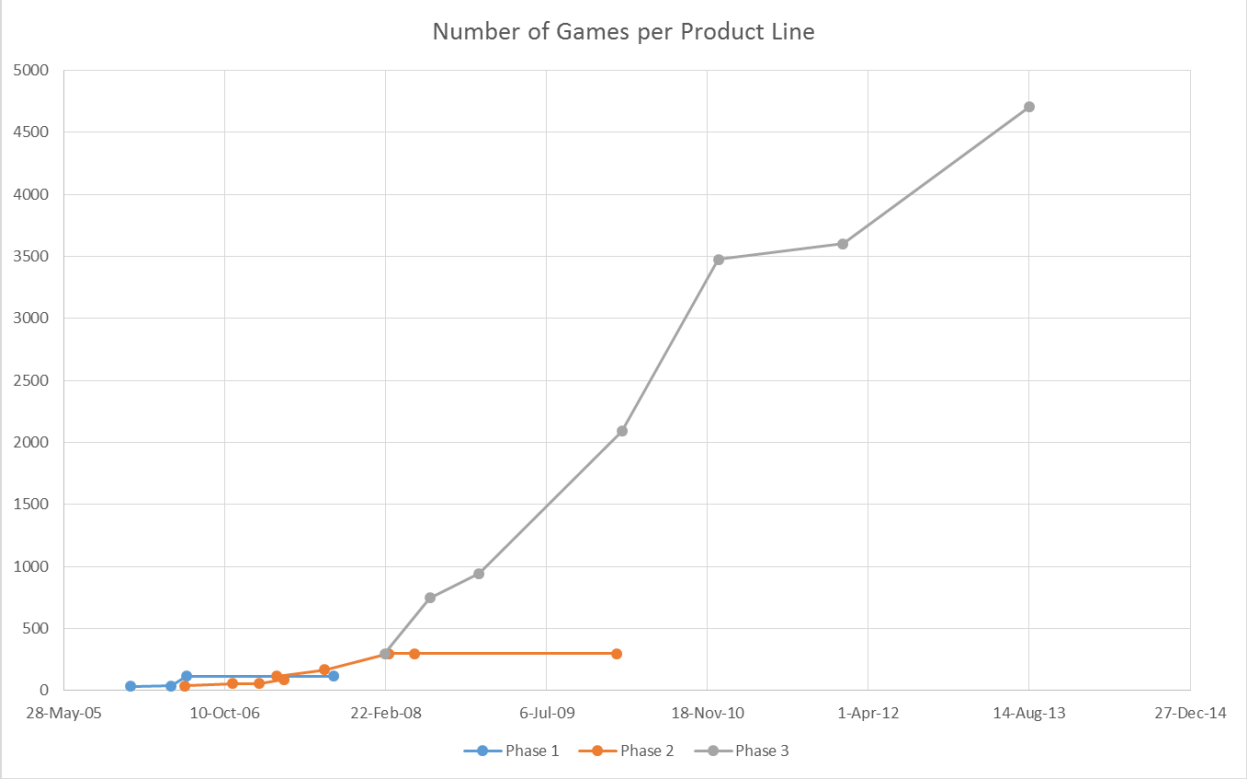


Figure 6 Rate of Game Development over Time

3.2.4 **Impacts of Increased Inter-Module Dependence**

Phase 3 Foundation development was itself divided into multiple releases (families) over time, each representing progressively increased levels of game feature support and jurisdictional compliance. Six of these releases were analyzed during this research. A set of exemplar games were selected at random per family all sharing similar criteria:

- The game *must* represent original development
- The game *must* be approved by its corresponding regulatory bodies and sold as a product
- The game *must* represent consistent expected development complexity (the set is of similar style, brand, and design)

Figure 7 Game Development Metrics over Time

The software package *Understand*, a static analysis tool developed by *Scientific Toolworks, Inc.,* was then used to generate a screening measurement of the level of coupling between these exemplar games and the Foundation for which the game was targeted. This measurement includes only direct method invocation, class inheritance, and other statically-detectable dependencies and does not include dependencies generated by message-passing or shared libraries. These results are shared in Figure 7, and show a generally increasing trend with the final set of games targeting Family 6 representing more than twice the original level of dependence.

Phase 3	# Dependencies	Mean Dev. Hours	Mean Test Hours	Mean # Defects
Family 1	11211	440	223	3
Family 2	15460	559	269	36
Family 3	17130	555	297	33
Family 4	14282	558	445	26
Family 5	17159	620	490	35
Family 6	23155	705	903	115
Correlation		0.95	0.87	0.95

Table 1 Summary of Game Development Measurements over Time

In addition to the levels of inter-module dependence, other metrics affecting game development: engineering hours, test hours, and number of found defects, were also measured for each of the exemplar games. Each also shared a generally increasing trend, and a simple correlation indicates a statistically significant ($p < 0.05$) correlation for two of the three metrics, which provided initial research direction despite not being enough to prove causation. This relationship could serve as the topic of an altogether separate course of research.

With the value of modularity in deployment and organizational structure already historically proven, the increasing overhead of game development over time generated great strategic value in both mitigating and controlling the levels of dependence between the Foundation and Game. This resulted in a concerted refactoring effort of the Foundation to support and drive this new formalized communication channel. It is this refactoring effort that is the subject of this research.

Chapter 4 Core-Periphery Software Analysis

4.1 Introduction

The primary tool applied to analyze the software system both pre- and post-refactoring is MacCormack and Baldwin's Core-Periphery analysis [2, 17]. This analysis considers each file within the software system to represent a node in a network. The software instructions contained in each file contain references to data structures and instructions contained not only within the file, but also commonly include dependencies on data and instructions defined in other files. Those familiar with programming will immediately recognize the common practice of *declaration* versus *definition* as a simple example of this type of dependence. Directionality of dependence in this analysis is represented in the network as a relationship *from* the file of use *to* the file of declaration.

The following relationships are considered as dependencies for the purposes of this research as previously applied by Sturtevant [2]:

- *From* the site of a function call *to* the site of the function's definition
- *From* the site of a class method invocation *to* the site of the class method's definition
- *From* the site of a class method invocation *to* the site of the class definition
- *From* the site of a subclass definition *to* the site of the parent class definition
- *From* the site of a non-trivial user-defined type instantiation *to* the site where that type is defined

The software package *Understand*, a static analysis tool developed by *Scientific Toolworks, Inc.,* has been applied to both views of the software under analysis in order to derive these relationships between C and C++ files in both systems.

4.2 Message Passing and Augmentation of Analysis

A common practice in modern software systems is the concept of *message passing*. In software engineering message passing is the act of a process constructing and sending a package of data to another process then responsible for its interpretation, and subsequent action based on the data contained in the message. The act of message passing is fundamental to the function of modern software systems, including communication within a program, across programs, and across computers (for example, an intra- or internet) [21]. Various techniques to achieve message passing may be applied, however they all fundamentally differ from the act of direct method invocation generally used to indicate dependence between two modules – as in the methodology described above. Because of the levels of indirection involved with message passing implementation, dependencies between components that send messages and components that receive messages are often difficult to detect using traditional static analysis.

The software system under analysis makes extensive use of this technique, obfuscating the actual architecture of the system when analyzed using traditional static analysis. To overcome this drawback the following analysis has been applied to augment the dependency list:

- For each event declared within the system, include a dependence *from* the file processing the event *to* the file posting the event.

Two methods of event dependence identification were attempted in order to fully capture the additional dependencies. The first method added instrumentation to the system process responsible for posting and distribution of internal events, capturing dependence information at runtime as the system was exercised. However, because this method required exercising all possible code paths within the system to ensure a complete analysis it proved unsatisfactory. The second method performed a second static analysis pass of the system, recognizing patterns

of event declaration and event handling applied within the software system – while this method proved robust in identifying a complete set of dependencies, it required intimate knowledge of the system under analysis making its direct application in other software systems difficult. In the future, it may prove useful to augment the analysis by *modeling* event posters and event handlers considered by the software firm to represent dependencies within its system. The act of modeling implementation-specific methods has been applied by other static analysis tools, such as *Coverity Prevent,* to handle similar firm-specific design patterns that require hints to the analysis engine as to their use. Results of the analysis both with and without this augmentation will be provided to allow effective comparison.

