The Impact of Software Design Structure on Product Maintenance Costs and Measurement of Economic Benefits of Product Redesign

By

Andrei Akaikine
B.S., Physics
Novosibirsk State University, 1997

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
JUNE 2010

Signature of Author: _____
System Design and Management
May 7, 2010

Certified by: _____
Alan D. MacCormack
Visiting Associate Professor, Sloan School of Management
Thesis Supervisor

Accepted by: _____
Patrick Hale
Director, System Design and Management Program

The Impact of Software Design Structure on Product Maintenance Costs and Measurement of Economic Benefits of Product Redesign

By
Andrei Akaikine

Submitted to the System Design and Management Program on May 7th, 2010 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Engineering and Management

Abstract

This paper reports results of an empirical study that aimed to demonstrate the link between software product design structure and engineers' effort to perform a code modification in the context of a corrective maintenance task. First, this paper reviews the current state of the art in engineering economics of the maintenance phase of software lifecycle. Secondly, a measure of software product complexity suitable to assess maintainability of a software system is developed. This measure is used to analyze the design structure change that happened between two versions of a mature software product. The product selected for this study underwent a significant re-design between two studied versions. Thirdly, an experiment is designed to measure the effort engineers spend designing a code modification associated with a corrective change request. These effort measurements are used to demonstrate the effect of product design complexity on engineers' productivity. It is asserted in the paper that engineer's productivity improvements have a significant economic value and can be used to justify investments into re-design of an existing software product.

Thesis Advisor: Alan D. MacCormack
Title: Visiting Associate Professor, Sloan School of Management

Table of Contents

1. Introduction

1.1 Research Motivation

On June 12, 2001, the online publication CNET News.com, published an article titled "Microsoft Exchange bug: Strike three?" which read that

"Microsoft contritely acknowledged Wednesday that its second attempt to fix an Exchange security hole went awry. Rather than fix the problem – and the security hole – the company's second attempt at a software patch included a catastrophic bug that caused many servers to hang. The company was not aware of the problem until alerted by CNET News.com." (Lemos, Microsoft Exchange bug: Strike three?, 2001)

A closer look at facts behind this article revealed the following story. It took three revisions and almost seven days to fix a code flaw that was later named "Exchange 2000 email spy bug". We would never know if the first two attempts to patch the security hole were successful at fixing this flaw. However, it is now well known that code changes that went into the first two patches had disastrous side effects and had to be reverted. Several software engineers were involved in building the patch. However, only a small amount of code was added in its final version. The amount of effort that went into building the fix was vastly out of proportion if compared to the efforts necessary to write the same amount of code in a newly developed piece of software. (Leyden, 2001) (Microsoft Corporation, 2001) (Lemos, Security hole found in Exchange 2000, 2001) (Lemos, Fix for MS Exchange causes mail problems, 2001)

This and other similar incidents demonstrate typical challenges that software vendors face when their products are in the maintenance phase of software life cycle. Yet, despite inextricable difficulties of software maintenance, most research on software products economics has been focused on costs management during the development phase of software lifecycle, ignoring costs incurred during the maintenance phase. This disparity is quite surprising since

prior research suggests that software maintenance activities represent considerable economic costs. It has been estimated that for most software products cost of maintenance activities exceeds the initial cost of development and can reach up to 90% of total life cycle cost of software development. (Seacord, Plakosh, & Lewis, 2003) Even in practice, despite its importance, software maintenance remains a highly neglected activity: less-qualified personnel is generally assigned to maintenance tasks; commonly accepted measurements of success in the maintenance phase usually revolve around cost saving and minimization of effort required for maintenance tasks; and optimizing around development costs and schedule criteria often leads to compromises in documentation, testing and structuring. These practices result in increased software maintenance costs.

Understanding drivers of software product maintenance costs should be useful to anyone who may be involved in post-release support of software products. This includes software engineers and designers who must consider trade-offs in the risk and uncertainty associated with various performance criteria of the change design activities. Specifically, they would likely be interested to understand tradeoffs between time to solution, amount of testing that may be required, and effects of code changes on maintainability of the product. This also includes system architects who may be involved in continuous evaluation of system architecture and leading redesign initiatives. Finally, managers need to be able to accurately estimate development effort required for maintenance activities.

This study focuses on software complexity as one of the main drivers of maintenance costs and represents an empirical analysis of effects of software complexity on costs associated with maintenance tasks within a large-scale commercial software product organization. Using a previously developed model for measuring the degree of design modularity (MacCormack, Rusnak, & Baldwin, 2004), this research estimates the impact of software complexity on

maintenance costs incurred by a large software development organization. Further, by applying the same analysis to different versions of the same product we aimed to measure the economic benefit of redesign efforts that have taken place between consecutive versions of the same product. In our work we assumed that a considerable amount of redesign happens between versions of the product while most existing functionality is preserved for backward-compatibility reasons.

This paper proceeds as follows. The next section, section two, reviews prior works on the topic of software maintenance and its economical significance. This section also discusses factors influencing ease of maintenance and traditional approaches to measuring these factors. In conclusion, section two asserts existence of a link between system complexity and its maintainability. Section three describes axiomatic design and design structure matrix methodologies in the context of measuring systems design complexity. It is proposed that a measure of complexity suitable for controlling the aspects of maintainability pertaining to system complexity can be designed based on these methodologies. Section four introduces the research methodology used in the study. This methodology uses design structure matrices to analyze system complexity associated with dependencies that exist between its component elements. Section five formulates hypotheses that were tested in the study. Section six and seven report empirical results and test them against hypotheses. Section eight concludes the paper.

2. Software Maintenance

2.1 Software Maintenance

Software maintenance is a broad term that refers to any changes that must be made to software products after they have been released to customers. IEEE Standard for Software Maintenance defines maintenance as "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment." (IEEE, 1998)

Another definition is offered by Seacord et al., who summarized that reasons for software change typically belong to one of the following four categories (Seacord, Plakosh, & Lewis, 2003) (IEEE, 1998):

1. Corrective. These changes are made to repair defects in the system. Defects cause software to behave inconsistently with an agreed upon specification. Defects are usually caused by design/logic mistakes or implementation errors. Corrective activity is usually associated with a documented "bug" report initiated by an end user who notices unexpected behavior of a software system.
2. Adaptive. These changes are made to keep pace with changing environments, such as new operating systems, language compliers and tools, database management systems and other commercial components.
3. Perfective. These changes are made to improve the product, such as adding new functional requirements, or to enhance performance, usability, or other system attributes. Perfective maintenance concerns functional enhancements to the product and improvements of system's operation from performance perspective or usability. Any change to specification should trigger a perfective change. Perfective changes are usually accompanied by a design change request. Such requests undergo reviews and need to gain approval before changes get implemented.

Approval for perfective change is contingent on feasibility of the improvement and on marketing/business justification for the change.

4. Preventive. These changes are made to improve future maintainability and reliability of a system. Unlike the preceding three reactive reasons for change, preventive changes proactively seek to simplify future evolution of the software product.

Some of the most recent studies of the distribution of changes between these four categories found that a majority of all changes are corrective or perfective. More than 90% of maintenance falls into one of these two categories. Depending on the type of software, corrective changes alone may represent up to 70% of all changes. (Schach, Jin, Yu, Heller, & Offutt, 2003)

2.2 Cost of Software Maintenance to Software Development Organization

Software maintenance accounts for more effort than any other software engineering activity. Multiple studies demonstrated that the cost of fixing a software defect grows in geometrical progression with the phase of software product life cycle where the defects have been discovered (Boehm, Software Engineering, 1976) (Pressman, 1982) (Rothman, 2000).

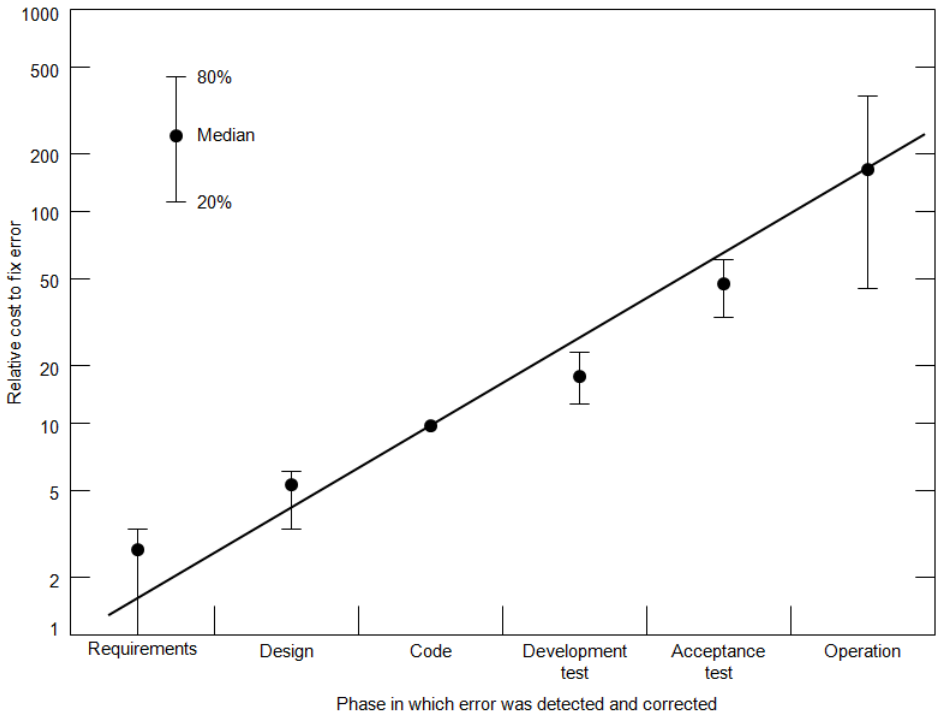


Figure 1: Increase in cost-to-fix or change software through life-cycle, based on industry data (adopted from Boehm, 1981)

As Figure 1 above demonstrates, the majority of software costs are incurred during the maintenance phase with maintenance activities consuming as much as 75-90% of the total life-cycle dollar. Traditionally, maintenance costs are attributed to the maintainers' effort, since maintenance costs are most directly a function of the professional labor component of maintenance activities. Regardless of the type of maintenance, corrective or perfective or preventive, there are three main activities that take place (Boehm, Software Engineering, 1976):

- Understanding the existing software
- Modifying the existing software
- Revalidating the modified software.

Studies of software maintainers have shown that approximately 50% of their time is spent in the process of understanding the code being modified. It is believed that a number of characteristics of existing software source code have

impact on the amount of maintenance effort required. Complexity of software was identified as one of these important characteristics that tend to have a direct effect on amount of effort required to perform a maintenance task (Banker, Datar, Kemerer, & Zweig, 1993). Complexity in the context of the mentioned studies referred to psychological complexity - a characteristic of software which makes it difficult for people to understand and work with. As defined by Curtis et al (Curtis, Sheppard, Milliman, Borst, & Love, 1979) psychological complexity assesses mental difficulty of working with source code through measuring human performance on programming tasks.

2.3 Why modify software after release

Case studies of multiple software systems performed by M.M. Lehman over a period of time that spans more than two decades resulted in the eight Laws of Software Evolution (Lehman, Ramil, Wernick, Perry, & Turski, 1997). Lehman's first, sixth and seventh laws of software evolution indicate the need for continuous process of software maintenance.

- The first law – Continuing Change: a large program that is used undergoes continuing change or becomes progressively less useful;
- The sixth law – Continuing Growth: functional content of a program must be continually increased to maintain user satisfaction over time;
- The seventh law – Declining Quality: programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

Commercial success of software vendors is the main driver behind the need to modify software products. Through the software's life cycle vendors are forced to maintain their products at "Good Enough" levels of quality. Good enough software must initially deliver high quality functions and features that end-users desire and may contain some known bugs in the implementation of more obscure or specialized functions. It is generally impractical and uneconomical to produce software, which does not need to be changed. Thus,

as end users' demand for high quality of specialized functions rises over time, software vendors need to modify their software to satisfy new demands of users in order to stay competitive. Hence, the system changes relate to changing needs of users of the system. The modification of software is not optional in maintaining software viability. From the life cycle planning aspect, this law combined with rising cost of software changes suggests that for each dollar spent on product development, a few more dollars need to be budgeted just to keep the software operational over its life cycle.

Clearly, it is beneficial if a software system is 'designed for change' during the design and implementation phases of product life cycle. Software vendors are utilizing multiple methods of developing their products that allow modifications to be applied at a low cost. This non-functional quality attribute of software that software vendors are trying to improve is called maintainability.

2.4 Maintainability

Maintainability – is the ease with which a software system or a component can be modified to correct faults, to improve performance, or other attributes, or to adapt to a changed environment.

Factors that affect maintainability of software include:

- Application age: aging software can have high support costs as it relies on old languages and requires increasingly rare expertise to maintain
- Size: number of files/modules, lines of code which need to be maintained
- Programming platform and languages
- Design methodologies, including use of design patterns
- Formatting and documentation: well written code is typically easier to read than automatically generated or ported code
- Modularization: decoupled components are easier to analyze and modify

- Documentation: maintaining documentation is expensive, thus it is often ignored. Many developers believe that “code is the best documentation”
- Management: attitudes of management toward maintenance tasks could be an additional hurdle.