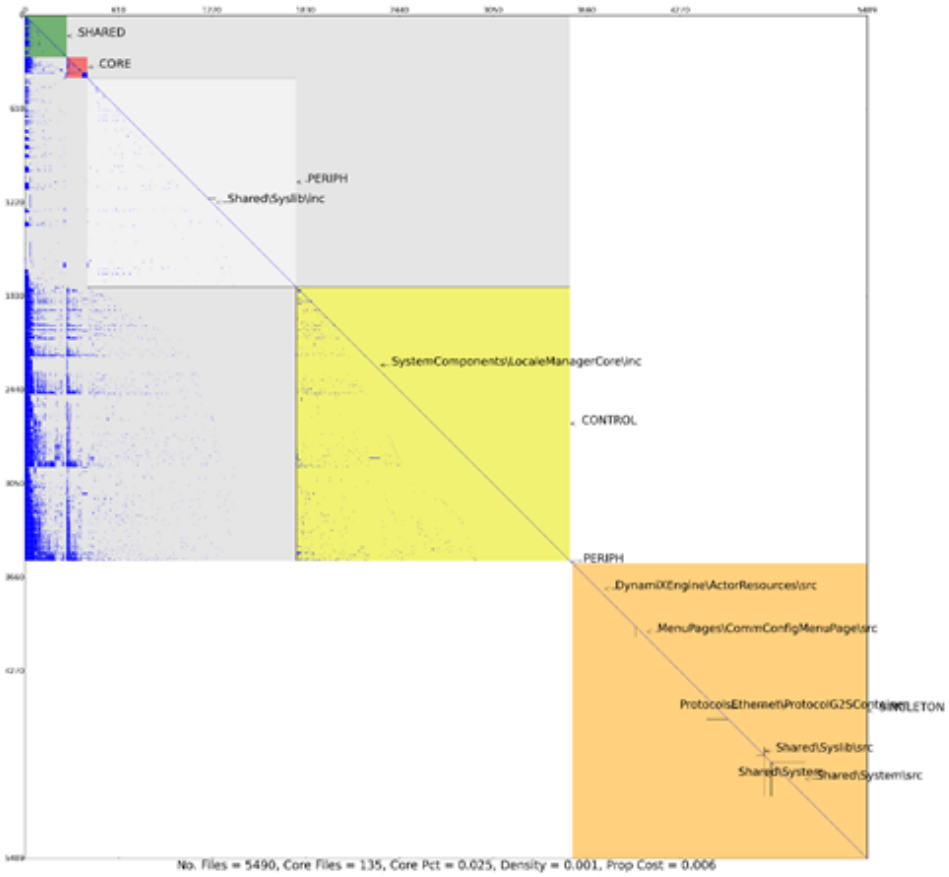
4.3 Pre-Refactoring Analysis

The analysis of both systems has been presented in three stages – the first presenting a system analysis using the classic MacCormack-Baldwin core-periphery method, the second additionally applying inter-file dependencies detected by the *Understand* static analysis tool, and the third additionally applying message passing dependencies identified within the system.

4.3.1 **Base Core-Periphery Analysis**

Error! Reference source not found.8 shows the results of base Core-Periphery analysis of the s oftware system which demonstrates several notable characteristics – it shows what appears to be an exceptionally small number of files designated Core and Shared, and an exceptionally large number of files designated as Control, Periphery, and Singleton. The Singleton entities represent those files which exist in an independent relationship network outside of the primary cluster of files – no detected relationships exist between these files and the remainder of the system. This relationship pattern demonstrates an exceptionally low level of coupling within the software system, and does not match the notional expectations of the system engineers. The Periphery in this case accounts for 25% of the total system, Control accounts for ~32%, and the Core only

2.5%. 35% of the system has been detected as a Singleton with no connection to the rest of the system. In an effort to further explore potentially missing relationships we explore other types of dependencies available through Understand static analysis.



[1]

Figure 8 Pre-Refactoring Core-Periphery Analysis using Classic MacCormack-Baldwin Relationships

[1] Subsequent to the main body of work for this thesis effort has continued on improved Core-Periphery segmentation. While primary findings remain unchanged updated images are included in Appendix A for reference.

4.3.2 **Extended Core-Periphery Analysis**

The extended Core-Periphery Analysis includes classic dependencies identified by MacCormack and Baldwin described in section 1.1 and enumerated in section 2.5 to additionally include the following:

- *From* the site of object initialization *to* the site of object definition
- *From* the site of object inheritance *to* the site of the inherited object's definition
- *From* the site of object implementation *to* the site of object declaration
- *From* the site of object override *to* the site of the overridden object's definition
- *From* the site of object use or modification *to* the site of object definition

In addition to considering a wider range of relationships, the use of the term "object" in these conditions allows finer granularity in contrast to considering specific software constructs such as functions, classes, sub-classes, and user-defined types. **Error! Reference source not found.**9 s hows the results of the modified analysis. While the outcome is closer to our expected results, the figure still shows an exceptionally small core comprising now ~2% of the total files in the system, and an extremely large periphery containing over 50% of the total system. The Singleton percentage has now dropped to 28%. We suspect information is still missing, and further explore augmenting the analysis.

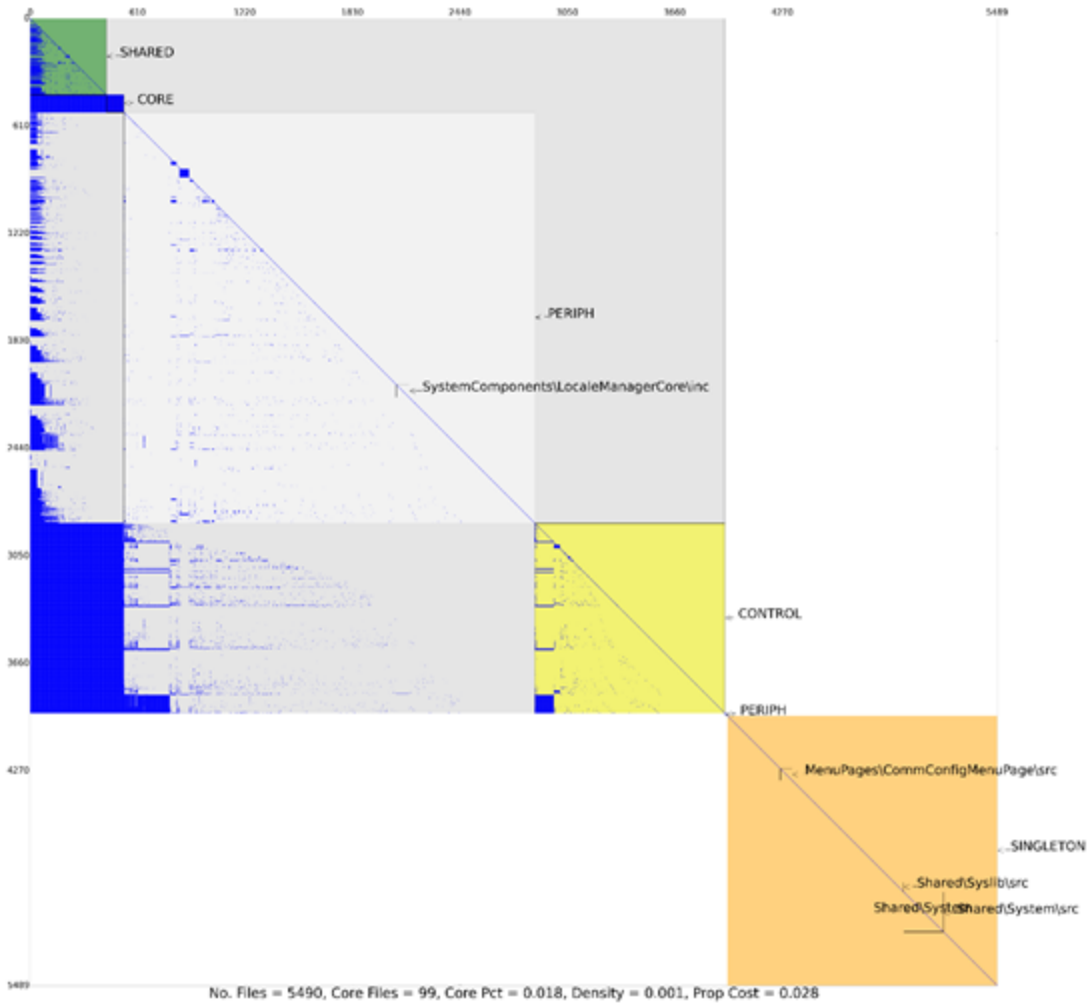


Figure 9 Extended Pre-Refactoring Core-Periphery Analysis

4.3.3 **Extended Core-Periphery Analysis with Message Passing**

The final pass of analysis applies the additional dependencies represented by information transmission between entities within the system as described in Section 4.2. **Error! Reference s ource not found.**10 displays the results of this analysis with an additional 1400 missing

dependencies, and shows the system with clear Core-Periphery module relationships. The Core has nearly doubled to 5% of files within the system, the Periphery now contains 34% of total, and Control and Shared-category files now represent a reasonable 18% each. 25% of the system is still detected as Singleton – a significant portion of these files represent menu pages designed for user configuration of the system's operating environment. The effects of the system's internal architecture on selection of area of refactor are discussed further in Chapter 5.