It has been suggested that software design variations should be monitored throughout the development of software products for their impact on maintainability. This monitoring should cover both quantitative and qualitative evaluations along various measures, including complexity to define and assess the quality of software. ISO and IEEE specifically suggest monitoring four maintainability sub-characteristics that address analyzability, changeability, stability and testability of software because of their effect on effort (not speed) and ease of software modifications. (ISO/IEC & IEEE, 2006)

The ISO model of software quality provides the below definitions of these four characteristics of maintainability:

1. Analyzability is an important quality that is related to code readability; use of easily recognizable design patterns; choice of programming language. Factors that affect analyzability the most include coupling between modules, lack of code comments, naming of functions and variables. This characteristic is related to the efficiency with which software developer can analyze the code to understand the impact of the code change.
2. Changeability is a measure of impact of changes made to a module on the rest of the system. Design of a system is believed to play a determining role in the system’s reaction to incoming changes.
3. Stability means that most of the system’s components remain stable over time and do not need changes. Stable components require less maintenance over the life cycle of the system. Stability is achieved when core components of the system are enduring over time with changes primarily applied only to periphery components. Thus, core components

remain completely stable both internally and externally. It is important to be able to identify those components. Periphery components on the other hand can be changed at will. If the system is designed correctly, changing periphery components should not ripple through the entire system. This constitutes a link between Changeability and Stability quality characteristics of software. (Fayad, 2002)

4. Testability is related to the fact that hard-to-test programs are difficult to modify. Unit testing along with rigorous regression testing are main tasks during software maintenance and together may account for up to 25-50% of efforts of modifying software. Testability positively impacts changeability. The easier it is to run regression tests - the more insight one can get into the impact of a change on the rest of the system.

2.5 Measuring maintainability

High software maintenance costs suggest that maintainability of a software system is a very critical attribute of software quality. Software engineering economics prompt software vendors to attempt to control maintainability of a software system over its life cycle. To that end, good measures of software maintainability can help software vendors better manage effort required for the maintenance phase of software lifecycle. Despite the importance of estimation and measurement of maintainability of software there is no universal measure of maintainability. This is partially caused by the fact that there is no direct way to measure maintainability. More general software quality metrics related to maturity, effort and complexity are used as indirect measurements of maintainability.

Maturity metrics

Software maturity metrics are designed explicitly as an attempt to measure stability of a software product. These metrics tend to track stability of a software product based on changes to the product that occur over the specified period of time or between two consecutive releases. Software maturity index

(SMI) proposed by IEEE Standard 982.1-1988 (IEEE, IEEE Std. 982.1-1988 IEEE Standard Dictionary of Measures to Produce Reliable Software, 1988) is often used to measure current product stability. The SMI may be calculated using the following formula:

$$SMI = (M_T - (F_a + F_c + F_d))/M_T$$

where

M_T – is the number of software functions (modules) in the current release;

F_a – is the number of software functions (modules) in the current release that are additions to the previous release;

F_c – is the number of software functions (modules) in the current release that include internal changes from a previous release;

F_d – is the number of software functions (modules) in the previous release that are deleted in the current release.

As SMI approaches 1.0, the product begins to stabilize and may not need additional changes, which indicates improved maintainability. However, IEEE publications indicate that Software maturity index as specified above is not a good measure of maturity of a software product (IEEE, IEEE Std. 982.1-2005 IEEE Standard Dictionary of Measures of the Software Aspects of Dependability, 2005). As defined above, SMI formula measures module change rate which may not be directly linked to stability of the entire software product. Another drawback of this measure is that negative values of SMI are difficult to interpret.

Other maturity measures have been proposed to track changes in terms of lines of code per software source file. Code churn and other repository metrics track changes made to a software component over a period of time. The extent of changes made to a component of a software system can be indicative of that particular component's stability.

Effort metrics

Common software metrics are attempting to estimate effort required to perform software maintenance tasks. Effort is one of those aspects of software maintenance that seem to directly affect costs. Hence, effort-based metrics are especially popular in the industry. Most obvious measure of effort is time. In his work, Roger Pressman introduced an effort-based metric, mean-time-to-change (MTTC) (Pressman, 1982). MTTC includes the time it takes to analyze a change request, design an appropriate modification, implement the change, test it, and distribute it to all users. On average, programs that are more maintainable will have a lower MTTC for equivalent types of changes than programs that are less maintainable. Major drawbacks of MTTC include lack of predictive qualities, and dependence on maintainers' skill.

Syntactic complexity family of metrics

Syntactic complexity family of metrics attempts to derive the maintainability measure from the static analysis of software source code. These complexity measures are syntactic in nature. They frequently involve counting one or more textual properties of software. In most cases, as frequency of the selected feature increases, while everything else remains the same, so should the complexity of software. Probably the oldest and most intuitively obvious notion of complexity is the number of statements in a program. This metric is often referred to as lines of code (LOC). The primary advantage of this metric is its simplicity. Other metrics of complexity are not always as easy to compute. Syntactic complexity family of metrics also includes such metrics as McCabe's Cyclomatic Complexity (CC), Halstead Volume (HV), and their combination also known as Maintainability Index (MI).

McCabe's Cyclomatic Complexity number

Cyclomatic Complexity number, also known as McCabe's V(G), is a graph-theoretic measure of logical complexity of a software program. McCabe proposes that complexity is not closely related to program size, but rather to

the number of independent paths through the program. Since it is infeasible to enumerate the total number of unique paths in most programs, the complexity measure is defined in terms of the number of "basic paths" – paths that when taken in combination can generate all possible paths. This theory is based on direct-graph representation of program's control flow and uses graph theory to compute the number of paths. A node in the flow graph corresponds to a sequential block of code; an arc or edge corresponds to transfer of control between nodes. For any such graph G, the cyclomatic complexity number V(G) can be calculated using the following formula:

$V(G) = E - N + 2 * p$

where

E - Number of edges in the flow graph of the program;

N - Number of nodes in the flow graph of the program;

p - Number of connected components, sets of nodes with mutual connectivity - where each node can be reached from all other nodes and vice versa.

There are a number of advantages of McCabe Cyclomatic Complexity number that make it an attractive metric to be used for measuring maintainability. Obviously, there is a direct link between the number of unique paths through the program and testability sub characteristic of maintainability. Higher V(G) numbers translate in difficulty to reliably test software system that has negative effect on overall system's changeability. Additionally, it has been shown that programs with lower Cyclomatic Complexity are easier to understand and less risky to modify. The size-independent nature of Cyclomatic Complexity also makes it a good measure of relative comparison of complexity of various designs.

Halstead Volume

Halstead Program Volume is one of the set of metrics called Halstead complexity measures. Computations of all metrics in the set are based on

several primitive measures of software source code. In his work “Elements of Software Science” (Halstead, 1977), Halstead proposed a number of syntactic measures of software to express such software product measures as the overall program length, potential minimum volume for an algorithm and actual program volume of information encoded with the program code, the program level as a measurement of software complexity, and even programming effort, development time, and projected number of faults in the software.

In his theory of “software science”, Halstead shows that program volume V – the information contents of the program – can be estimated using listed above primitive measures.

$V = (N_1 + N_2) \log_2 (n_1 + n_2)$, where

N_1 – total number of operators in the program;
N_2 – total number of operands;
n_1 – number of distinct operators that appear in the program;
n_2 – number of distinct operands.

The computation of V is based on the total number of operations performed and operands handled in the program. Theoretically, a minimum volume V* must exist for a particular program. Since V* is not a purely syntactic notion it is obviously difficult to compute. Halstead uses a volume $L = V^* / V$ to demonstrate the difference of a particular implementation from the optimum. Halstead gives an approximation for the volume ratio:

$L = (2 / n_1) * (n_2 / N_2)$

Volume ratio L must always be less than 1 and represents implementation compactness of the algorithms in a program. Difficulty of a program D is the inverse of L:

$D = 1 / L = n_1 N_2 / 2 n_2$

A simple formula for Halstead Effort calculation is

$$E = D * V = n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2) / 2 n_2$$

This formula attempts to quantify the mental effort required to develop and maintain a particular program. The lower the value of this measure, the simpler it was to develop and test the program, the simpler changes to the program will be.

Halstead measures are not as widely accepted as Lines of Code metric or even McCabe Cyclomatic Complexity. The underlying theory has generated a massive controversy and has been criticized for a variety of reasons, among them the claim that there is a weak logical link between lexical complexities of code reflected in Halstead's measures and derived software metrics. However, numerous industry studies provide empirical support for using Halstead metrics in predicting effort and mean number of programming bugs. Range of metrics that can be computed using Halstead's theory and considerable simplicity of calculations make the proposed approach very attractive to practitioners of software engineering.

Maintainability Index

Maintainability Index (MI) is a composite metric based on a number of traditional metrics. Maintainability Index was originally proposed by Oman and Hagemeister (Oman & Hagemeister, 1992) to overcome drawbacks of any particular standalone metric and to combine many metrics into a single index of maintainability. MI is given as a polynomial equation comprised of weighted predictor variables. A series of polynomial regression models have been defined by the authors of the MI to determine the weights for predictor variables. Through a series of studies it was demonstrated that there is a strong correlation between such predictor variables as Halstead Volume, McCabe's Cyclomatic Complexity, lines of code, and number of comments to the

maintainability of the software system. The original polynomial equation for Maintainability Index was defined as follows:

MI = 171 – 3.42 * ln(aveE) – 0.23 * aveV(g') – 16.2 * ln(aveLOC) + 0.99 * aveCM,

where

aveE – is the average Halstead Effort per module;

aveV(g') – is the average extended cyclomatic complexity per module;

aveLOC – is the average number of lines of code per module;

aveCM – is the average number of lines of comments per module.

Based on the proposed equation, two quality cutoffs were identified to help analyze systems. Values above 85 indicate the software that is highly maintainable, values between 85 and 65 suggest moderate maintainability, and values below 65 indicate the system that is difficult to maintain. (Coleman, Assessing Maintainability, 1992)

Over time, the equation for MI has been fine-tuned by practitioners so that MI better represents system maintainability (Coleman, Ash, Lowther, & Oman, 1994). In particular, Halstead predictor variable has been modified to incorporate volume instead of effort. The comment predictor has been modified to include a comments-to-code ratio, which was identified to have a maximum additive value to the overall Maintainability Index of industrial size software systems. Modified definition for Maintainability Index is:

MI = 171 – 5.2 * ln(aveV) – 0.23 * aveV(g') – 16.2 * ln(aveLOC) + 50 * sin(sqrt(2.4 * perCM)),

where

aveV – is the average Halstead Volume per module;

perCM – is the average percent of lines of comments per module.

In the current form, Maintainability Index is fully derived from the source code of the software system. MI is very effective when used to analyze and evaluate different systems by comparing their MI values. High risk modules of

the source code can be identified with the use of MI. It gives an excellent insight into the source code of a system for direct manual analysis to highlight areas of code which require human attention.

2.6 Maintainability and Complexity

A large number of studies suggest the existence of a direct link between maintainability of a software system and its complexity (Banker, Datar, Kemerer, & Zweig, 1993) (Woodward, Hennell, & Hedley, 1979) (Curtis, Sheppard, Milliman, Borst, & Love, 1979) (Agresti, 1982) (Harrison, Magel, Kluczny, & DeKock, 1982). Many of these studies propose that software complexity is the primary driver behind software maintainability. Such metrics as Lines of Code, McCabe's Cyclomatic Complexity, and Halstead's Volume claim to measure complexity of a software system in one way or another. Many successful attempts were made to demonstrate correlation between metrics mentioned above and maintainability as measured by maintainers' effort to understand, modify and test the software. However, no single best approach to measure software complexity has emerged.

There is an ongoing debate about applicability of metrics developed prior to wide acceptance of structured programming to software systems developed using modern approaches. It was asserted that use of structured programming methodologies such as reduced branching and increased modularity has a significant impact on changeability of software (Stevens, Myers, & Constantine, 1974). Gibson and Senn (Gibson & Senn, 1989) in their experiments demonstrated that more structured versions of the same software required less time for completion of maintenance tasks. They also confirmed that such metrics as McCabe's Cyclomatic Complexity and Halstead's Effort correlate with improvements caused by using structured programming approach. However, the effects of re-structuring software on traditional complexity metrics were not linear. Hence, to reliably measure complexity of newly developed systems traditional metrics need to be recalibrated for each language

and programming approach used in each particular instance of software system development.

Critics of software complexity measures point out that currently used metrics provide only a crude index of software complexity. (Kearney, Sedlmeyer, Thompson, Gray, & Adler, 1986) The essential properties of good measures such as robustness, normativeness, specificity, and prescriptiveness are not uniformly addressed with traditional metrics. The following properties of measures should help practitioners to determine the way in which measures can be used:

- Robustness – a measure which should reliably predict complexity of software. Decrease of the measure is consistent with improved complexity of the program;
- Normativeness – a metric which should provide a norm against which programs' measurements can be compared;
- Specificity – a measure which should be able to find deficiencies of a software system that can be used as a guide to testing and maintenance;
- Prescriptiveness – a measure which should prescribe techniques and direct their application to reduce complexity.

Traditional software metrics do not always meet the needs of their users whether it is a software engineer or a system architect. Also, lack of influence of traditional metrics on programming behaviors limits metrics' managerial use.

Kearney et al. propose an approach to the creation of complexity measures: "Before a measure can be developed, a clear specification of what is being measured and why it is to be measured must be formulated. This description should be supported by a theory of programming behavior. The developer must anticipate the potential uses of the measure, which should be tested in the intended arena of usage."

What would be the best metric to use in the context of software maintenance? The following chapter attempts to make a case for a use of complexity metrics family from the field of technology management and systems design. Proposed metrics family is based on product architecture rather than syntactic measures of source code. Subsequent chapters discuss an empirical study of the proposed metrics family based on data from the industry.

3. Design Complexity Measure for Maintainability

3.1 Metrics specification

As discussed above, a good complexity measure to be used in the context of maintenance should address four sub-characteristics of maintainability: analyzability, changeability, stability and testability. Expanding on the definitions from the ISO software model:

- Analyzability is related to readability of the code and how easy it is to discern an underlying algorithm. Factors that affect analyzability include coupling between modules and discoverability of functions in the modules. This characteristic is related to the efficiency with which a software developer can analyze code to understand the impact of code changes.
- Changeability measures the impact of changes made to a module on the rest of the system. Design of a system is believed to play a determining role in the system's reaction to incoming changes.
- Stability is achieved when core components of the system are enduring over time and changes primarily apply to periphery components. Designing for stability relies on ability of a software engineer to identify those components. If the system is designed correctly, changing periphery components should not ripple through the entire system. At the same time changes to core components should be avoided.
- Testability measures how easy it is to test components in isolation (unit testing) and along with other components (regression testing). Regression testing one of the main tasks during software maintenance. Its primary purpose is to ensure that code modifications did not have a 'ripple effect' on the rest of the system.

A critical aspect of a good complexity measure for maintainability is that it should help reduce cost of maintenance tasks through reduction of effort spent

on such tasks as understanding the program, devising the modification, and accounting for the 'ripple effect'. A good measure should also focus on engineers' behaviors reinforcing good practices without hindering development processes.

Decomposing the above specifications shows that a good maintainability complexity measure has the following specific purposes:

- Demonstrate how effort of a software maintenance practitioner relates to coupling between modules and positional cohesion of functions within a module or closely related modules;
- Help identify system components and explore whether the components are in the core or on the periphery of the system;
- Bring out existing dependencies between modules and reduce potential for a 'ripple effect'.

It is apparent that the modular structure of a software product is the underlying product characteristic that one needs to focus on to measure software complexity aspects that have the greatest effect on maintainability and costs associated with maintenance tasks. This structure typically emerges from mapping product functions onto physical components – creating the product architecture (Ulrich, 1995). Hence, complexity of products can be managed through appropriate application of design principles and methods to product architecture.

There is a large body of knowledge in the field of systems design that suggests that large-scale systems are often complex. Complex systems are characterized by dependencies between their numerous components. Axiomatic Design is a methodology that systemizes complexity analysis and prescribes steps towards complexity reduction through removal of dependencies of functional requirements (Suh, The Principles of Design, 1990). Axiomatic Design methodology was developed further into Complexity Theory (Suh,

Complexity: Theory and Applications, 2005). Complexity Theory uses Axiomatic Design to analyze systems. It focuses on non-deterministic nature of complex systems and aims to reduce inherent uncertainty in achieving specified functional requirements through proper mapping of functional requirements onto physical design attributes. Approaches and methods of Axiomatic Design (AD) and Complexity Theory (CT) can be applied to the reduction of software systems complexity.

Complementary to Axiomatic Design method is the Design Structure Matrix (DSM) approach to managing dependencies by manipulating design system components into modular architecture (Baldwin & Clark, 2000) (Eppinger, Whitney, Smith, & Gebala, 1989). DSM approach provides a basis for measuring software system complexity in such a way that specifications listed above are satisfied.

3.2 Axiomatic Design and Complexity

The Axiomatic Design framework can be summarized as follows (Suh, Complexity: Theory and Applications, 2005):

- The design world consists of four domains: customer domain, functional domain, physical domain and process domain. Each domain is characterized by domain specific attributes (Figure 2)Figure 2.

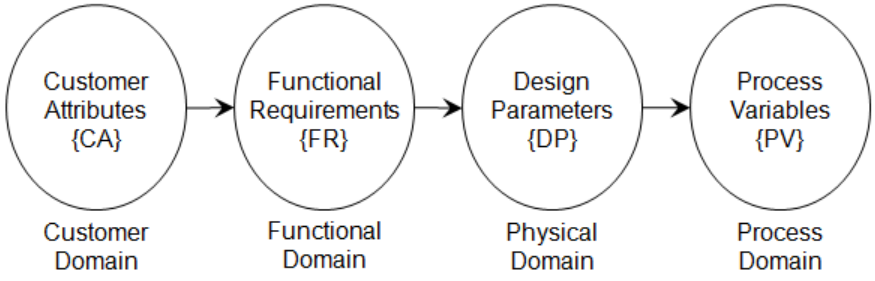


Figure 2: Four domains of the design world (adopted from Suh, 2005)

- Decomposition and zigzagging are used to construct the attributes in each domain (Figure 3). Through the design decomposition process, the designer is transforming design intent into realizable design details.

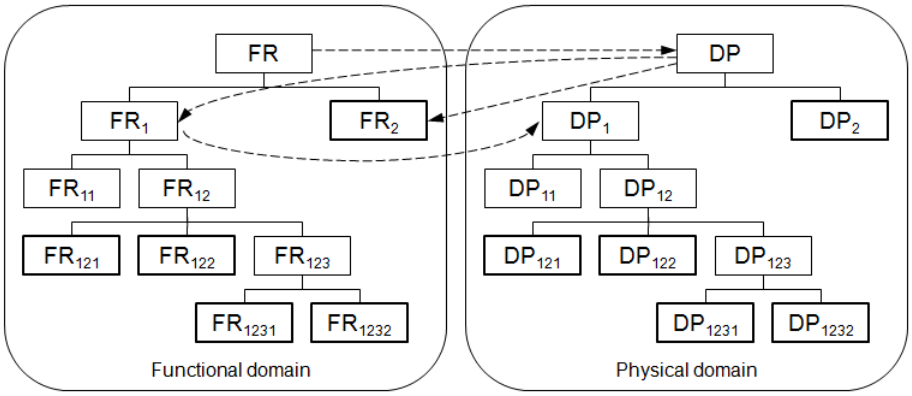


Figure 3: Zigzagging to decompose FRs and DPs (adopted from Suh, 2005)

- Mappings are translations of characteristics vectors from one domain to another. For example, once Functional Requirements (FRs) in the functional domain are chosen, designer maps them to the physical domain to conceive a design with specific Design Parameters (DPs) that can satisfy FRs. Design equations are used to represent the mappings. For example, mapping between FRs and DPs can be represented by: {FR} = [A] {DP}, where {FR} is a vector of all FRs, {DP} is a vector containing all DPs of the design, and [A] is the "design matrix" that defines the relationships between the design parameters and the functional parameters. If the number of FRs equals the number of DPs, equals number n, [A] is a square matrix of size n x n.

  For n = 3, the equation will take the following form:

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix}$$

- Two fundamental axioms were identified to govern the design process: The Independence Axiom and The Information Axiom. The Independence Axiom states that the independence of Functional Requirements must be

maintained for robustness, simplicity, and reliability of systems. The Information Axiom states that the system must be designed to minimize uncertainty in achieving the FRs defined in the functional domain.

In a simplified form, the values of the "design matrix" elements will either be 'X' or '0'. 'X' represents a mapping between the corresponding components of a vector {FR} and vector {DP}. '0' signifies no mapping between components of vectors being mapped. Examination of the structure of the "design matrix" provides for design characterization:

- Coupled design – is a design that does not maintain the independence of functional requirements. The design matrix is a full matrix.

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} X & 0 & X \\ X & X & X \\ 0 & X & X \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix}$$

- Decoupled design – is a design that maintains the independence of functional requirements if and only if the design parameters are determined in a proper order. One Functional Requirement may be satisfied by more than one Design Parameter. This kind of design maintains the independence of Functional Requirements, provided that the design parameters are specified in a sequence so that for each Functional Requirement there is one unique Design Parameter that ultimately controls that Functional Requirement. The design matrix is triangular.

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} X & 0 & 0 \\ X & X & 0 \\ X & X & X \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix}$$

- Uncoupled design – is a design that has all of its functional requirements independent from other functional requirements. There is a one-to-one mapping between functional and physical domain attributes. The design matrix is diagonal.

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} X & 0 & 0 \\ 0 & X & 0 \\ 0 & 0 & X \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \end{Bmatrix}$$

- Ideal design – is a design that has the same number of Functional Requirements and Design Parameters and satisfies Independence Axiom with zero information content.
- Redundant design – is a design that has more design parameters than the number of functional requirements. The design matrix is not square.

$$\begin{Bmatrix} FR_1 \\ FR_2 \\ FR_3 \end{Bmatrix} = \begin{bmatrix} X & 0 & X & X & X \\ 0 & X & X & 0 & X \\ 0 & X & 0 & X & X \end{bmatrix} \begin{Bmatrix} DP_1 \\ DP_2 \\ DP_3 \\ DP_4 \\ DP_5 \end{Bmatrix}$$

It has been demonstrated that analysis techniques of Axiomatic Design are effective when applied to reduction of complexity of both mechanical and software systems (Suh, The Principles of Design, 1990) (Suh, Axiomatic Design: Advances and Applications, 2001). Use of Axiomatic Design facilitates characterization of software system architecture in the functional domain and Axiomatic Design decomposition approach is a good foundation for redesign efforts. However, since Axiomatic Design is dependent on the existence of system specifications its methods may not be readily applicable to the maintenance phase of software lifecycle.

Design Structure Matrix is a promising tool for measuring structural systems complexity that may be able to address the requirements of the maintenance phase of software lifecycle in dealing with software product complexity.

3.3 Design Structure Matrix

Hierarchical relationships and interdependencies among design parameters can be formally mapped using a tool called the Design Structure Matrix (Baldwin & Clark, 2000). The DSM characterizes the “topography” of a design

domain by displaying hierarchical relationships and interdependencies among design elements in a matrix form. To construct a DSM, one assigns all individual design elements to the rows and columns of a square matrix. A dependency link between two elements is indicated by a mark in the corresponding element of the matrix. For example if a design element B is an input to a design element A, this relationship may be depicted by a mark (an 'X') in the column of B and the row of A. It is said that the element A depends on the element B, or in other words modification to the element B may have an effect on the element A. (Figure 4: A Design Structure Matrix with 6 design elements). Hence, the resulting matrix captures both hierarchical dependencies among design elements (an element B calls into elements C and D) and their interdependencies (a change in D makes a change in B, C and E desirable).

	A	B	C	D	E	F
A		X				
B			X	X		
C	X			X		
D						X
E				X		
F						

Figure 4: A Design Structure Matrix with 6 design elements

DSMs are a powerful tool in assessing modularity of products. By reordering rows and columns of a DSM designers can achieve clustering of components so that underlying modular structure of a product becomes apparent. Clustering of design elements in the DSM is also known as partitioning. In his work "The Design Structure System", D.V. Steward discussed partitioning methods (Steward, 1981). The goal of DSM partitioning is to achieve a "block triangular" matrix structure by applying same re-ordering transformation to the rows and the columns of the DSM. "Block diagonal" matrix structure emerges when dependency marks are confined to appear either below the main diagonal or

within square blocks on the diagonal. For example the following two DSMs represent identical components dependency (Figure 5: Clustering of design elements into modules). After reordering of columns and rows a potential modular structure emerges. By definition developed by Baldwin and Clark (Baldwin & Clark, 2000), modules are units of a larger system whose structural elements are powerfully connected among themselves and weakly connected to elements in other units. In the presented DSM, modules are marked by two bold-lined boxes. Modules contain strong dependencies within themselves. Weak interdependence between modules is indicated by the fact that there are only a few 'X' marks outside the bounded boxes.

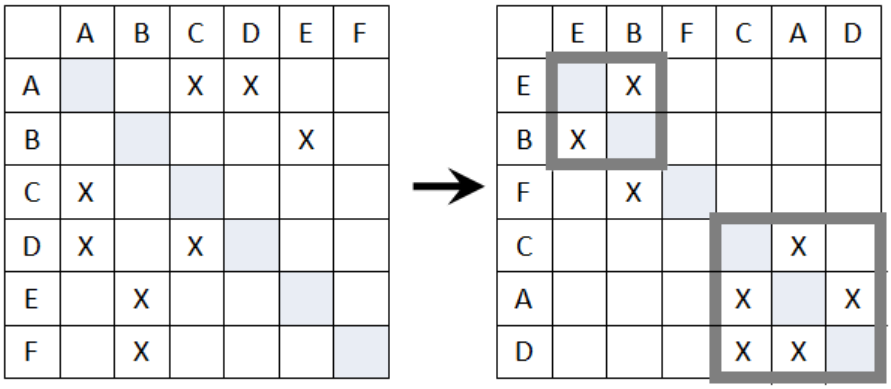


Figure 5: Clustering of design elements into modules

Understanding system design modularity helps reduce complexity of modification and system maintenance tasks. Ideas of abstraction, information hiding and interface when applied in the context of modular systems reduce complexity. This is achieved through the process of hiding information into separate abstractions that have simple interfaces. Abstraction hides the complexity of design elements and their interactions within a module, while the interface defines the interaction of the module with other components in the system.

3.4 DSM in application to analysis of system complexity

DSM methodology provides a foundation for designing a software system complexity measure suitable for use during the maintenance phase of the software lifecycle. It was demonstrated in many studies that DSM methodology can be used in the assessment of system modularity (Baldwin & Clark, 2000) (Eppinger, Whitney, Smith, & Gebala, 1989). Understanding of the modular structure of a system provides basis for analysis of cohesion of elements within each module and measure of coupling between modules. MacCormack et al. (MacCormack, Rusnak, & Baldwin, WP# 08-038, 2008) (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010) demonstrated a method of discerning core-periphery structure of a software system using DSMs. Using the suggested approach one can measure the relative stability of different modules of the system and predict the 'ripple effect' of code modifications. Thus, DSMs provide a robust and repeatable way to analyze and measure the characteristics of a design complexity of a system as a whole and at the component level.

4. Research Methods

This study applies DSM methodology to analyze the structural complexity of a software system. Chosen approach draws extensively on methodology developed by MacCormack, Baldwing and Rusnack (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010) (MacCormack, Rusnak, & Baldwin, Exploring the Structure of Complex Software Designs: An Emperical Study of Open Source and Proprietary Code, 2004) (MacCormack, Rusnak, & Baldwin, WP# 08-038, 2008). Subsections below provide a description of the approach used to measure important complexity attributes of the software system architecture and how these measures relate to the maintainability measure developed in this study.

4.1 Applying DSM to software

Traditionally, DSM is used to discern the modular organization of any system. In contrast to traditional usage, this study employed DSMs in calculation of metrics that measure a degree of structural complexity of a system. In software development, modules are usually easy to identify. Software design engineers tend to group source files of related nature into directories (Figure 6: Source code directory structure). This results in clustering of files into groups that are defined by a project directory structure. This clustering closely resembles product's modularization and represents engineer's view of the system. This overt architecture of the product often dictates managerial decisions in regards to feasibility evaluation of a proposed maintenance task, assignment of engineers to tasks, and estimation of effort that may be required.