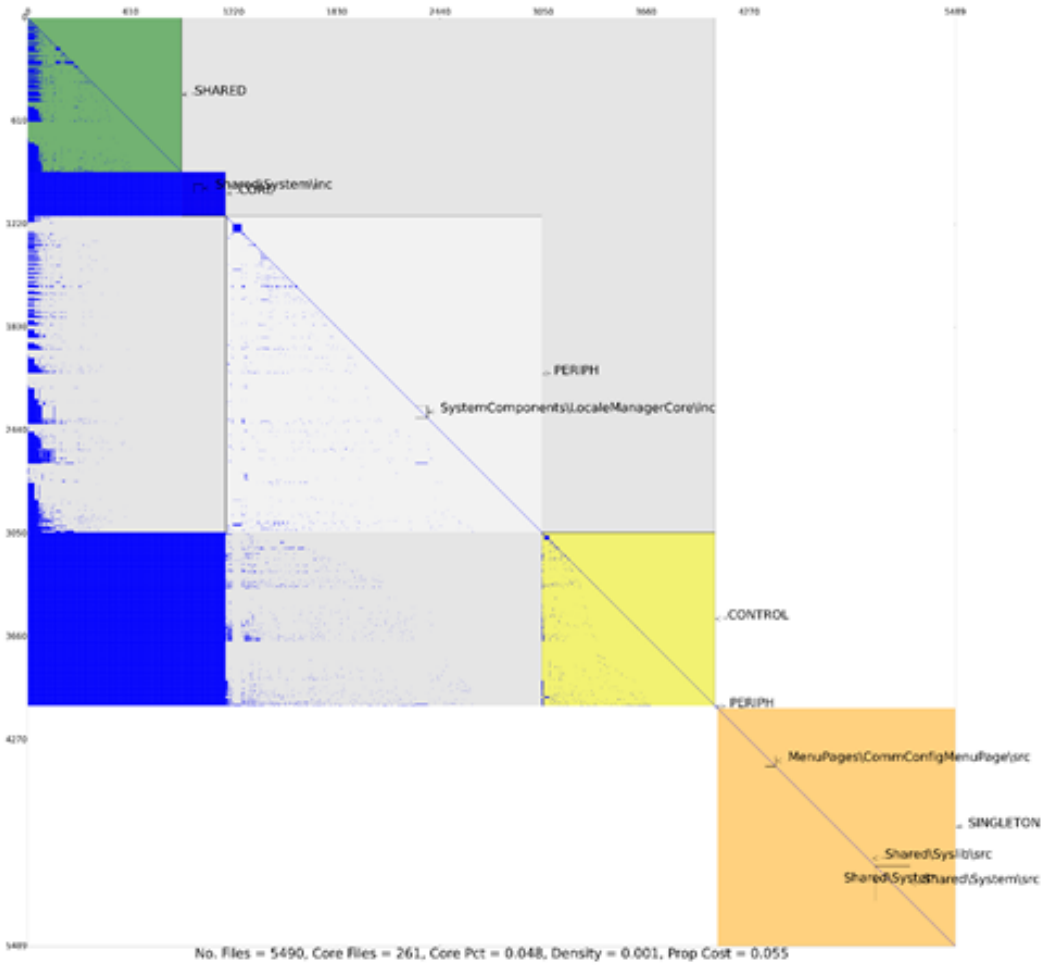


Figure 10 Pre-Refactoring Core-Periphery Analysis including Message Passing

4.4 Post-Refactoring Analysis

Error! Reference source not found.1 shows the results of the analysis applied above to the r efactored system in abbreviated form. The effects of additional message-passing dependencies are exaggerated in this view of the system with a notable initial absence of files in the Shared category, and the eventual emergence of the Core. We see the same progression from a largely hierarchical system to the Core-Periphery system we expect.

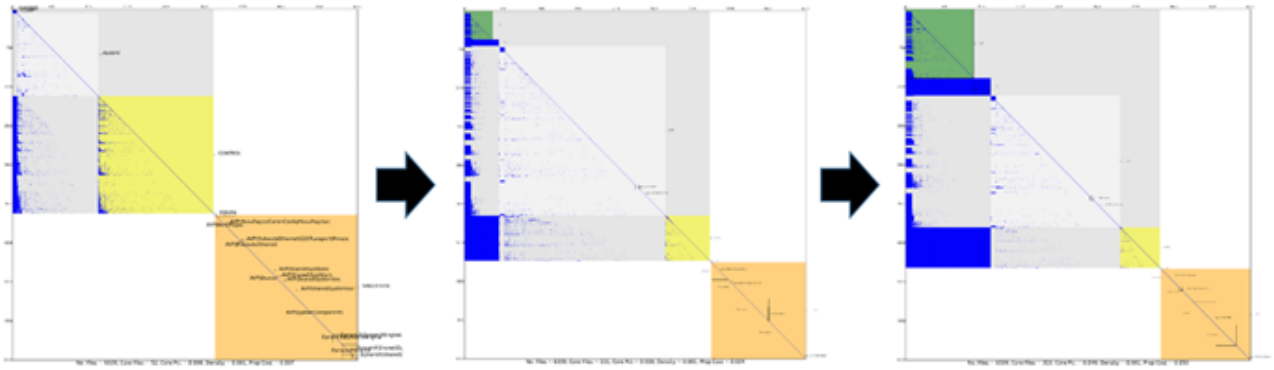


Figure 11 Post-Refactoring Core-Periphery Analysis Progression

4.5 Game-to-Foundation Refactoring

Extending Core-Periphery analysis to the combined Game-and-Foundation software package highlights the architectural results of the refactoring and its success in eliminating inter-module dependence. Baldwin and Clark [3] link the existence of off-diagonal dependence between modules to systemic inefficiency, creating cycles that require cross team conferences, trade-offs, and compromises to resolve. To address these inefficiencies, they describe a process by which design engineers and architects carry the knowledge of these previous difficulties forward as products evolve, solidifying decisions earlier in the development process. While these decisions restrict the viable solution space by designating certain design characteristics as “privileged”

they remove potentially troublesome off-diagonal dependencies both in the product and team structure. Figure 12 shows side-by-side comparison of a laptop computer's system structure as studied in Design Rules: The Power of Modularity [3]. The system structure shifts from ad-hoc with many off-diagonal dependencies to structured modules using design rules.

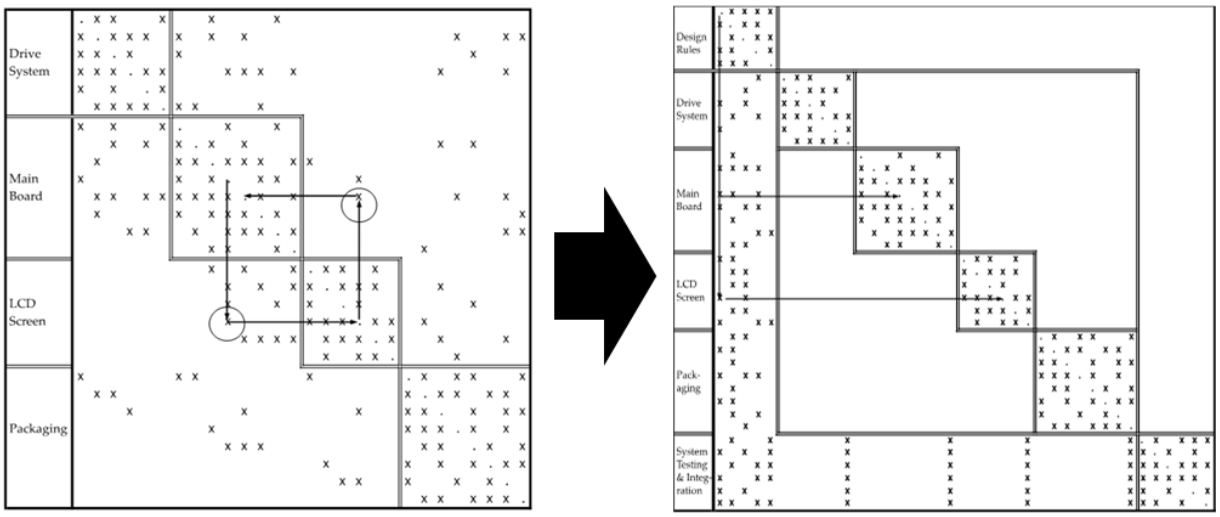


Figure 12 Baldwin & Clark Laptop System Modularization

This same shift is visible in the architectural structure of the pre- and post-refactored system. Figure 13 shows an extended Core-Periphery analysis now including the Foundation (in the upper left) and the Game (in the lower right), sorted with respect to directory structure. The analysis reveals undesirable inter-module dependencies in the pre-refactoring view, while the post-refactoring view shows these dependencies have since been removed. This demonstrates a clear modular decomposition as described by Baldwin and Clark, and mirrors the organizational modularization discussed in Chapter 3.