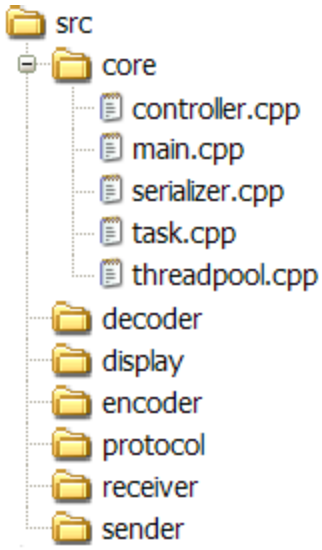


Figure 6: Source code directory structure

In building DSMs, this study uses a source file as a unit of analysis. Prior research in the field suggests that such level of analysis is suitable for a number of reasons (MacCormack, Rusnak, & Baldwin, WP# 08-038, 2008):

1. Source files tend to contain functions and data structures associated with the same functional requirement;
2. A source file, as a unit of analysis, reflects how programmers perceive and shape the product structure;
3. Traditionally, software development tools operate with source files as units to organize the software code;
4. Other empirical studies that focus on analysis of software structure typically use source file as the unit of analysis.

Thus, DSMs that were built and analyzed have rows and columns corresponding to source files. The order of files was preserved as it was found in the project's directory structure.

"Function Call" was used to identify dependencies between source files. "Function Call" is one of several important types and one of the most universal types of dependencies between source files in a software system. It is applicable

when working with almost all modern programming languages and technologies. A “Function Call” is an instruction that uses a function name to request a specific task to be executed. Called function may or may not be located within the source file originating the request. When the function is not located in the same file, this creates a directional dependency between two source files. For example, if FunctionA() in SourceFile1 calls FunctionB() in SourceFile2, then we note that SourceFile1 depends upon SourceFile2. This dependency can be indicated in the DSM by placing a mark in the matrix element located at (row: SourveFile1, column: SourceFile2). This dependency does not imply that SourceFile2 depends upon SourceFile1; the dependency is not symmetric unless SourceFile2 also calls a function defined in SourceFile1.

To record the dependencies between files a binary matrix was used such that dependency between two files is indicated by ‘1’ situated in a corresponding element of the matrix. In absence of the corresponding dependency, elements of the matrix hold the ‘0’ value. In this approach, only the presence of dependencies between two files was recorded, not nature of this dependency or strength of coupling between two files.

To capture function calls, a product called Understand 2.5 was employed. It is distributed by Scientific Toolworks, Inc. (www.scitools.com). This product is capable of extracting function calls and other types of dependencies from the source code tree provided to the tool as an input. It uses static code analysis to extract dependencies. Use of a static call extractor is justified as the resultant data represents the structure of the product from the programmer’s perspective. Data captured by the Understand is output in a format that can easily be converted into a DSM.

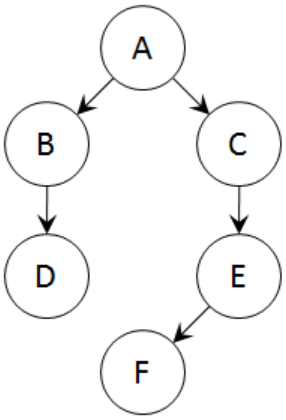
4.2 Visibility Matrix

First metric of software system complexity measures the degree of ‘ripple effect’ propagation through the system directly – through an existing

dependency, or indirectly – through a chain of dependencies that exist across elements. *Propagation cost* predicts the percentage of system elements that can be affected, on average, when a change is made to a randomly selected design element (MacCormack, Rusnak, & Baldwin, Exploring the Structure of Complex Software Designs: An Emperical Study of Open Source and Proprietary Code, 2004). Measured in percentage points this metric is independent of size of the project, which lends this metric to be useful for projects of different sizes.

In computing propagation cost, first the "visibility" (Sharman & Yassine, 2004) of design elements is identified. To compute visibility of any given element for any given path, a reachability matrix is built using a technique that employs matrix multiplication and summation (Warfield, 1973). A simple example below illustrates the chosen approach.

Consider the system with the following element relationships, given as a dependency graph and in a DSM form:



	A	B	C	D	E	F
A		1	1			
B				1		
C					1	
D						
E						1
F						

Figure 7: Example system dependency graph and DSM

Design element A depends upon elements B and C, so a change to element B may have a direct impact on element A. Element B depends upon element D, so a change to element D may have a direct impact on element B and an indirect impact on element A. The path through which change impact – ripple

effect – propagates from the element D to the element A has a length of two. Similarly, change to the element F may have indirect impact on the element A with a propagation path length of three. Note that there are no indirect dependencies between elements for path lengths of four or more.

To build a visibility matrix in addition to direct dependencies it is necessary to identify all indirect dependencies between elements. By raising a binary DSM to successive powers of n, one can find all indirect dependencies that exist for the dependency propagation path lengths of n. The visibility matrix V is derived by summing all resulting matrices together and with diagonal matrix (to demonstrate that design elements of the system depend upon themselves). Computed this way visibility matrix shows the dependencies that exist between all system design elements.

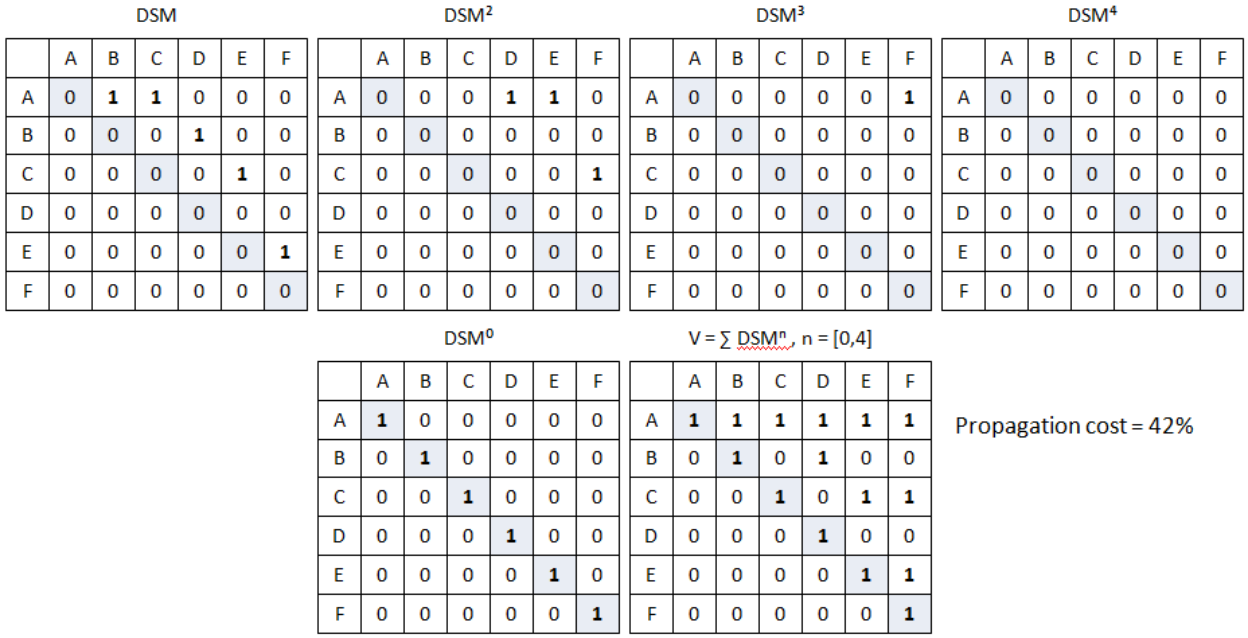


Figure 8: Computation of the visibility matrix

4.3 Design element visibility metrics

Design elements visibility measures can be derived from the visibility matrix. A measure of dependencies that flow into an element – Fan-In Visibility (FIV) – can be computed by summing down the column of visibility matrix

which corresponds to the element, and dividing by the total number of elements. An element with high Fan-In Visibility has many other elements that depend on it. Fan-Out Visibility (FOV) – the measure of dependencies that flow out from the element – can be obtained by summing along the row of the visibility matrix which corresponds to the element, and dividing by the total number of elements. An element with high Fan-Out Visibility depends upon many other elements. In our example, element A has a Fan-Out Visibility of 100% meaning that it depends upon all elements in the system. The same element has Fan-In Visibility of 17% (1/6th) meaning that it is visible only to itself.[1]

	A	B	C	D	E	F	FOV
A	**1**	**1**	**1**	**1**	**1**	**1**	1.0
B		**1**		**1**			.33
C			**1**		**1**	**1**	.5
D				**1**			.17
E					**1**	**1**	.33
F						**1**	.17
FIV	.17	.33	.33	.5	.5	.67	Avg: .42

Figure 9: Computation of Fan-In Visibility and Fan-Out Visibility

An average of Fan-In Visibility values across all elements of the system provides a system wide measure of visibility. This metric is referred to as "Propagation Cost". Please note that due to the symmetry between aggregate of Fan-In Visibility and Fan-Out Visibility measures – every dependency contributes to both aggregate Fan-In and aggregate Fan-Out – propagation cost can also be computed as an average of Fan-Out Visibility values across all elements of the system.

[1] This and following paragraphs draw extensively from MacCormack, Rusnack and Baldwin (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010)

4.4 Core and Peripheral Components

In modular system architecture not all modules are created equal. Some components are more favorable to modifications than others. Modifiability of a component depends on the nature of its interactions with other components. As noted before, stability of the system is achieved when core components of the system are enduring over time and changes primarily apply to periphery components. If the system is designed correctly, changing periphery components should not ripple through the entire system. Stability of core modules should improve system maintainability.

As defined in literature (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010), core components are those that are tightly coupled to other components in the system. Peripheral components are characterized by weak coupling to other components. As noted in the previous section coupling between components can be characterized by the direction of dependency propagation. Due to this duality it is appropriate to define two more component types: shared components and control components (Figure 10: Characterization of components by visibility measures).

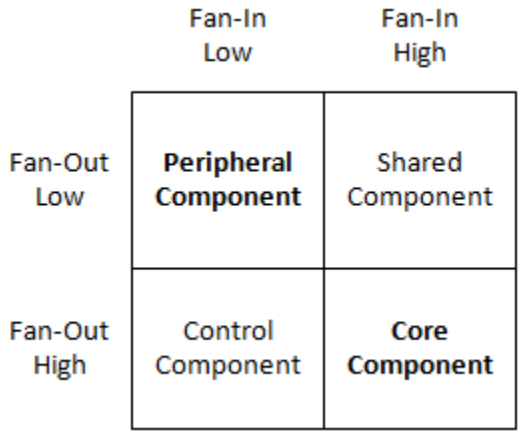


Figure 10: Characterization of components by visibility measures

- Core Components (High FIV, High FOV) – are components with high visibility on both measures. "Core" modules are "seen by" many modules

and "see" many other modules as they are implementing core system functionality.

- Periphery Components (Low FIV, Low FOV) – are components with low visibility in both directions. They typically implement auxiliary functions.
- Shared Components (High FIV, Low FOV) – are components that depend on few components while many other components depend on them. They usually provide shared functionality to many different parts of the system. Shared libraries of basic functions are a good example of shared components.
- Control Components (Low FIV, High FOV) – are components that usually are responsible for directing the flow of program execution. They depend on many different parts of the system, while only a few other components demonstrate dependency on control components.[2]

To evaluate whether a component meets a high or low criteria for visibility measures this study uses an approach proposed by MacCormack at al. (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010). Components that exceed 50% of the maximum level of a visibility measure across all components are valued as "High" while components that don't meet this threshold are valued as "Low" for the corresponding visibility measure.

Grouping components in these four types can be used in several ways. Obviously maintainability of components differs depending on their types. Potential system stability can be evaluated by measuring relative size of the "core". The smaller the core, the more stable a system is likely to be.

Studies of many software systems demonstrated that modules (source files) of the core type are not located in only a small number of distinct directories (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010). Core modules are distributed throughout the system. For many systems it is not always clear

[2] These definitions of four canonical types of components were first introduced by MacCormack, Rusnack and Baldwin (MacCormack, Rusnak, & Baldwin, WP# 08-038, 2008)

from the directory structure which components are core and which are peripheral. This finding denotes the challenge facing a system maintainer in evaluating maintainability of individual components. Core components are difficult to identify due to effort that may be required to trace all indirect dependencies contributing to high FIV and high FOV measures. Metrics evaluated in this study can assist system maintainers in evaluating how system complexity affects maintenance tasks.

5. Hypotheses

This study utilizes the outlined methodology to explore the link between complexity of software products and maintenance costs. The study focuses specifically on the effort software developers spend implementing corrective modifications to software. The following three hypotheses were tested in the course of the study.

Hypothesis 1: The amount of effort to implement a corrective code change is positively related to the overall system complexity as measured by the level of interconnectedness of source files comprising the system.

Hypothesis 2: The likelihood of next modification going in a particular source file is higher for core components of the system. Core components have higher potential for causing additional rework cycles and as a consequence higher maintenance costs.

Hypothesis 3: Product redesign has quantifiable economic benefits when applied to source files and components that are measured to be contributing to the overall complexity of the product.

6. Data

6.1 Description of the data

For this study, two consecutive versions of a mature software system were analyzed. In this case maturity of the software system is defined by a wide market acceptance of the product, and stability of its features. For ease of description first version of the product will be referred to as “old product”, while the newer version will be referred to as “new product”. As it will be demonstrated later the architecture of the product changed significantly, so it is quite reasonable to treat the two versions of the studied system as two different products.

A major redesign effort was undertaken in-between studied versions of the product. Most of the source code was rewritten using new languages. In the old product the majority of source files (17570 out of 18512) were written in C/C++ languages. In the new product, the share of C/C++ files is much lower: 4949 out of 13139. The majority of files in the new product were written in C# programming language (8149 out of 13139, a large increase from 605 in the old product). The redesign effort was justified by the need to improve some system quality attributes. Most of the system’s functional requirements were transferred from one version of the product to another without much change. Two specific system quality attributes that the development team focused on during the redesign effort were scalability and performance. Increase in both targeted system quality attributes was achieved through increased modularity and better mapping of functional requirements onto components of the system. With new system architecture, it became possible to increase system throughput by sharing load between several similar components performing the same function. Better defined interfaces between components allowed the system to be distributed between many computer systems. More targeted mapping of functional requirements onto components permitted removal of

unused components from the system at the time of system deployment into production.

Three types of data were collected for this study. For each version of the product we collected:

- Snapshots of the whole source code tree at the time each version of the product was released to customers;
- Source code change logs for the period of approximately 30 months after each product release;
- Data from the bug tracking system that corresponds to source code changes.

The snapshots of the source code tree were analyzed using Understand 2.5 (www.scitools.com) to derive dependency structure between source code files. MATLAB (MathWorks, 2010) was used for matrix manipulation and graphing of design structure matrices (DSMs).

Source code change logs were analyzed using custom scripts. Information about files that were modified with each submitted change and related bug identifiers were extracted from the change logs.

Based on the bug data, corrective changes were identified. Records pertaining to corrective work items were analyzed to measure the effort a software design engineer spent to devise the corresponding code changes.

Resulting dataset contains about 400 corrective changes for each version of the product. This number is not a total number of known defects, but rather the number of defects that have been fixed in the studied period of time. The number of corrective code changes submitted to the code base depends primarily on the development organization throughput, and by no means should be used as a measure of overall product quality. Increase in productivity associated with the product redesign may lead to a larger number

of issues getting fixed, however this does not imply that the quality of the product code decreased with a redesign. The measurement of number of defects and overall product quality is outside of scope for this study.

Focus on corrective maintenance tasks

This study specifically focused on corrective code changes made to the software system in its maintenance phase of the life-cycle. The choice to analyze only corrective modifications was made for a number of reasons, primarily related to the high quality of data related to corrective modifications. In the development organization used for the study, corrective changes are always associated with a bug reported by a customer. Customers' change requests are handled with increased urgency. Because of this urgency associated with corrective change requests, solutions are delivered as soon as possible. This results in the absence of idle time when issues are waiting for developer resources to become available. Importance of corrective change requests exhorts the development organization to apply more control to the process of tracking open issues, hence each transaction with source code related to the issue gets documented. This allows for more precise measurement of time software engineers spend working with code.

Measuring Resolution Time

In the targeted organization software engineers record in the bug tracking system the time when they start working on a code modification and the time when the code modification is done. This provides two data points that can be used in assessing the time spent by an engineer working with code. This measure accounts only for one component of mean-time-to-change metric (MTTC)(Pressman, 1982). MTTC includes the time it takes to analyze a change request, design an appropriate modification, implement the change, test it, and distribute it to all users. This study only focuses on the time engineers spend designing an appropriate code modification. This precludes any influence from ingredients of MTTC that may not be affected by complexity of the product

structure, the target of this study. Other components of MTTC, such as time to investigate of a problem, testing time, and time to distribute the modification to users may not have as strong of a relationship with product complexity.

In this study, a simplified approach is used to calculate the time engineers spend designing code modifications. Eight-hour work days are assumed. Weekends and nights are excluded from the computation of developer's effort. If the code modification undergoes multiple rework cycles, time is computed separately for each rework cycle that has been reported in the bug tracking database. Total time for a particular code modification is a sum of all individual rework cycle time measures.

The following formula was used for resolution time computation:

$RT = (T_{END} - T_{BEG}) - 16*Days(T_{END} - T_{BEG}) - 16*Weekends(T_{END} - T_{BEG})$, where

T_{BEG} is the time when the engineer reported to start designing the code modification,

T_{END} is the time when the engineer reported to finish working on the code modification.

Most studied code modifications took a software engineer more than one day to develop. The formula above is justified by the fact that only a few issues out of the studied set were worked on continually day and night. In most cases, engineers took time off from working on the fix. Resolution time computed this way does not account for variability of engineer's effort over time. This study addresses uneven distribution of engineer's effort with an adjusted effort measurement discussed in the next section.

Accounting for Variability of Effort

Based on the experience with the targeted organization and the insight into the work practices of software engineers the following scheme for accounting

for variability of engineer's effort was developed for this study. The following scheme attempts to capture the following dynamic.

Urgent nature of corrective code changes that this study focuses on requires that software engineers spend effort to get to a solution as soon as possible; thus, most changes should be coded within 1-2 weeks due to time pressure. This goal is supported by the change request acceptance process employed in the targeted organization. Change request acceptance process prevents code and design modifications of larger scope from being treated the same way as corrective changes. During the code change request assessment, incoming code change requests are evaluated and classified based on the effort and urgency that may be required to implement the change. Changes that are not as urgent, require a significant design and test efforts, or may require more than a month of work are designated to the Design Change Requests (DCRs) category. Change requests of an urgent nature are designated to the Corrective Changes category. The focus of this study was exclusively on Corrective Changes. Design change requests were excluded from this study.

It is assumed, that if an engineer reports more than 10 work days for designing a code modification, it is due to some extraneous circumstance, not because of difficulties of working with the source code. For an example of such extraneous circumstance consider the situation when an engineer needs to consult a subject matter expert on a particular product feature. Operational aspects of such consultation should not be counted as developer's effort coding the solution. It is assumed that in the first 5 days of working on the issue developers dedicate at least 75% of their working time to that issue. This translates into 6 hours of work out of 8 work day hours. After 5 days of working on the issue almost exclusively, focus of the developer may be shifted to other issues that may seem more urgent. In the next 5 days of working on a fix developer may be spending 25-40% of his/her time in the office working on an issue from the last week. If an engineer reports more than 10 days working

on a particular code change, it is assumed that the engineer was working on something else, so we should not count any time for that change that goes beyond 10 working days. The following graph (Figure 11: Cumulative effort spent as a function of number of days reported by an engineer) depicts the accumulation of effort spent as a function of reported number of days.

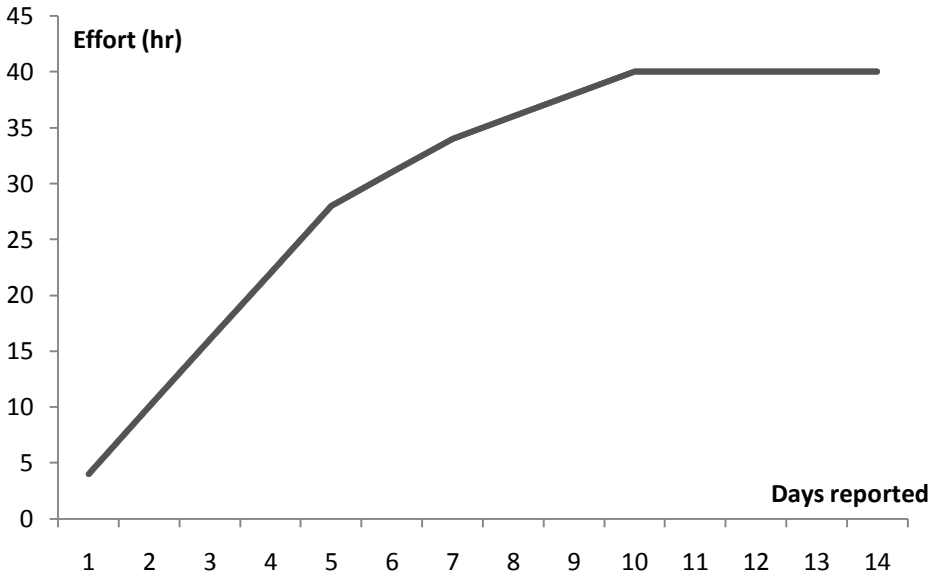


Figure 11: Cumulative effort spent as a function of number of days reported by an engineer

The following scheme was used to account for productivity variability of an individual software engineer working on a corrective change for a period of multiple days:

If the engineer reported to begin and finish working on the fix on the same day, the effort is computed using this formula:

$E = (T_{END} - T_{BEG})$, where

T_{BEG} is the time when the engineer reported to start working on the code modification,

T_{END} is the time when the engineer reported to finish working on the code modification.

If the software engineer reports T_{BEG} and T_{END} on two consecutive work days, the effort is computed using this formula:

$E = (T_{END} - T_{BOD}) + (T_{EOD} - T_{BEG})$, where

T_{EOD} is the time of the end of a work day. T_{EOD} = 20:00 in this study

T_{BOD} is the time of the beginning of a work day. T_{BOD} = 08:00

If the software engineer reports T_{BEG} and T_{END}, so that there are n working days between the first and last days of work on the fix an additional element is used to measure the effort spent by the engineer on those days. Effort will be computed using this formula:

$E = (T_{END} - T_{BOD}) + (T_{EOD} - T_{BEG}) + E_{DAYS}(n)$, where

$E_{DAYS}(n)$ is the effort an engineer spends on a long-running issue. In this study the following table function was used:

Days reported, n	**$E_{DAYS}(n)$**	**Effort on day n**
1	6	6
2	12	6
3	18	6
4	21	3
5	24	3
6	26	2
7	28	2
8	30	2
9	30	0
…	30	0

Table 1: Effort spent on an additional day

Computed this way, any code change should not take more than 46 hours of engineer's effort. However, because of rework cycles total effort for some code modifications surpassed this threshold. Effort spent by engineers in rework cycles was included in the mean engineer's effort calculations.

7. Results

7.1 Comparison of Design Structures of Products

It is always a challenge to characterize the architecture of a software product. Intuitively, engineers characterize architecture of products through descriptions of patterns of interactions between parts of the product. The challenge of a holistic characterization of a product design structure comes from the fact that patterns of interactions between modules take many different forms and shapes, from independent or sequential to bus architecture or a full mesh. Most software products have a design structure that can only be characterized as a hybrid including elements of modularity, independent structures and bus design structure.

Design Structure Matrices aid analysis of architectures of products. In this study DSMs for both versions of the product under review were built to facilitate comparison of their design structures (Figure 12: DSMs of the product before and after redesign).

Table 2 shows quantitative data comparing the product before and after the re-design.

	Old Product	New Product
Source Files	18,612	13,139
Entities (macros, types, variables, functions, files, ...)	1,793,627	1,108,423
Dependencies	182,235	112,634
Density of DSM	0.053%	0.065%

Table 2: Products comparison

Looking at the DSMs presented in Figure 12 one can notice evidence of significant structural changes between the two studied versions of the product. The following differences between representations of design structures of studied products are now observable:

- DSM for the new product is smaller due to the smaller number of files comprising the system.
- Perceived prevalence of vertical lines in the DSM for the old product denotes the existence of modules that are called into from many parts of the system. These “widely used” interfaces are evenly distributed across the components of the product indicating a mesh like architecture of the system.
- Perceived prevalence of white space away from the main diagonal in the DSM for the new product combined with the increased measure of DSM density indicates a greater cohesion of modules (source files) within components.
- Few components in the DSM for the new product have close to diagonal matrix representation pointing to sequential call relationship between source files in those components.

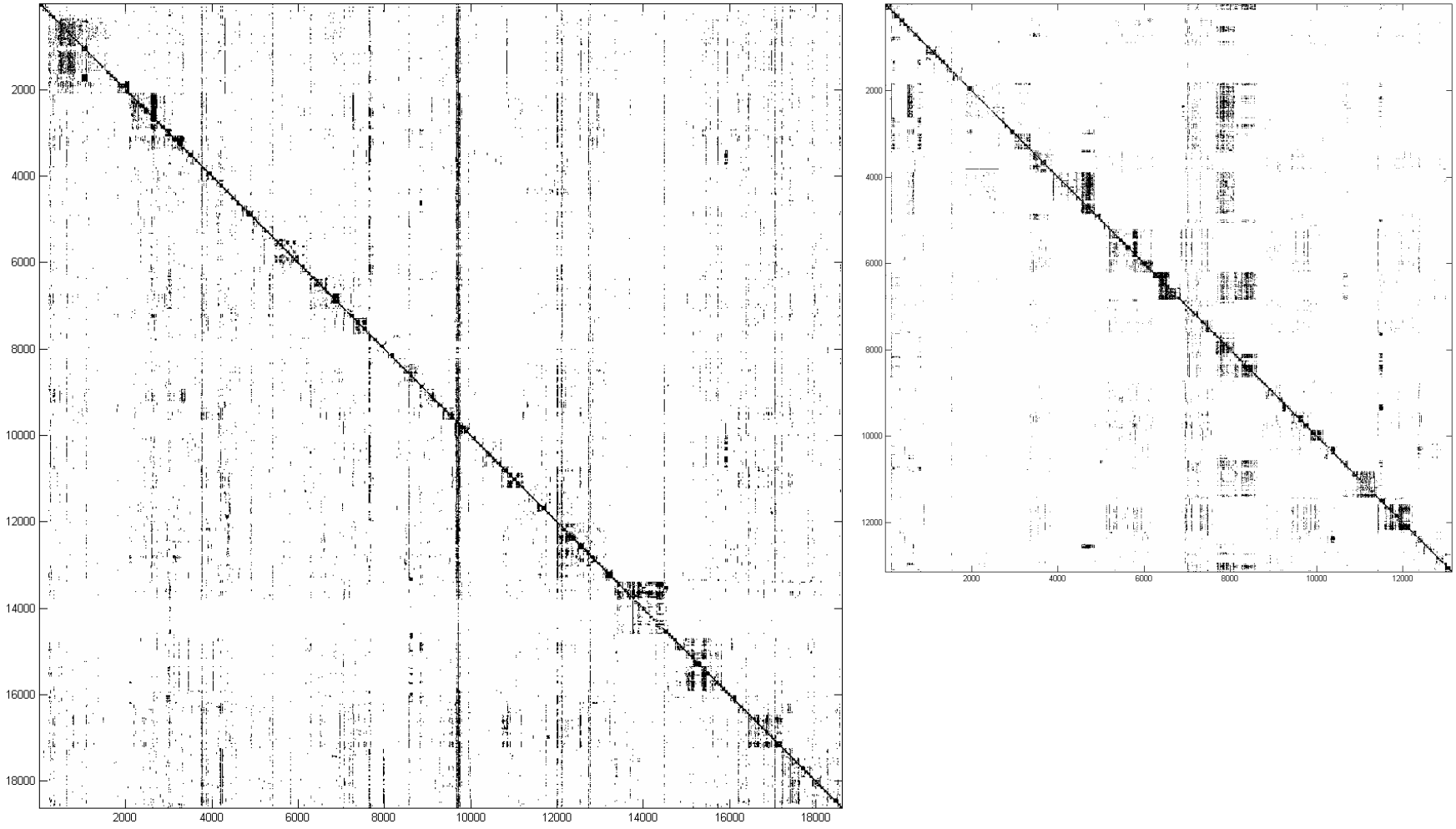


Figure 12: DSMs of the product before and after redesign

7.2 Comparison of Visibility Matrices

Complexity of the product manifested through indirect dependencies between source files can be studied using visibility matrices. In both cases, to arrive at the matrices reviewed below, DSMs were multiplied to the power 6. Further multiplication did not introduce any new dependencies. This means that the files that appear on the visibility matrix as dependent on each other have at most six degrees of separation. This means that the chain of dependencies connecting these two files has no more than 5 intermediaries.