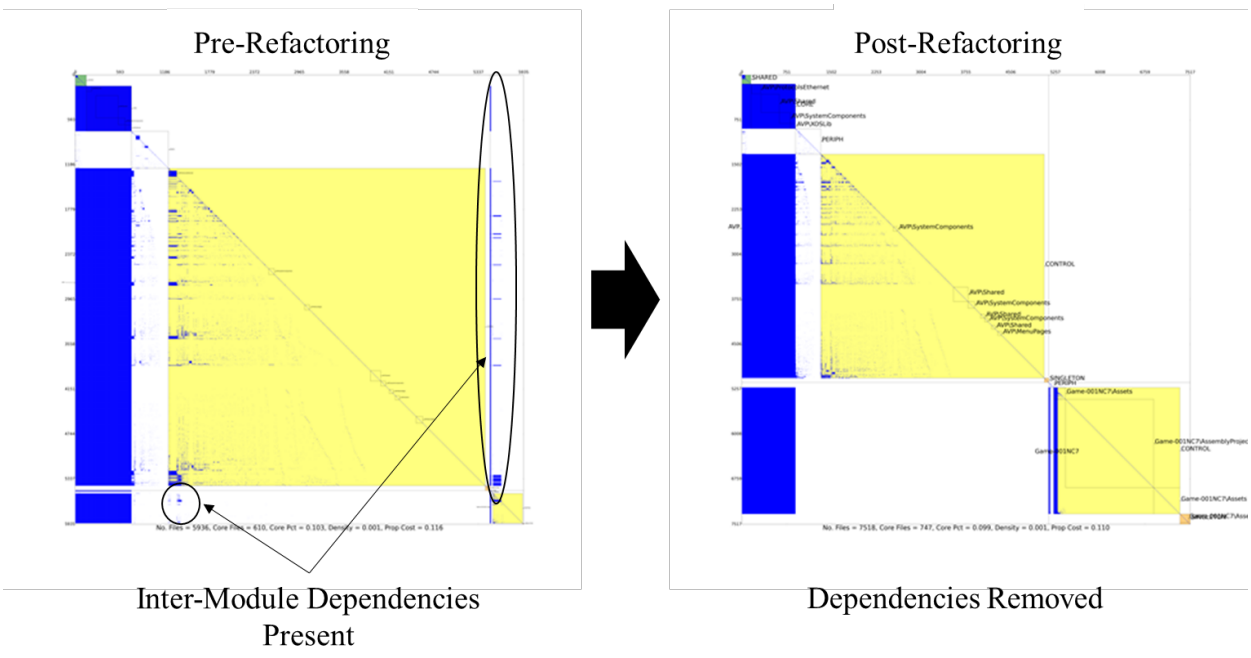


Figure 13 Pre- and Post-Refactoring System Modularization

4.6 Summary

Table 2 Summary of Dependence Analysis Methodologies summarizes the findings between the various methods of dependence analysis. While message passing accounts for only 1400 missing dependencies (4% of total) in both views of the system these dependencies account for up to 66% of dependencies for files considered Core and up to 75% of dependences for files considered Shared. They additionally reduce Singletons by up to 10% and represent sufficient coupling to reduce Periphery files by up to 20%. This pattern of behavior matches the intuitive expectation that Core files are more likely to pass information between themselves, and that there are significant amounts of internal system coupling not detectable through classic methods of static code analysis.

	Pre-Refactor			Post-Refactor		
	Base Analysis	Extended Analysis	Message Passing	Base Analysis	Extended Analysis	Message Passing
System type	Hierarchical	Hierarchical	Core-Periphery	Hierarchical	Hierarchical	Core-Periphery
# Direct Dependencies	31884	38315	39757	26024	35117	36416
Core Percent	2.46%	1.80%	4.75%	0.82%	1.75%	4.94%
Shared Count	272	434	911	28	525	1251
Core Count	135	99	261	52	111	313
Peripheral Count	1376	2346	1885	1521	3117	2399
Control Count	1783	1078	1024	2121	825	730
Singleton Count	1924	1533	1409	2617	1761	1646

Table 2 Summary of Dependence Analysis Methodologies

Chapter 5 Quantitative Impact of Refactoring

5.1 Introduction

In addition to analyzing the architectural impact of increased modularity through refactoring, this study also analyzes the quantitative impact of refactoring on both software systems. To understand the benefits gained and costs incurred by the underlying architectural changes the following measurements of file activity are derived from the firm's code base:

- Number of files changed per work item type (defect or new development)
- Number of in-file modifications required per work item type
- Number of unique authors per work item type
- Rate of change of modifications in areas of the code base directly impacted by the refactoring

Utilizing the API provided by the firm's version control software, defect and work tracking systems, and source code the following information has been derived for each revision of each file in both product lines:

- File name and path
- File revision number
- Date of modification

- Author
- Associated change request
- Change request type (defect or new development)
- Lines added, modified, and removed
- Age of change request prior to completion of work

5.2 Developer Workflow

Established practice among developers at the software organization provides a strong linkage between modifications required in the code base and the work item initiating the change. A common defect workflow applied at the firm begins with a defect identified by either internal test or the quality assurance department entered into a *defect tracking database*. This item is then synchronized to the *change request system* utilized by the software development teams, and eventually linked to the *version control system* once a modification is committed that corrects the defect. In this manner a direct chain of responsibility is established and maintained directly from the uniquely identified defect to the changes effected in the software system. The following diagram depicts the complete workflow for both defects and new development:

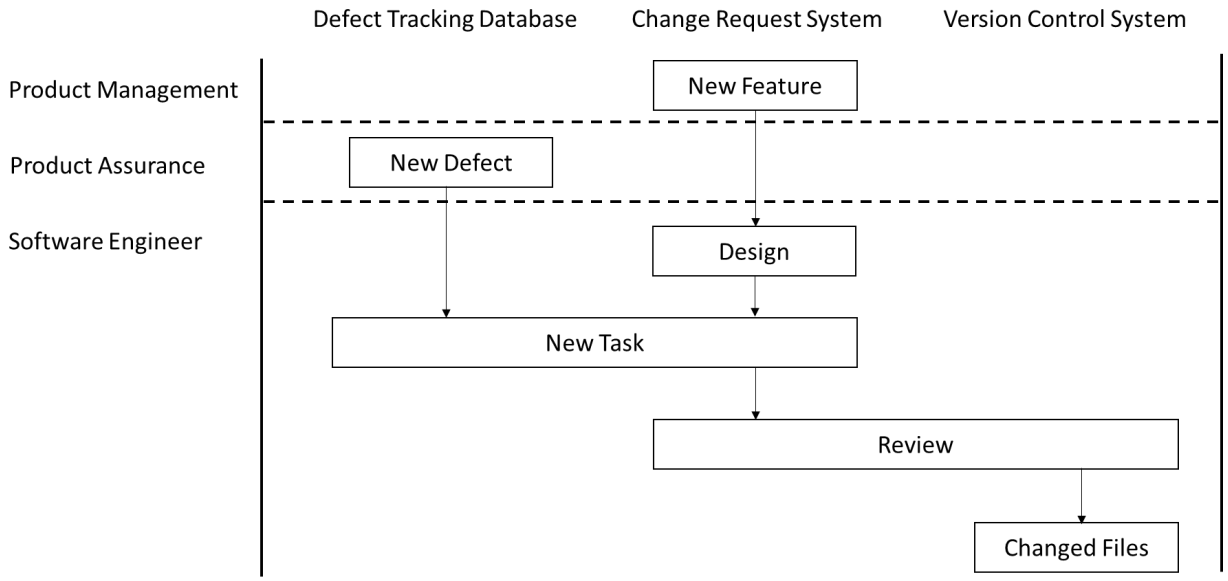


Figure 14 New Feature and Defect Workflow

5.3 Data Gathering Methodology

The following rules were applied to include or exclude files from analysis:

- Third-party or automatically generated files have been excluded from analysis
- Files excluded from the released product have been excluded from analysis
- Files that do not include at least 1 line of executable code have been excluded from analysis
- Header files have been excluded from analysis

We differentiate files as belonging into two groups, either *along the area of refactoring* or *rest of the system*. In this analysis, "along the area of refactoring" refers to those modules that directly contain modifications to facilitate communication between the game and that component. In general these files contribute to the accounting, security, configuration, and management of the game state machine.

5.4 Statistics of Refactoring and Complexity

For both pre- and post-refactoring releases of the system, approximately 2400 files were analyzed following the exclusions listed above. Over the lifetime of the pre-refactoring system approximately 150,000 modifications have been made in 17,000 files by just under 100 engineers; 2/3^{rds} of those modifications corrected defects and the remainder implemented new features. The post-refactoring system represents a much shorter development timeline with a total of 25,000 modifications in 10,000 files by 50 engineers – with the same ratio of defects-to-features.

Statistical analysis applied in this study follows the approach used by Kim et al. [10] to analyze refactoring by Microsoft engineers in the Windows Vista operating system. This analysis uses a two-sided, unpaired Wilcoxon rank sum test to identify statistically significant differences in a variety of software engineering metrics between the area of refactoring and the rest of the system in the pre-refactoring product, and identifies statistically significant impacts of refactoring in the post-refactoring product. This analysis begins with a discussion of code churn due to its level of relative importance to the software organization during this period of refactoring.

Refactoring impact on the software metrics between the pre- and post-refactoring systems has been normalized to reflect the relative impact on the primary area of refactor versus improvements made in the remainder of the system. Kim describes this process:

> *"Suppose that the top 5 percent of most preferentially refactored modules decreased the value of a software metric by 5 on average, while the bottom 95 percent increased the metric value by 10 on average. On average, a modified module has increased the metric value by 9.25. We then normalize the decrease in the top 5 percent group (-5)*

and the increase in the bottom 95 percent group (+10) with respect to the absolute value of the average change (9.25) average of all modified modules, resulting in -0.54 and +1.08 respectively"

Table 3 contains results of this analysis.