Table 3 continues the quantitative data comparison of the product before and after the re-design. Qualitatively, it is obvious that product structure was significantly improved with the redesign. Smaller share of files demonstrating "core" and "control" qualities is a sign of good system architecture. Increased portion of "shared" modules is indicative of a modular design and higher code reuse.

	Old Product	New Product
Source Files	18,612	13,139
Entities (macros, types, variables, functions, files, …)	1,793,627	1,108,423
Dependencies	182,235	112,634
Density of DSM	0.053%	0.065%
Direct and Indirect Dependencies	131,045,503	18,971,910
Propagation Cost	38%	11%
Source files classified by type (fraction of total)		
"Core" - Visibility: High Fan-In, High Fan-Out	7245 (39%)	0 (0%)
"Periphery" - Visibility: Low Fan-In, Low Fan-Out	1889 (10%)	8445 (64%)
"Shared" - Visibility: High Fan-In, Low Fan-Out	1180 (6%)	3328 (25%)
"Control" - Visibility: Low Fan-In, High Fan-Out	8298 (45%)	1366 (10%)

Table 3: Quantitative comparison of products complexity

Figure 13 shows the difference between visibility matrices for the studied products. Visibility matrix for the old product is obviously much denser. This is

an indication of the high propagation cost. Propagation cost measured numerically for the old product is at 38%. This means that a change to a source file in the old product has a potential to ripple through average of 38% other files in the system. In other words, when making a code change an engineer should consider how this change may affect other 7,072 source files in the system (7,072 = 18,612 * 38%). In contrast, new product's propagation cost is only 11%. From engineer's perspective, this means that on average there are 1,445 source files that a random code change can ripple through. Obviously, the new product is 4 to 7 times more stable than the old product.

Visibility matrices provide a managerial tool for estimation of the risk of ripple effect depending on which source files are modified. By finding the visibility matrix column corresponding to a source file that is being modified, decision makers can assess the risk of the change causing the ripple effect. More importantly source files that may be affected can be identified by walking down the column and listing all the source files that directly and indirectly depend on the modified code.

A very important difference between the two studied products is that they have very different structures from the "core-periphery" perspective (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010). In the old product, files exhibiting "core" (39%) characteristic prevail over "periphery" (10%) files. "Control" (45%) files represent the largest share of files in the old product, while "Shared" (6%) source files represent the smallest share. In contrast, the majority of files in the new product are represented by "periphery" (64%) and "shared" (25%) modules. More important, new product does not have any "core" files, files with High Fan-In and High Fan Out visibility. These changes in the design structure may be attributed to the scalability requirement that provoked the re-design effort between the studied versions of the product. It can be hypothesized that the "periphery" structure of software products is more suitable to satisfy high scalability demands for large-scale software systems.

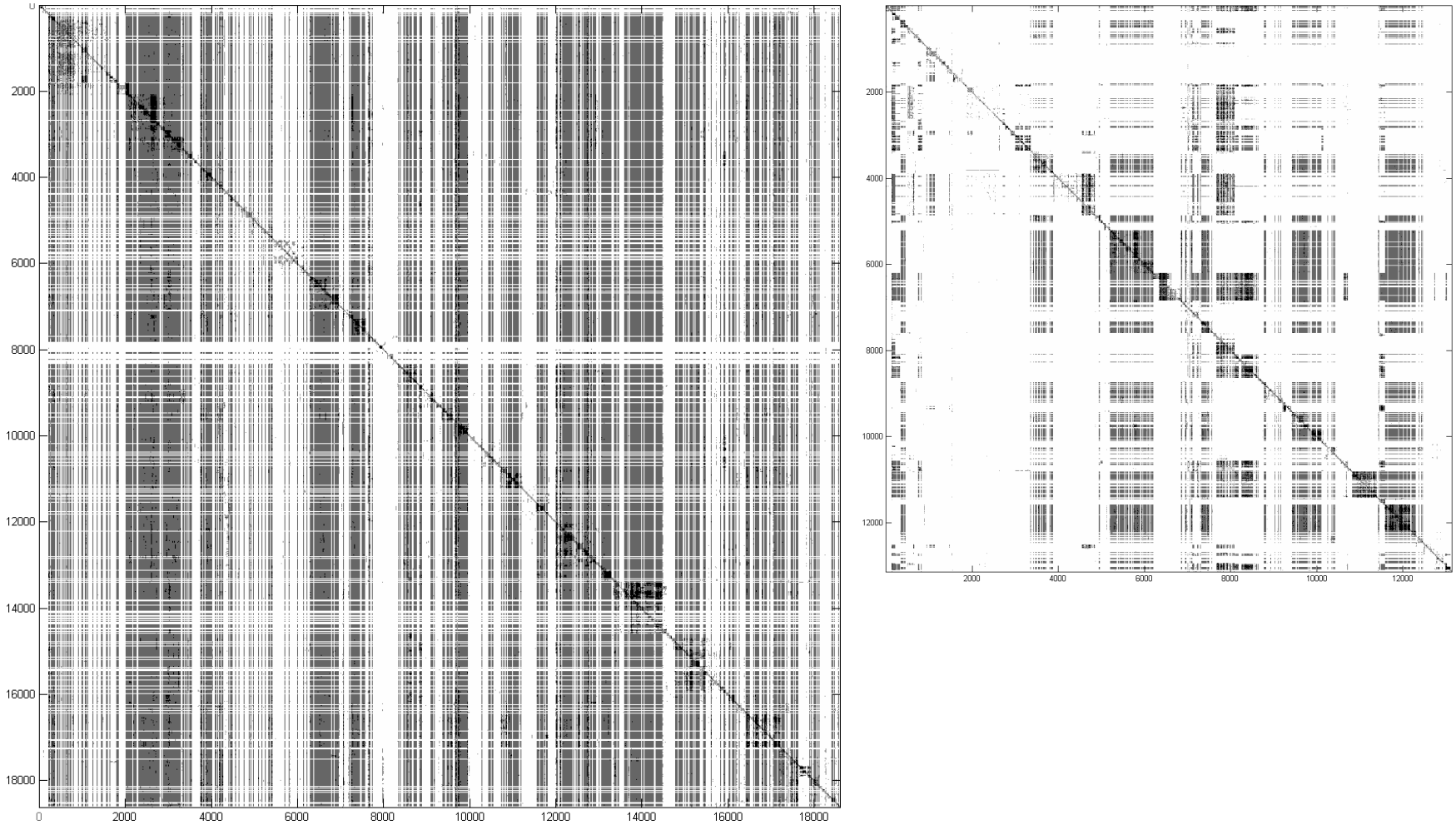


Figure 13: Visibility matrices of the product before and after redesign

7.3 Effort Data

There has not been much research done in the area of measurement of engineering effort spent performing maintenance tasks. An obvious reason for this gap is that research in this area can be quite challenging. On one hand, validity of data collected from artificial experiments can be criticized as non-representative of industry practices. On the other hand, data collected in the industry setting is difficult to obtain for a number of reasons, including lack of discipline in companies to collect relevant data and their unwillingness to share any information with researchers.

Data obtained for this study holds a promise of being both representative of the industry practice and valid from the experiment design perspective. Analyzed data set was selected based on the assumption that corrective code changes were urgent enough, so that the productivity of software engineers working on corresponding code modifications was consistently high and the need for precise tracking of issues demanded by customers prompted engineers to track their progress with utmost rigor. Finally, collected data represents the work of the same development team where the average experience level of engineers working with the products was maintained the same between the studied products.

Figure 14 shows distributions of observed code changes by the time software engineers spent designing a change. Cumulative distribution of empirical measures of resolution time is presented in the Figure 15 to facilitate results comparison. Table 4 has a quantitative description of the same set of the data except for outliers found beyond 120 hours threshold. The decision was made to not include outliers in the mean value calculation. These outliers represent misdiagnosed issues that required significantly greater effort than can be justified by a change request of corrective maintenance nature. Removal of the outliers from the statistical analysis adds a slight bias to the results against the proposed theses because of a longer tail for the old product.

Figure 14: Side-by-side comparison of products. Distribution of change requests by time spent developing a code modification and polynomial trend lines.

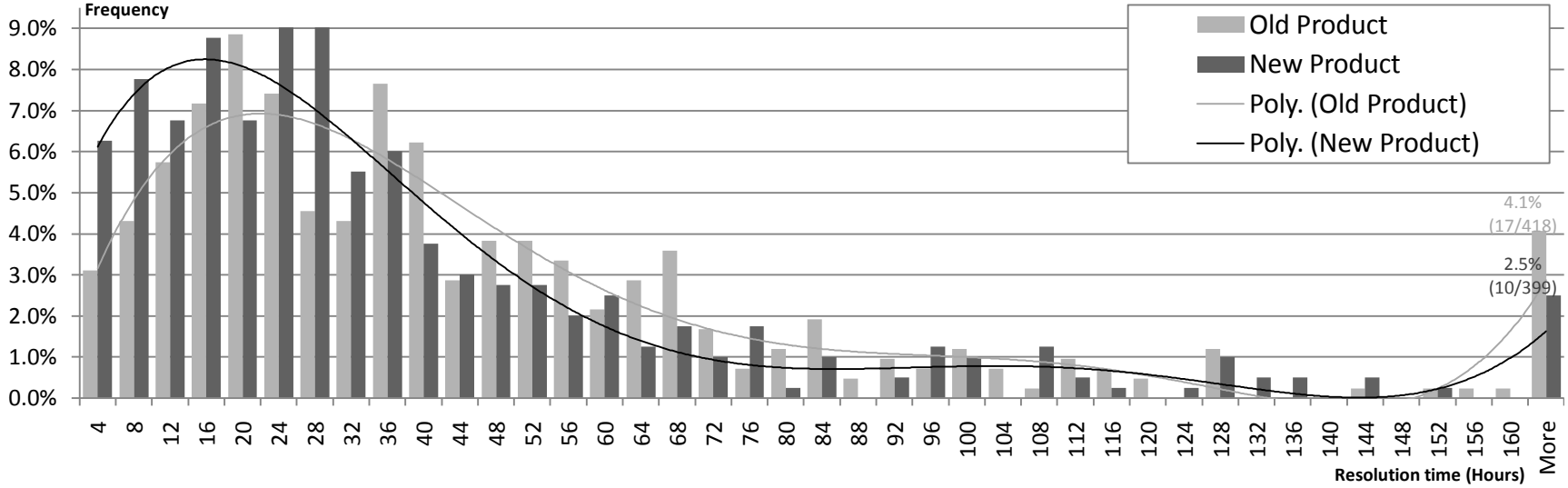
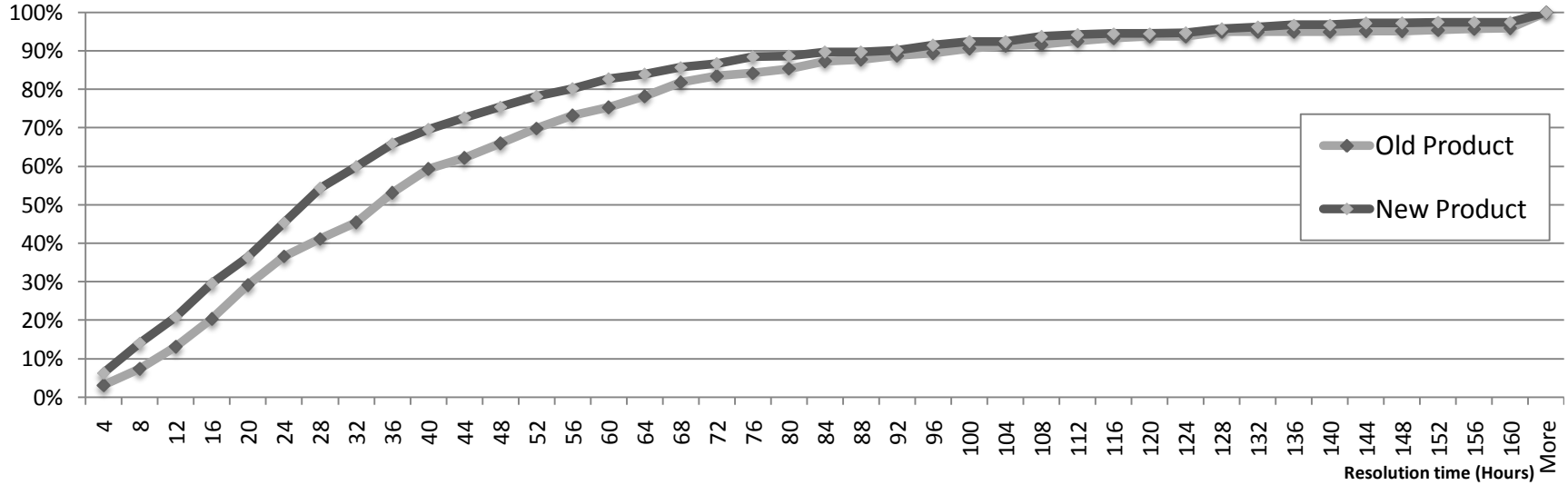


Figure 15: Cumulative distribution of empirical measures of resolution time



	Old Product	New Product
Mean	38.2	32.8
Standard Error	1.3	1.3
Median	33	26
Standard Deviation	26.4	25.4
Sample Variance	697.4	644.7
Range	119	120
Cumulative Effort	14,964	12,333
Modifications Count (number)	392	376

Table 4: Resolution time statistics (hours per corrective code change, unless noted otherwise)

Analysis of the data shows that on average implementation of a corrective code modification took engineers working with new product code base 14% less time than engineers working with the old product. 14% performance improvement resulted in more expeditious delivery of solutions. As can be seen from the cumulative distribution of resolution time measures graph (Figure 15: Cumulative distribution of empirical measures of resolution time), a corrective code modification targeted at new product code base has 70% probability of being finished in less than 40 hours of reported time, which amounts to approximately one person work week. In comparison, code modifications against the old product codebase have only 60% probability of being finished after a similar amount of time spent by an engineer designing the change.

Figures 16 and 17 show distributions of observed code changes by the total effort, adjusted to reflect its variability over time, that software engineers spent designing the change. Table 5 (Table 5: Productivity adjusted effort statistics (hours per code change, unless noted otherwise)) has a quantitative description of the same data. A few outliers were not included in the mean value calculation. In the presented figures, issues that required more than 76 hours of productivity adjusted effort were excluded from the quantitative analysis. These outliers represent misdiagnosed issues that required significantly greater effort than can be justified by a change request of corrective maintenance nature.

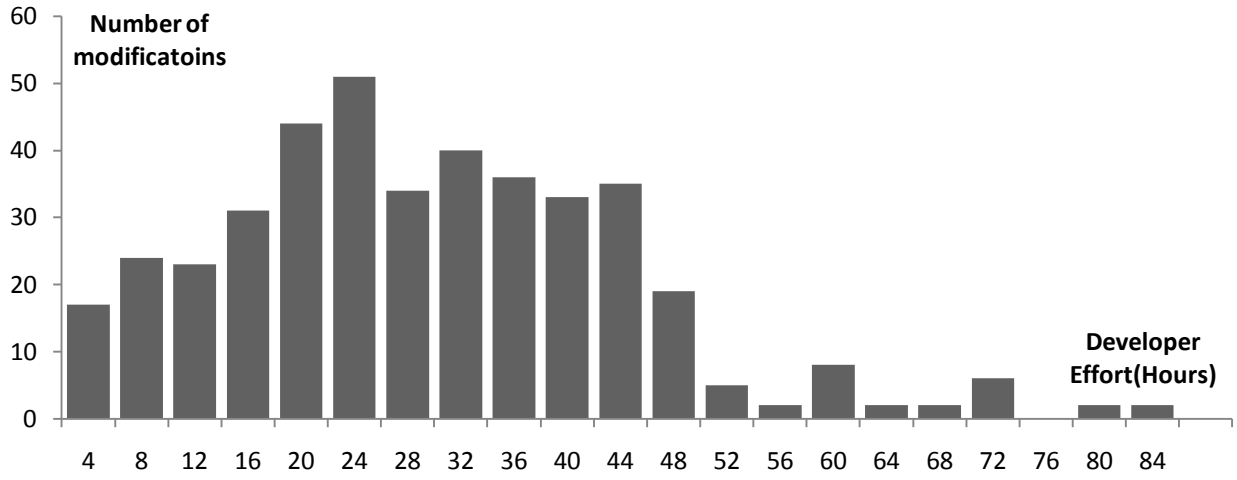


Figure 16: Distribution of corrective code changes by effort, adjusted to productivity variability, spent developing a code modification (Old Product)

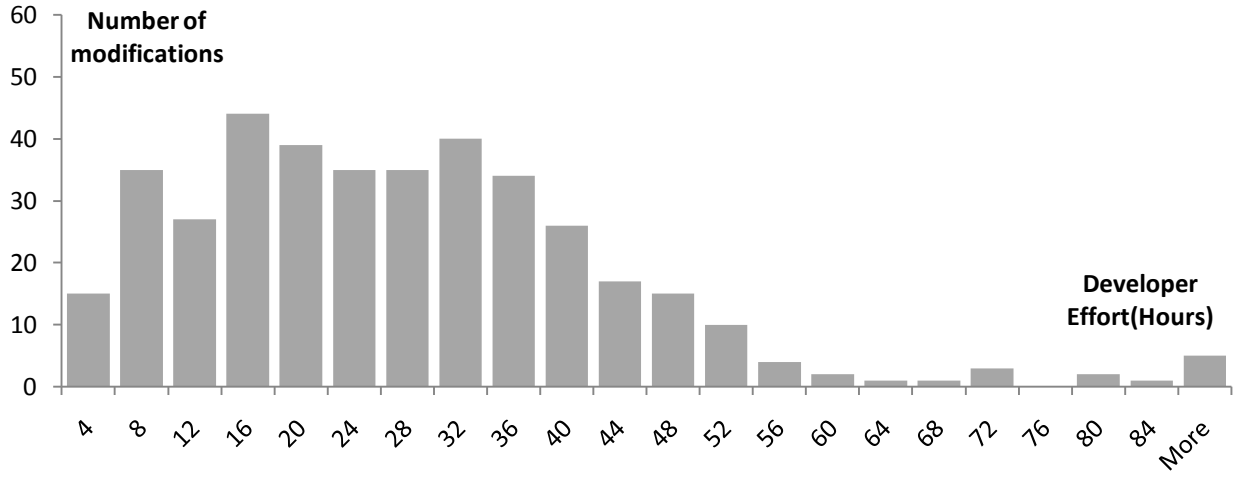


Figure 17: Distribution of corrective code changes by effort, adjusted to productivity variability, spent developing a code modification (New Product)

Analysis of the data shows that on average performance of engineers working on corrective code modifications improved by 10% after the re-design of the product. While working on code changes for the old product developers reported on average 27.5 hours per single change, with the new product average effort dropped to 25 hours. For a side by side comparison of

normalized distributions of changes by effort spent developing a corresponding code modification see figure 18 (Figure 18: Side-by-side comparison of products. Distribution of change requests by productivity adjusted effort spent developing a code modification and polynomial trend lines.).

The share of change requests that had at least one cycle of rework decreased significantly with re-design. In the studied context rework cycles may be caused by a new defect or incompleteness of a proposed code modification found during peer code review or the testing phase of the corrective maintenance work process. Once a defect is identified, the developer responsible for the modification is pulled back to work on the code change, so that resulting code is bug free. Fraction of issues that had at least one rework cycle dropped from 12% to 9% after the product redesign. 25% reduction in the number of modifications that undergo additional rework cycles is a significant improvement. Rework cycles are usually a result of hidden dependencies that were not discovered by an engineer designing the initial code modification. Undiscovered hidden dependencies often lead to functional regressions getting submitted as a part of corrective code modifications. Engineer's ability to identify all dependencies while designing the code modification is crucial to maintaining existing functionality of the product through the maintenance phase of the product lifecycle.

It is noteworthy that the average number of source files "touched" per corrective change increased from 2.6 to 2.8. It may be due to a higher cohesiveness of code in source files within a single component. However, the average time spent working with a source file decreased from 10.6 hours to 8.9 hours, which translates into a 16% improvement.

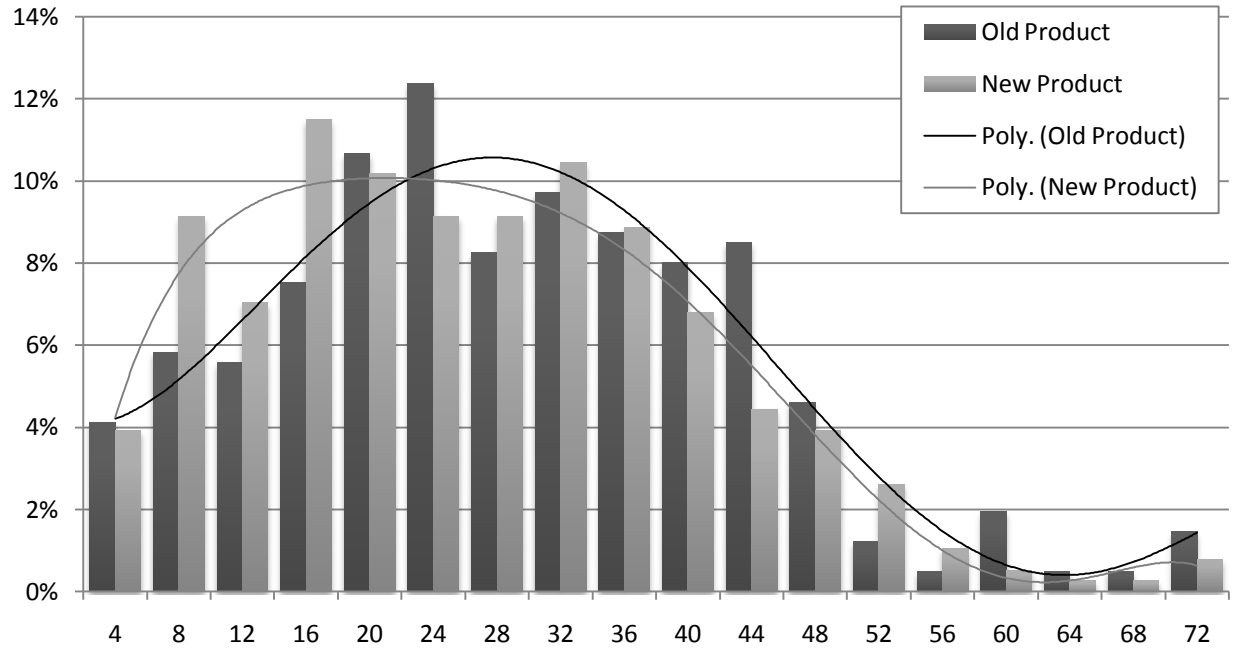


Figure 18: Side-by-side comparison of products. Distribution of change requests by productivity adjusted effort spent developing a code modification and polynomial trend lines.

	Old Product	New Product
Mean	27.5	25.0
Standard Error	0.7	0.7
Median	26	24
Standard Deviation	14.7	13.9
Sample Variance	215.4	194.5
Range	71	71
Cumulative Effort	11332	9582
Modifications Count	412	383
Avg. number of files touched per change	2.6	2.8
Avg. effort per source file	10.6	8.9
Re-work data		
Number of issues with rework cycles	50	35
Rework issues as a fraction of all issues	12%	9%

Table 5: Productivity adjusted effort statistics (hours per code change, unless noted otherwise)

7.4 Hypothesis One: Link between complexity of the product and maintenance effort

The results discussed above strongly support the first hypothesis proposed for this study. Decrease in product's propagation cost from 38% to 11% can be linked to the 10-14% improvement in developer productivity. Besides engineer productivity such metrics as amount of rework and effort needed for a single file modification improved by 16-25%.

7.5 Hypothesis Two: 'Core' source files are more susceptible to change

The second proposed hypothesis was that core modules - files that demonstrate High Fan-In and High Fan-Out visibility measures - get modified more often than any other type of components. In this study only the old product had a distinct "core-periphery architecture (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010). Hence, the measurements below are based on the data collected for the old product. Analysis of all source files that were touched during the first 30 months of maintenance from the old product release date shows that core files are significantly more likely to be modified (Figure 19: Distribution of individual source files changed by module type). Out of 542 unique files that were touched in the old product code base 404 were of the "core" type. Frequency with which "core" files were modified is disproportionate to their share of the whole system. As described in table 3 (Table 3: Quantitative comparison of products complexity) "core" files amount to only 39% of all modules comprising the old product.

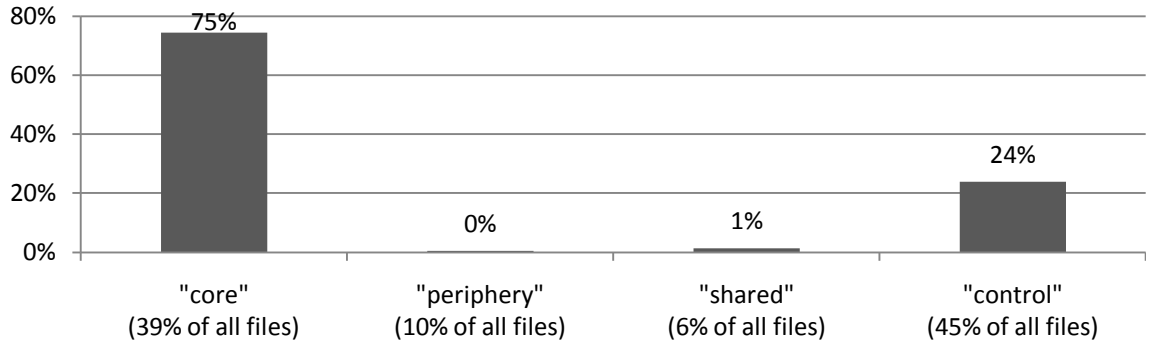


Figure 19: Distribution of individual source files changed by module type for the old product

Structural analysis of all changes showed that most changes contained at least one "core" file. Only 8% of corrective maintenance code change submissions did not contain a "core" file. It is notable that most non-'core touching' fixes touched only one source file. This indicates that developers were able to localize code modification to a single source file in order to implement a required corrective change when non-'core' file contained the implementation for the affected functional requirement. In other words ripple effect of non-'core' modules was low.