	Pre-Refactoring			Δ Pre-Post		
	Area of Refactor	Rest of System	p-value	Area of Refactor	Rest of System	p-value
Churn						
# Changes	1376	6152		671	1674	
# Changed Files	2001	17467		2871	8927	
Total Churn	15.93	13.13	0.66	(1.18)	(0.96)	0.48
Defect Churn	0.10	0.03	0.04	(4.44)	(0.14)	0.44
Feature Churn	0.12	0.04	0.07	1.76	0.81	0.49
Size						
Mean LOC	510.30	157.79	0.03	(1.65)	1.66	0.09
# Files	284.00	2294.00		471.00	1878.00	
Complexity						
Visibility Fan In	573.70	235.89	0.01	(1.96)	(0.76)	0.74
Visibility Fan Out	457.83	460.31	0.89	0.70	0.69	0.48
Mean Cyclomatic	23.79	9.78	0.05	(0.45)	1.36	0.04
Defects						
Total Defects	819	4993		294	756	
Files Changed for Defects	1062	10852		569	1718	
Mean Defects per File	8.60	3.55	0.48	(1.94)	(0.76)	0.54
Mean Files Changed per Defect	1.36	2.44	0.00	0.54	(1.39)	0.09
Mean Modifications per Defect	4.08	6.48	0.02	(0.67)	(1.08)	0.37
Mean Number Developers per Defect	1.16	1.07	0.81	(1.99)	0.75	0.92
Mean Age of Defect (days)	24.36	16.01	0.04	0.88	1.03	0.61
New Development						
Total New Features	557	1669		377	918	
Files Changed for Features	939	17615		2302	7209	
Mean Files Changed per Feature	1.64	3.87	0.00	1.40	0.90	0.67
Mean Modifications per Feature	5.73	9.74	0.04	0.85	(1.46)	0.11
Mean Number Developers per Feature	1.19	1.12	0.41	(1.79)	(0.80)	0.23
Mean age of feature request	88.94	62.08	0.05	1.92	0.77	0.54

Table 3 Comparison of Metrics and Significance Pre- and Post-Refactoring

5.5 Summary of Results

5.5.1 **Churn**

The results of the analysis on the pre-refactoring system indicate a statistically significant difference along only one primary metric – the number of files modified to correct defects as a ratio of the total number of files within the module as defined by ($\frac{number\ files\ changed\ for\ defects}{total\ files\ in\ module}$). Along the area of refactor a mean 10% of the files within the module required modification in order to correct a defect, as opposed to a mean 3% in the rest of the system. On the other hand, while more than three times the number of files on average required modification in order to implement new features, and more than 3.5 times the number of files required modification overall, no other results were found to be statistically significant in comparison to the distribution of results from the remainder of the system. It is notable, however, that levels of total churn ($\frac{total\ changed\ files}{total\ files\ in\ module}$) and levels of defect churn were both reduced in the post-refactoring product while levels of feature churn on average increased.

5.5.2 **Size and Complexity**

Comparing the area of refactor versus the remainder of the system indicates a significantly larger number of both average number of lines of code in the files identified for refactor (an average of three times the number of executable lines of code) as well as the levels of McCabe cyclomatic complexity (an average of 2.5 times more complex). Notably, the VFI levels of modules targeted for refactoring are significantly higher than the remainder of the system, while the VFO levels show no statistical difference.

Following the refactor we find that lines of executable code in the area of refactor showed a relative decrease, however the distribution is not statistically different from the remainder of the system. On the other hand, we see mean levels of cyclomatic complexity showing a small but statistically significant ($p < 0.04$) decrease in the area of refactor as opposed to small increase in the rest of the system.

5.5.3 **Defects**

Applying the method of analysis to the impact of defects on the area of refactor finds that the number of files changed per defect, as well as the number of modifications required within those files, is significantly lower than the remainder of the system, however the mean amount of time required to correct those defects was approximately 50% higher on average. The number of developers involved in correcting defects was on average just over one, consistent with the number of developers required in the remainder of the system.

Significance analysis shows no statistically significant shift in the area of refactor as compared to the remainder of the system. Modifications required to correct defects per-file showed a general decrease, while the average time required to correct defects increased in both cases.

5.5.4 **New Development**

We find that the impact of new development is overall very similar to the observed behavior of defects. The mean number of total files changed, as well as the number of required modifications within those files, is significantly lower in the area of refactor than the remainder of the system. However, the average age of a new feature is approximately 50% higher in the area of refactor, indicating an increase in implementation time. As before the number of

developers involved in the implementation of new features is just over one in all areas of the system under analysis.

Our analysis found no statistically significant changes for new feature development between the area of refactor and the remainder of the system post-refactoring. In general the total number of files requiring modification to implement new features as well as the time required to implement new features shows a trend of increasing following the refactor.

5.6 Multivariate Regression Analysis

5.6.1 **Pre-Refactoring Defect Density and Related Software Metrics**

In order to identify potential trends and ideal areas of focus for future refactoring effort we perform multivariate regression analysis on the data with a primary focus on the relationships between defects within a module and other metrics both within and outside the area of refactor. This analysis is performed on the pre-refactoring data set in order to capture metric relationships without the impact of refactored modules.

Table 44 shows the results of regression analysis predicting the *defects per file* of a given module as other metrics shift in the area of refactor. In order to identify potential model variables we first perform a correlation analysis between all measured metrics and the number of defects per file, selecting metrics with correlation $p < 0.05$ as our initial model, a dummy variable (Location) is added to the model in order to differentiate modules along the area of refactor from those in the remainder of the system. We then begin regression by assigning defects detected per file as the dependent variable, and assigning independent variables as { LOC, Location, McCabe's Cyclomatic Complexity, VFI, VFO, Average Files Changed per Defect, Mean Developers per

Feature, Amount of Feature Churn, Amount of Total Churn, and # Commits to the Module }. Stepwise backward and forward refinement is then applied to reduce the number of model variables.

Coefficients:	**Estimate**	**Std. Error**	**t value**	**Pr(>|t|)**	
(Intercept)	-3.8911925	1.370915	-2.838	0.0176	*
LOC	0.0144937	0.0028471	5.091	0.00047	***
Cyclomatic Complexity: Low	4.1724378	1.7936651	2.326	0.04232	*
Cyclomatic Complexity: Moderate	4.4941601	1.5144789	2.967	0.01411	*
Cyclomatic Complexity: Very High	8.8401945	2.3565035	3.751	0.00378	**
Total Churn	0.7006372	0.024185	28.97	5.60E-11	***
Total Commits	-0.0004372	0.0001991	-2.196	0.05277	.

Table 4 Regression Analysis of Software Metrics to Defect Density Prior to Refactoring

The regression results verify our intuition that the number of defects per file is primarily impacted by lines of code in a file and total amount of churn within its module as defined by total changes within a module divided by total files within the module. This is indicative not only of significant rework related to feature implementation, but defect correction – an indication that defect changes either themselves introduce defects, or do not completely correct the targeted defect.

Additionally, the results are indicative of problematic modules sharing excessive levels of internal coupling requiring modifications for both features and defects to span a high proportion of files within a module, reducing change locality and correspondingly introducing further defects. These results also verify our expectations of the impact of McCabe's cyclomatic complexity as it becomes not only increasingly impactful to levels of defects but simultaneously more statistically significant. The results indicate that such factors as location of the file in the

system relative to the area of refactor, number of developers per feature, and amount of feature churn are not significant indicators of defects.

5.6.2 **Delta Analysis of Defect Density, Cyclomatic Complexity and Related Metrics**

In an effort to identify impactful modifications performed during the refactoring, Table 55 presents multivariate regression of contributory factors in changes for both defects and cyclomatic complexity between the Pre- and Post-Refactor systems. As before we begin the regression by performing a preliminary correlation test and accepting variables with $p < 0.05$ as our initial model. Stepwise forward and backward regression is then performed.

Coefficients:	**Estimate**	**Std. Error**	**t value**	**Pr(>|t|)**	
(Intercept)	-0.36299	0.20486	-1.77	0.09818	.
Δ Total Churn	1.10225	0.07285	15.13	4.54E-10	***
Δ Cyclomatic Complexity	0.42727	0.09614	4.444	0.00056	***

Table 5 Regression Analysis of Refactored Metrics to Defects per File

Regression analysis of the impact of refactored metrics on defect density reinforces the prior results and again follows expectations showing significant correlation between a pair of primary factors. Modules displaying an increase in defect density shared a corresponding increase in total churn, following the initial regression results of the pre-refactored system. The same increase in defect density between the pre- and post-refactored systems can also be seen in modules that shared an increase in cyclomatic complexity – a factor specifically targeted by the refactoring engineers for improvement. The regression analysis did not show a statistically significant correlation between defect density and modules both within and outside of the area of refactor, the number of new features introduced, and other metric deltas.

In order to explore the impact of refactoring on cyclomatic complexity in various software modules Table 6 presents multivariate regression relating contributory factors to changes in the cyclomatic complexity following the same method outlined above.

Coefficients:	**Estimate**	**Std. Error**	**t value**	**Pr(>|t|)**	
(Intercept)	-0.44593	0.33252	-1.34	0.20956	
Δ LOC	0.25232	0.06567	3.842	0.00326	**
Δ Total Churn	-1.56094	0.27492	-5.68	0.0002	***
Δ Defects per File	1.2872	0.23472	5.484	0.00027	***
Δ Mean Age of Defect	0.31289	0.09296	3.366	0.00717	**
Δ Defect Churn	0.4227	0.14489	2.917	0.01537	*
Δ Mean Modifications per Defect	-0.14623	0.06175	-2.37	0.03939	*

Table 6 Regression Analysis of Refactored Metrics to Cyclomatic Complexity

Results of the regression analysis reinforce the strong correlation to changes in defects per file as shown above, however complexity shares an inverse relationship to total churn. As expected, the regression indicates a direct relationship with lines of executable code as well as the number of files modified within a module to correct defects and the average age of a defect before correction. Finally, the analysis indicates a direct relationship between change in cyclomatic complexity and changes in the number of files necessary to correct defects, but an inverse relationship with the change in number of lines of code necessary to correct defects within those files.

5.7 Discussion of Results

In order to effectively correlate results to the initial research hypotheses we divide the discussion of the results of this analysis into two sections with the first discussing results of hypotheses meant to test software metrics affecting the selection of modules for refactor, and the second discussing results of refactoring on those software metrics.

5.7.1 **Attributes of Modules Targeted for Refactor**

5.7.1.1 ***H1 – VFI / VFO***

While we find that modules targeted for refactoring shared an elevated level of VFI, their VFO levels were not statistically different from the remainder of the system. While contrary to our initial hypothesis this behavior is consistent with Kim et al's [10] findings while studying refactoring efforts at Microsoft – engineers targeted areas of the code base with high levels of undesirable dependence and higher levels of test coverage and verifiable behavior. Sturtevant [2] describes files in the Shared category as "relied upon by a large portion of the system" with the "potential to be self-contained and stable," and files in the Core category as "containing large cycles in which components are directly or indirectly co-dependent." The VFI / VFO levels of modules targeted for refactor indicate that these files share markers identified in this research and previously as predisposing them to targeted refactoring. To better understand this impact on the selection of refactored modules, Table 7 compares the distribution of categories in the pre-refactored system. We find that while files selected for refactoring do share elevated percentages of files in the Shared and Core categories, the distribution of these files is the same as the remainder of the system, eliminating Core-Periphery categorization as a leading indicator of preference for this refactoring. This hypothesis is therefore rejected.

	Area of Refactor	**Rest of System**
Shared %	0.19	0.16
Periphery %	0.40	0.33
Control %	0.16	0.19
Core %	0.06	0.05
Singleton %	0.18	0.27
P-value	1	

Table 7 Core-Periphery Distribution of Files

5.7.1.2 ***H2 – Defect Density***

We find that while the area of refactor averages more than twice the number of defects present in other areas of the software system, and accounts for 14% of overall defects, there is no statistically significant difference in the distribution per module versus the rest of the system. The great difference in these values is driven primarily by a high number of defects in a small subsection of modules targeted for refactoring, which means that while distribution is lop-sided the difference is primarily noise. While these data refute the hypothesis that defect density within the area of refactor is statistically higher than other areas of the system, the question arises of what caused the refactoring engineers to refactor the entire targeted subset instead of those modules displaying the highest defect density. Dependence analysis between these modules and interviews with participating engineers find strong first and second-degree coupling between the modules with the highest defect density and the remainder of the modules selected for refactor. These results indicate that while high defect density is one driving factor in selecting target modules its value is impacted by strength of connectedness and defect density the target module shares with the rest of the system.

While at first seemingly contradictory to our findings that on average *fewer* files were modified to correct defects (1.36 vs. 2.44 average files), we find this is due to the increased density of the files in general. Lines of executable code per file were found to be more than three times higher in the area of refactor.

5.7.1.3 ***H3 – Cyclomatic Complexity***

Results of pre-refactoring analysis find the levels of cyclomatic complexity in the area of refactor is greater than 2.5 times levels in the remainder of the system under analysis. This value is additionally $p<0.05$ significant when compared with the distribution of the remainder of complexity metrics indicating not only a strong correlation with cyclomatic complexity and those

modules selected for refactor, but also indicating these modules reside in an entirely separate classification of complexity (Moderate-High as opposed to Low-Medium). We find H3 confirmed by these results.

5.7.1.4 ***H4 – New Feature Development***

The area of refactor accounted for fully 25% of total new feature development in the software system while containing only 11% of the total size of the system in executable lines. Even so we find the mean time required for full feature development in the area of refactor to be 43% longer than the mean time elsewhere in the system, indicating that while feature demand was significantly higher engineers encountered difficulty in implementation. We again find the density of changes required to implement features to be statistically lower in the area of refactor, which while at first leads us to believe the modules share decreased levels of coupling is actually indicative of the increased number of executable lines of code. We find H4 confirmed by these results.

5.7.2 **Impact of Refactoring on Software Metrics**

5.7.2.1 ***H5 – VFI / VFO***

Analysis of the VFI and VFO attributes indicate a general trend of reduced VFI following refactoring, but an increased VFO across the system. No statistical difference was found between the area of refactor and the remainder of the system, causing us to reject this hypothesis. No evidence was found that refactoring alone was responsible for changes in inter-module dependence.

5.7.2.2 ***H6 – Defect Density***

While the results of analysis indicate a decrease more than twice the magnitude of the rest of the system in defects per file along the area of refactor, it has been found to not be statistically

significant. This reduction was, again, driven by significant changes in a subset of modules included in the refactoring effort, and was not indicative of improvement elsewhere. Interestingly, while the number of files requiring modification to correct defects increased in the area of refactor, the modifications necessary within those files decreased. We see this result reflected in the decreased code density within the area of refactor shown by a decrease in the lines of executable code accompanied by an increase to almost double the number of files within these modules. As expected, these results are consistent with the regression analysis indicating a strong correlation between defect density and total churn – a metric primarily driven by the relationship of defects to number of files in the module. Despite the significant decrease in mean defects per file, we must refute this hypothesis as a side-effect of improvements made elsewhere in the system and not directly a result of the refactoring effort.

5.7.2.3 ***H7 – McCabe's Cyclomatic Complexity***

The findings indicate not only a strong relative decrease in the cyclomatic complexity of modules along the area of refactor, but a strong relative increase in the same across the remainder of the system, resulting in a statistically significant difference in the distributions of this metric. Regression analysis of this metric finds the strongest correlation between total churn and defects per file, with a secondary relationship with executable lines of code, and the length of time required to correct defects within the module. These results confirm our notional expectations of the behavior and impact of this value – increased complexity increases the defects detected within a module, but itself decreases as the logic within those files is effectively dispersed amongst other files. This presents as a negative relationship with total churn within the analyzed modules. An unobvious result is the inverse correlation between mean time required to correct defects and cyclomatic complexity, however upon further inspection this relationship is a

result of decreased levels of engineering time (increased velocity) negatively impacting the maintainability of the code base through an increase in complexity.

Based on the statistical significance of the decrease in cyclomatic complexity as a result of this refactoring, we confirm this hypothesis.

5.7.2.4 ***H8 – New Feature Development***

We again see the majority of new feature development centered on the area of refactoring, increasing from 25% of total features previously noted to just under 30% in the area of refactoring – now representing 20% of the total system's files. It is notable that mean files necessary for new features, mean modifications per feature, and time required to implement features all increased in the area of refactoring, with the remainder of the system performing better in only one metric as compared to the pre-refactored system. Development time required for new features increases across the system following the refactoring. We refute this hypothesis based on these results.

Chapter 6 Conclusions and Future Work

As software systems age and legacy systems comprise larger portions of existing software it is important for leading software development organizations to carefully consider strategies for refactoring and modularization. In order to effectively refactor – maintain external function while improving internal implementation - part of this strategy should incorporate an understanding of the structure of interdependence within the system in addition to lines of code, defect density, cyclomatic complexity, and other classic markers. Additionally, even when refactoring is targeted effectively the measurable effects of refactoring should be closely

monitored, as this and previous research has shown that some perceived impacts may either not align with realized improvement, or may become locked in a "worse before better" trend requiring patience, monitoring, and cultivation until intended benefits become apparent.

This research examined a significant refactoring effort at a leading software development firm, seeking to identify markers along the area of refactor indicating potential markers of predisposition to a targeted refactoring effort. We then analyzed metrics measured in both the pre- and post-refactored systems in order to identify realized systemic benefits, as well as isolate those benefits which uniquely impacted the area of refactor. As part of this analysis we applied the MacCormack-Baldwin Core-Periphery analysis in order to gain an understanding of the underlying architecture of the system as it applied to both the area of refactor and the remaining system. An improvement to this analysis intended to capture potentially missing dependencies was proposed, implemented, and applied to provide the best architectural representation possible.