Analysis of how frequently isolated source files were touched showed that most frequently modified files are of a "core" type. 85% of files that were modified more than once were "core" files (115 out of 135). Some of "core" source files were modified 10-12 times in the studied time period.

Second most frequently modified type of files was "control" – files that have High Fan-Out and Low Fan-In visibility measures. In the old product 24% of files that were changed in the first 30 months of maintenance were "control" files. Even for the new product, 19% of modified files were "control" files, which is disproportionate with 10% share that "control" files occupy in the new product code base (Figure 20: Distribution of individual source files changed by module type for the new product).

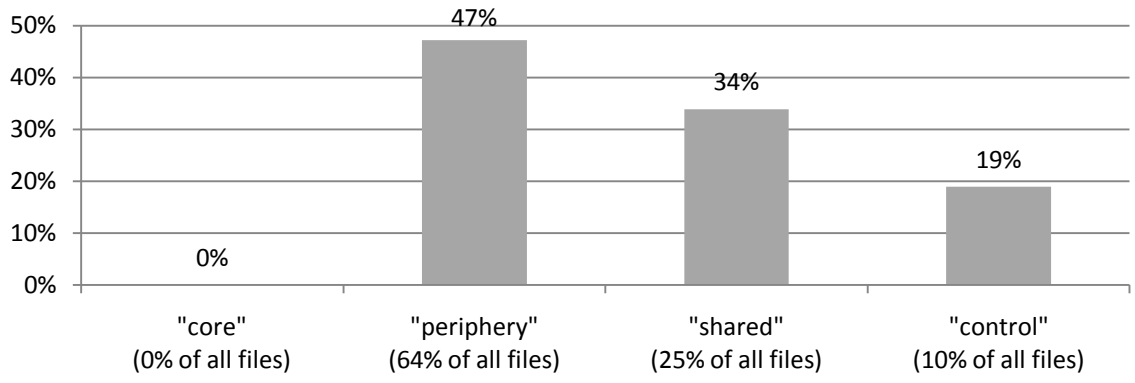


Figure 20: Distribution of individual source files changed by module type for the new product

Across both products the probability of modification of a "Periphery" file is lowest. In the old product codebase, probability of a code change going into a "periphery" file was close to 0%. In the new product the absolute probability value increased to 47%. This is mostly due to the fact that the vast majority of the files in the new product code base are "periphery" files (64% of all files).

7.6 Hypothesis Three: Measuring economic benefit of reduction of product complexity

As a software product goes through its lifecycle, the focus of a software development organization shifts from functional product improvements to cost reduction. As mentioned earlier in this paper costs of maintenance overshadow costs of initial software development by a large margin. Maintenance costs may reach up to 90% of the total product lifecycle cost. In such setting, 10% productivity improvement during the maintenance phase may result in savings almost as large as the cost of the initial product development. As the pressure for lowering costs of the maintenance phase increases, redesign that reduces structural complexity of the product becomes more economically viable and desirable.

Of course, any redesign may have downsides that need to be considered by management before the decision to invest resources into redesign of an existing

product can be made. Redesigns often introduce a lot of new code that will have new defects. Fixing of these new defects will have a cost to the organization. If existing software system reached a superior level of stability – very few corrective change requests are made over the prolonged period of time – redesign may not be as beneficial. Redesigned products have marginal new value for existing customers - if functionality is the same there is no reason to switch to the new product. From the marketing perspective redesigns are not as desirable as new product features. There are also deployment costs that any development organization needs to account for before committing resources and time to product redesign effort. It is essential for a development organization to perform a comprehensive Net Present Value (NPV) analysis before a decision to redesign an existing product can be made. Obviously, NPV-negative initiatives should be avoided.

8. Conclusion

This study aimed to demonstrate the link between design structure complexity of a software system and the maintenance costs. Empirical data analysis presented in this paper supports formulated hypotheses and complements the wealth of public knowledge on the topics of complexity and software engineering economics.

The scientific contribution of this work is in development of an industry based experiment to reliably measure software engineer's effort of working with sourced code. Resulting data was used to estimate the effect of design structure complexity on software developer's productivity. By measuring both, software complexity and engineers' effort required to perform similar maintenance tasks for two distinct versions of the software product it was possible to demonstrate the link between product design structure and maintenance costs.

Design of code changes is only one of the steps of the whole corrective change process. However this step is affected the most by the design structure complexity of the product. Traditionally, corrective code change process includes such steps as problem investigation, design of the code change, regression testing and fix distribution. Out of these four major steps problem investigation is the most difficult to control. The effort spent by engineers performing this step varies tremendously depending on the skill and experience of an engineer, data availability, and criticality of the problem. Regression testing and fix distribution steps can be improved greatly through more efficient operations and automation. These steps depend on product complexity, however the dependency is weak. Designing corrective code changes step depends on productivity of highly skilled engineers and represents a significant portion of costs of the maintenance phase of software lifecycle. As demonstrated in this paper, re-design of a product can improve productivity of engineers by more than 10%. This improvement can translate in a substantial cost reduction.

The software complexity measures presented in this paper are based on module visibility measures developed by a group of researchers that includes Alan MacCormack, John Rusnak and Carliss Baldwin (MacCormack, Rusnak, & Baldwin, Exploring the Structure of Complex Software Designs: An Emperical Study of Open Source and Proprietary Code, 2004) (MacCormack, Rusnak, & Baldwin, WP# 08-038, 2008) (MacCormack, Baldwin, & Rusnak, WP# 4770-10, 2010). These complexity measures are well suited for measuring maintainability of software and can be used in the industry setting. As was demonstrated in the paper, by measuring propagation cost, Fan-In/Fan-Out Visibility of modules, and the "core-periphery" structure of a software system one can create a universal metric that satisfies robustness, normativeness, specifity and prescriptiveness requirements of a good maintainability measure. This metric can be used for measuring complexity of software products found

in different phases of product lifecycle and can be effectively employed in the process of controlling complexity of new and legacy software products.

Economic benefits of controlling complexity of products have significant managerial implications. Development organizations should use a combination of metrics to measure the complexity of new and legacy software products to control their lifecycle costs. Module interconnectedness and overall product complexity metrics introduced in this paper is one of the approaches that can be used. Measurable economic benefit of redesign may prompt managers of development organizations to dedicate time and resources to more frequent redesigns of both individual components and the product as a whole. Role of technical architects involved in new product development should be extended to include the evaluation of current product design complexity and finding ways to reduce complexity and improve maintainability of the final product while software system is being developed for the first time. As this study indicated, “core-periphery” product architecture has an inherent cost to it. Unless there is a functional requirement for having a large amount of modules of the “core” type - “core-periphery” design structure should be avoided.

9. Works Cited

Abernathy, W. J., & Utterback, J. M. (1978). Patterns of Industrial Innovation. *Technology Review , 80* (7), 40-46.

Agresti, W. W. (1982). Measuring Program Maintainability. *Journal of Systems Management , 33* (3), 26-29.

Baldwin, C. Y., & Clark, K. B. (2000). *Design Rules.* Cambridge, MA: The MIT Press.

Banker, R. D., Datar, S. M., Kemerer, C. F., & Zweig, D. (1993). Software Complexity and Maintenance Costs. *Communications of the ACM , 36* (11), 81-94.

Boehm, B. W. (1976, December). Software Engineering. *IEEE Transactions on Computers* , 1226-1241.

Boehm, B. W. (1981). *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice-Hall, Inc.

Coleman, D. (1992). Assessing Maintainability. *Software Engineering Productivity Conference* (pp. 525-532). Palo Alto, CA: Hewlett-Packard.

Coleman, D., Ash, D., Lowther, B., & Oman, P. (1994). Using Metrics to Evaluate Software System Maintainability. *IEEE Computer , 27* (8), 44-49.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., & Love, T. (1979). Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Transactions on Software Engineering , SE-5* (2), 96-104.

Eppinger, S. D., Whitney, D. E., Smith, R. P., & Gebala, D. A. (1989). *Organizing the Tasks in Complex Design Projects.* Massachusetts Institute of Technology. Cambridge, MA: Massachusetts Institute of Technology.

Fayad, M. (2002). Accomplishing Software Stability. *Communications Of The ACM* (Vol. 45. No. 1), 111-115.

Gibson, V. R., & Senn, J. A. (1989). System Structure and Software Maintenance Performance. *Communications of the ACM , 32* (3), 347-358.

Halstead, M. H. (1977). *Elements of software science.* New York: Elsevier.

Harrison, W., Magel, K., Kluczny, R., & DeKock, A. (1982). Applying Software Complexity Metrics to Program Maintenance. *IEEE Computer , 15* (9), 65-79.

IEEE. (1998). *IEEE Standard for Software Maintenance.* New York, NY: Institute of Electrical and Electronics Engineers, inc.

IEEE. (1988). *IEEE Std. 982.1-1988 IEEE Standard Dictionary of Measures to Produce Reliable Software.* New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

IEEE. (2005). *IEEE Std. 982.1-2005 IEEE Standard Dictionary of Measures of the Software Aspects of Dependability.* New York, NY: The Institute of Electrical and Electronics Engineers, Inc.

ISO/IEC, & IEEE. (2006). *Software Engineering - Software Life Cycle Processes - Maintenance.* Geneva: ISO/IEEE.

Kearney, J. K., Sedlmeyer, R. L., Thompson, W. B., Gray, M. A., & Adler, M. A. (1986). Software Complexity Measurement. *Communications of the ACM , 29* (11), 1044-1050.

Krishnan, M. S. *Cost, Quality and User Satisfaction of Software Products: An Empirical Analysis.* Graduate School of Industrial Administration, Carnegie Mellon University.

Lehman, M. M. (1980, September). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE* , 1060-1076.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., & Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. *Software Metrics Symposium* (pp. 20-32). Albuquerque, NM: IEEE.

Lemos, R. (2001, June 8). *Fix for MS Exchange causes mail problems*. Retrieved March 7, 2010, from CNET News.com: http://news.cnet.com/Fix-for-MS-Exchange-causes-mail-problems/2100-1001_3-268133.html

Lemos, R. (2001, June 12). *Microsoft Exchange bug: Strike three?* Retrieved March 7, 2010, from CNET News.com: http://news.cnet.com/2100-1001-268296.html

Lemos, R. (2001, June 7). *Security hole found in Exchange 2000*. Retrieved March 7, 2010, from CNET News.com: http://news.cnet.com/Security-hole-found-in-Exchange-2000/2100-1001_3-268022.html

Leyden, J. (2001, June 30). *MS patches Exchange 2000 email spy bug*. Retrieved March 7, 2010, from The Register: http://www.theregister.co.uk/2001/06/30/ms_patches_exchange_2000_email/

MacCormack, A., Baldwin, C., & Rusnak, J. (2010). *The Architecture of Complex Systems: Do Core-periphery Structures Dominate?* Cambridge, MA: MIT Sloan School of Management.

MacCormack, A., Rusnak, J., & Baldwin, C. (2004). *Exploring the Structure of Complex Software Designs: An Emperical Study of Open Source and Proprietary Code.* Boston: Harvard Business School.

MacCormack, A., Rusnak, J., & Baldwin, C. (2008). *The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry.* Cambridge, MA: Harvard Business School.

MathWorks, T. (2010, April 20). *MATLAB - The Language Of Technical Computing*. Retrieved April 20, 2010, from MathWorks: http://www.mathworks.com/products/matlab/

Microsoft Corporation. (2001, June 06). *Microsoft Security Bulletin MS01-030: Incorrect Attachment Handling in Exchange OWA Can Execute Script*. Retrieved March 7, 2010, from Microsoft TechNet: http://www.microsoft.com/technet/security/Bulletin/MS01-030.mspx

Oman, P. W., & Hagemeister, J. (1992). Metrics for Assessing a Software System's Maintainability. *Conference on Software Maintenance* (pp. 337-344). Los Alamitos, CA: IEEE Computer Society Press.

Pressman, R. S. (1982). *Software engineering: a practitioner's approach.* New York: McGraw-Hill.

Rothman, J. (2000, October). *What Does It Cost You To Fix A Defect? And Why Should You Care?* Retrieved March 13, 2010, from Rothman Consulting Group, Inc.: http://www.jrothman.com/Papers/Costtofixdefect.html

Schach, S. R., Jin, B., Yu, L., Heller, G. Z., & Offutt, J. (2003). *Determining the Distribution of Maintenance Categories: Survey versus Measurment.* The Netherlands: Kluwer Academic Publishers.

Seacord, R. C., Plakosh, D., & Lewis, G. A. (2003). *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.* Boston: Addison-Wesley.

Sharman, D. M., & Yassine, A. A. (2004). Characterizing Complex Product Architectures. *Systems Engineering , 7* (1), 35-60.

Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured Design. *IBM Systems Journal , 13* (2), 115-139.

Steward, D. V. (1981). The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management , EM-28* (3), 71-74.

Suh, N. P. (2001). *Axiomatic Design: Advances and Applications.* New York: Oxford University Press.

Suh, N. P. (2005). *Complexity: Theory and Applications.* New York, New York: Oxford University Press.

Suh, N. P. (1990). *The Principles of Design.* New York, New York: Oxford University Press, Inc.

Ulrich, K. (1995). The role of product architecture in the manufacturing firm. *Research Policy* (24), 419-440.

Warfield, J. N. (1973). Binary Matrices in System Modeling. *IEEE Transactions on Systems, Man, and Cybernetics , SMC-3* (5), 441-449.

Woodward, M. R., Hennell, M. A., & Hedley, D. (1979). A Measure of Control Flow Complexity in Program Text. *IEEE Transactions on Software Engineering , SE-5* (No. 1), 45-50.

www.scitools.com. (n.d.). Retrieved March 12, 2010, from Understand: Source Code Analytics & Metrics: http://www.scitools.com