Markers of Targeted Refactoring	**Confirmed?**	**Comments**
H1. Visibility Fan In / Visibility Fan Out	No	Refactored components displayed elevated VFI levels
H2. Defect Density	No	Refactored components displayed highly elevated code density per file
H3. McCabe's Cyclomatic Complexity	Yes	Refactored components displayed elevated MCC levels
H4. New Feature Development	Yes	Refactored components accounted for significantly more new features
Effects of Refactoring		
H5. Visibility Fan In / Visibility Fan Out	No	Identical distribution of architectural categories
H6. Defect Density	No	Refactored components displayed significantly decreased code density per file
H7. McCabe's Cyclomatic Complexity	Yes	Displayed a negative correlation with time invested per defect corrected
H8. New Feature Development	No	Feature Development time increased system-wide following refactoring

Table 8 Summary of Hypotheses and Findings

While the refactoring engineers involved in the software system's development did not conduct a robust analysis ahead of the refactoring effort we analyzed a common set of software metrics including levels of architectural complexity, lines of code per file, levels of churn for both fixing defects and implementing new features, and McCabe's cyclomatic complexity. These values were then tested for statistical significance when compared to the remainder of the software

system to determine any applicable trends. Several notable distinctions were found that, while not formally taken into account at the time of refactor, demonstrate indicators of targeted refactoring. These factors included increased levels of visibility fan in, high levels of defect churn, significantly elevated levels of code density per file, and higher amounts of time necessary for implementing new features and correcting defects. While no correlation could be identified between the Core-Periphery architectural category of files identified for refactor versus the remainder of the system, files in the "Shared" and "Core" area of the system appear to display attributes indicated as key indicators of preferential refactoring in other research.

Churn	**Indicator of Refactoring?**	**Improved by Refactoring?**
Total Churn	No	No
Defect Churn	Yes	No
Feature Churn	No	No
Size		
Mean LOC	Yes	No
Complexity		
Visibility Fan In	Yes	No
Visibility Fan Out	No	No
Mean Cyclomatic	Yes	Yes
Defects		
Mean Defects per File	No	No
Mean Files Changed per Defect	Yes	No
Mean Modifications per Defect	Yes	No
Mean Number Developers per Defect	No	No
Mean Age of Defect (days)	Yes	No
New Development		
Mean Files Changed per Feature	Yes	No
Mean Modifications per Feature	Yes	No
Mean Number Developers per Feature	No	No
Mean age of feature request	Yes	No

Table 9 Summary of Lead Indicators of Refactoring and Improvements Realized by Refactoring

Interestingly the research found evidence of phenomena the refactoring engineers had experienced but was not explicitly tested for. While performing regression analysis between cyclomatic complexity and various other code metrics the research found a negative correlation between time invested correcting a defect and increases in cyclomatic complexity within those files. This finding displays indications of decreased code maintainability as a result of increased code development velocity, a behavior notionally understood by the developers and system maintainers to be true.

These findings also bear managerial implications when determining which indicators of code health to monitor during a targeted refactoring effort. While it is likely to see incidental improvement in the key code health metrics analyzed here as a result of architectural refactoring, these metrics are intended to measure localized health of software systems and not system-level improvement. In this case attention should instead shift to architectural structure and inter-module dependencies to determine success. This is the case for the software system under analysis – architectural improvements have been realized as a result of this targeted refactoring as evidenced in Figure 13, however we see limited improvement in the selected set of traditional software health metrics outside of those specifically targeted by engineering. Software architects and development managers should therefore be careful to consider the locality and intent of refactoring when determining both which metrics to measure and the tools used to measure them.

6.1 Future Work

This research presents the realized effects of refactoring and attributes found to be common amongst refactored components of a complex software system, and while this analysis is performed at a leading software development organization these results may not apply elsewhere. It is important to understand not just the metrics measured as lead indicators of

refactoring, but the underlying architecture and structure of the software that leads to these metrics. In the case of the system under analysis we find that the area of refactor is comprised primarily of files many times larger than the remainder of the system, with fewer files in each module. This means that measurements of file-based churn will skew lower for these modules, while interconnectedness metrics will skew much higher as a higher percentage of these files will likely require modification per defect or feature. This skewness is compounded by the majority of refactoring research and analysis being performed after the change has already taken place, analyzing what has already happened. Future work should focus not only on expanding indicators of refactoring and the benefits and risks of the activity, but should begin to collate the various studies that have been performed to prescriptively identify areas of ideal refactor. Elevated levels of VFI in refactored components of this study, and results of related existing research, suggest it is likely that a targeted refactoring effort within "Shared" architectural components with high levels of code density and cyclomatic complexity would meet with a great deal of success. The results of such a targeted refactoring experiment would go far in untangling the contradictory reports of refactoring benefit and cost.

A secondary direction includes the exploration of hidden dependencies within software systems as discussed in Chapter 4. While this research extracted message passing dependencies in order to augment existing analyses, the approach utilized here is highly specialized. A generalized approach to such analysis would be much preferred to the by-hand modeling required currently. Additionally, the problem statement should be expanded from "identifying dependence between software modules" to "identifying paths of information exchange through software." With this extension other forms of information transfer become open for inclusion, such as disparate files modifying external non-compiled configuration files, or processes utilizing shared memory to facilitate communication. As evidenced by the significant number of files identified as

"Singleton" during this research there still remains a significant amount of inter-module dependence and communication not captured by today's tools.

6.2 Concluding Remarks

At the outset of this research we expected to find a high level of correlation between architectural complexity and components identified intuitively by engineers for facilitating modularity and communication within their software. While the refactored modules did in fact share a great deal of statistically significant differences between this area and the remainder of the system, the distribution of architectural complexity of the individual pieces of the refactored modules matched the overall distribution of complexity almost identically. While it is not surprising that these modules should span multiple areas of architecture that they would match so well was an interesting finding. It is possible this is a side-effect of the nature of the system under analysis, or that the intuition of the developers simply did not match our expectations. Regardless, a targeted formalized refactoring effort utilizing the findings of this and other research to identify ideal refactoring components would be a natural extension of this work.

Finally, while this research set out to formally measure the decisions of engineers responsible for some of the most complex software developed by this firm the sheer amount of data available for analysis threw into stark contrast the data that was not available – the internal complexity of the software system is understood by its developers at a high level, however the actualization of the architecture and its shifts over time have had significant impact on both the organization and the product. With the interplay between staff, management, product direction, and architectural shift, answering the deceptively simple question, "What do I refactor for the most benefit?" becomes at once both intimidating and compelling.

Chapter 7 Bibliography

[1] Lehman, Manny M. "Laws of software evolution revisited." *Software process technology*. Springer Berlin Heidelberg, 1996. 108-124.

[2] Sturtevant, D. (2013). *System Design and the Cost of Architectural Complexity*. Massachusetts Institute of Technology.

[3] C. Y. Baldwin and K. B. Clark, Design rules: The power of modularity, Volume 1 vol. 1: The MIT Press, 1999.

[4] Sanchez, R., & Mahoney, J. T. (1996a). Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management Journal, 17*, 63-76.

[5] Gauthier, R. L., & Ponto, S. D. (1970). Designing systems programs.

[6] MacCormack, A., Rusnak, J., & Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science, 52*(7), 1015-1030. doi: 10.1287/mnsc.1060.0552

[7] Mens, T., & Tourwe, T. (2004). A survey of software refactoring. *Ieee Transactions on Software Engineering, 30*(2), 126-139. doi: 10.1109/tse.2004.1265817

[8] W. Opdyke, Refactoring, Reuse & Reality, Lucent Technologies, Murray Hill, NJ, USA, 1999.

[9] Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using metrics to evaluate software system maintainability. *Computer*, *27*(8), 44-49.

[10] Kim, M., Zimmermann, T., & Nagappan, N. (2014). An Empirical Study of RefactoringChallenges and Benefits at Microsoft. *Ieee Transactions on Software Engineering, 40*(7), 633-649. doi: 10.1109/tse.2014.2318734

[11] P. Weissgerber and S. Diehl, “Are refactorings less error-prone than other changes?” in Proc. ACM Int. Workshop Mining Softw. Repositories, 2006, pp. 112–118.

[12] J. Ratzinger, T. Sigmund, and H. C. Gall, “On the relation of refactorings and software defect prediction,” in Proc. ACM Int. Working Conf. Mining Softw. Repositories, 2008, pp. 35–38

[13] Steward, D. V. (1981). The design structure system- A method for managing the design of complex systems. *IEEE transactions on Engineering Management*, *28*(3), 71-74.

[14] K. T. Ulrich and S. D. Eppinger, Product design and development, first ed.: McGraw-Hill, 1995.

[15] Eppinger, S. D., Whitney, D. E., Smith, R. P., & Gebala, D. A. (1994). A MODEL-BASED METHOD FOR ORGANIZING TASKS IN PRODUCT DEVELOPMENT. *Research in Engineering Design-Theory Applications and Concurrent Engineering, 6*(1), 1-13. doi: 10.1007/bf01588087

[16] Sosa, M. E., Eppinger, S. D., & Rowles, C. M. (2007). A network approach to define modularity of components in complex products. *Journal of Mechanical Design, 129*(11), 1118-1129. doi: 10.1115/1.2771182

[17] Rusnak Jr, J. J., & Advisor-MacCormack, A. (2005). *The design structure analysis system: a tool to analyze software architecture*. Harvard University.

[18] Conway, M. E. (1968). How do committees invent. *Datamation*, *14*(4), 28-31.

[19] Bailey, S. E., Godbole, S. S., Knutson, C. D., & Krein, J. L. (2013, October). A decade of Conway's Law: A literature review from 2003-2012. In *Replication in Empirical Software Engineering Research (RESER), 2013 3rd International Workshop on* (pp. 1-14). IEEE.

[20] MacCormack, A., Baldwin, C., & Rusnak, J. (2012). Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis. *Research Policy, 41*(8), 1309-1324. doi: 10.1016/j.respol.2012.04.011

[21] Siegel, S. F., & Gopalakrishnan, G. (2011). Formal Analysis of Message Passing (Invited Talk). *Verification, Model Checking, and Abstract Interpretation, 6538*, 2-18.

[22] T. J. McCabe, "A complexity measure," Software Engineering IEEE Transactions on, pp. 308-320, 1976.

Appendix A

Before Refactoring

Figure 15 presents a pre-refactoring view of both the system module (upper left) and game module (lower right) sorted and split into distinct networks. In comparison to the original diagram (Figure 10) this DSM shows a smaller number of files identified as Singleton and a relatively larger Core. The undesirable cross-module dependencies remain clearly visible.

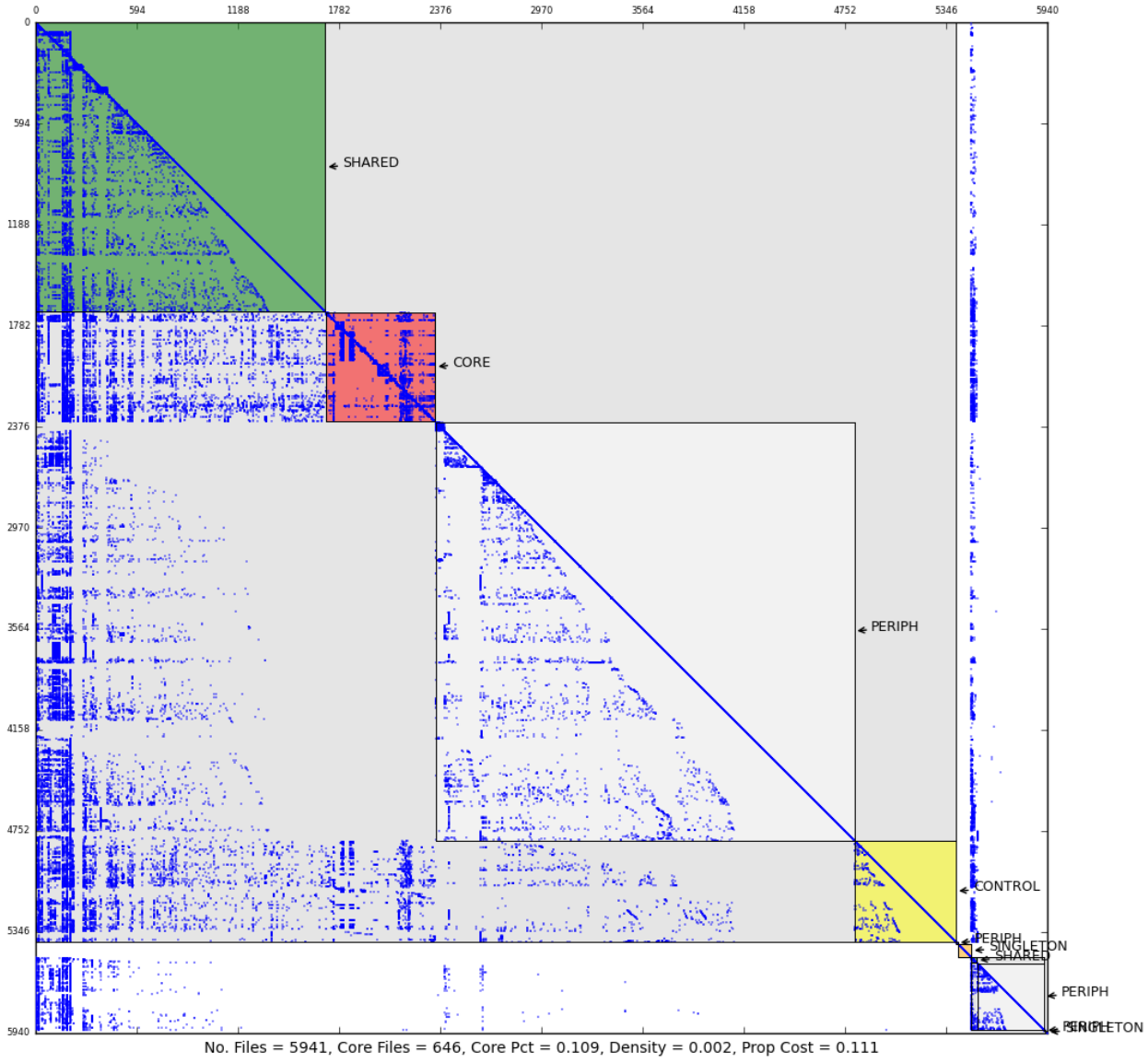


Figure 15 Extended Core-Periphery Analysis of Before-Refactoring System and Game

After Refactoring

Figure 16 presents a post-refactoring view of both the system module (upper left) and game module (lower right) sorted and split into distinct networks. In comparison to the original diagrams (Figures 11 and 13) this DSM shows a much larger Core present in the system, and a reduced Singleton count. The undesirable cross-module dependencies have now been eliminated.

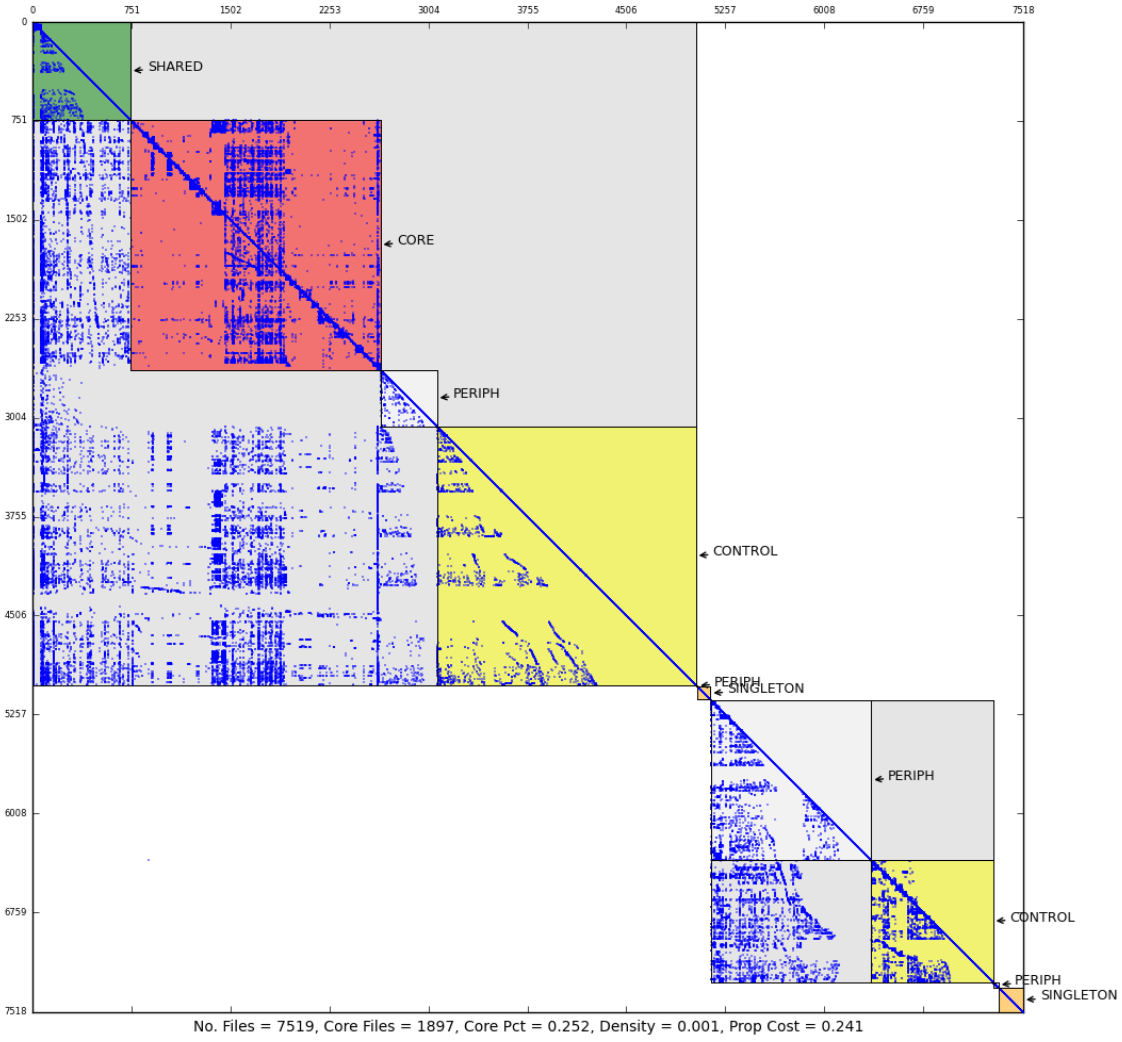


Figure 16 Extended Core-Periphery Analysis of After-Refactoring System and Game