Abstract

Today, software is found in almost all systems, vehicles, communication devices, medical equipments, and entertainment, for example. The size and complexity of these systems has grown continuously over the last forty years – the time span for modern computing. The latest release of the Windows operating system, called Vista, is expected to be more than fifty million lines of code, about 40% bigger than the previous version.

Some of the reasons for this are numerous feature demands and the need to support multiple platforms, and need for compatibility with legacy software and hardware. Each line of code, in these large systems, requires perhaps several technical decisions, often, but not always simple. The sheer volume of this decision making process is daunting. No single human can fully understand a system of high complexity. To help ameliorate this problem, systems are decomposed into subsystems, libraries, modules, and classes. Most of these components have interdependencies, in order to provide services, one to another. However, in systems of great size, the dependencies often become a dense web of relationships. It is exactly this problem on which we focus in this research.

We propose that static dependency structure is an important element to understand the state of large software system. We conduct various analyses using well-known existing open-source, commercial and expert developed projects, including our own projects to evaluate the overall effectiveness of our approaches. We detect structural problems in large software development projects, and present a structure metric to rank software files according to their risk contribution to the software system. Additionally, we present a model that indexes software components according to their potential for reuse. We design and conduct experiment to investigate the impact of change in one file on other files. Furthermore, we provide tools needed to support analysis, project visualization and monitoring. Finally, we investigate corrective procedures and simulate their application, monitoring improvements in observed defects.

STRUCTURAL MODELS FOR LARGE SOFTWARE SYSTEMS

By

MURAT KAHRAMAN GÜNGÖR

B.S. Sakarya University, 1997
M.S. Syracuse University, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements for the

degree of Doctor of Philosophy in Computer and Information Science

in the Graduate School of Syracuse University

July 2006

Approved: _____

Advisor Professor James W. Fawcett

Date _____

The Graduate School
Syracuse University

We, the members of the Oral Examination Committee,
hereby register our concurrence that

Murat Kahraman Güngör

satisfactorily defended his dissertation on

July 2006

Examiners:

Ehat Ercanlı _____
(Please sign)

Can Işık _____
(Please sign)

Daniel J. Pease _____
(Please sign)

Marek Podgorny _____
(Please sign)

Advisor:

James W. Fawcett _____
(Please sign)

Oral Exam Chair:

Yıldıray Yıldırım _____
(Please sign)

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my sincere gratitude to

- Dr. James W. Fawcett for his support and guidance throughout this research. He always encouraged me and gave me hope whenever I encountered problems during my research. He is my advisor, my mentor, and my friend. I am unable to find right words to express my appreciation for all he has done for me.

- My dissertation committee members, Dr. Ehat Ercanlı, Dr. Can Işık, Dr. Daniel J. Pease, Dr. Marek Podgorny, and Dr. Yıldıray Yıldırım.

- Many friends and colleagues for their support and valuable comments: Kanat Bolazar and Bing Xue, I thank you for valued comments and criticisms. Arun V. Iyer thanks for your collaboration on our early Mozilla data extraction, and contributions of several useful tools Sergey Karamov, Bülent Çetinkaya and everyone from our research group thanks for your support.

- My wife, Reyhan, who shared my trials during my Ph.D. studies, for her support, understanding, encouragement and patience.

- My daughter, Nilufer Sena, and my son, Suat Emin for becoming a source of happiness in our lives.

- My mother, Ayşe and my father Ali Rıza Güngör and parents-in-law Nurhan and İsmail Aydemir for everything they have done for me.

Chapter 1

Introduction

The primary objective of this research is to understand how to detect structural problems in large software development projects, then, to generate algorithms and methods to diagnose specific structural flaws. Another objective is to provide tools needed to support analysis and project monitoring. The final objective is to explore possible corrective procedures and simulate their application, monitoring improvements in observed defects.

Chapter 1 provides an overview of our research, discussing its focus, its methods, our accomplishments, and their relationship with the work of others. Subsequent chapters explore each of our focus areas, and the last chapter summarizes our conclusions and proposed areas for future work.

1.1. Motivation

Modern software systems are often very large and complex. For example, Windows Vista, the latest version of the Windows operating system, will be released within a few months, with more than 50 million lines of source code [2]. Each line of that code required perhaps several technical decisions, often, but not always simple. The sheer volume of this decision making process is daunting. No single human can fully understand a system of this complexity, and, because the decisions are made by humans, not all will be correct.

To help ameliorate this problem, systems are decomposed into subsystems, libraries, modules, and classes. Most of these components have interdependencies, in order to provide services, one to another. However, in systems of great size, the dependencies often become a dense web of relationships, as we will show in the next section, and in more detail in Chapter 3. These dense relationships make development difficult. It is exactly this problem on which we focus in this research. Our goal is to provide automated analysis of structure, and structural problems, and to demonstrate explicit means to resolve problems, so discovered.

Our goal is to provide techniques for managers of large software projects to view the current state of their project's products, throughout software development and maintenance. We have studied existing projects to try to understand ways to do that. Our current work has shown that static dependency structure is an important element of that analysis.

Screening static structure provides both quantitative and qualitative information regarding structural problems, as shown in Chapter 3. Structural data can be obtained automatically via source code dependency analysis. For large software systems, this is a key attribute of our analysis approach, showing how pieces are interconnected with each other. Some of the important characteristics that dependency information reveals are size of fan-out, fan-in and strong components (each strong component is a set of mutually dependent files).

For example, depending on scores of other files (large fan-out) may indicate a lack of cohesion – the file is taking responsibilities for too many, perhaps only loosely related, tasks

and needs the services of many other files to manage that. Fan-in is the number of files that depend on a file. This indicates a lot of reuse, which is good, but high fan-in coupled with low quality creates a high probability for consequential change, and risk to the cost and schedule of the project. If we need to improve the widely reused file's code, that may break the implementation of many of its using files.

Strong components occur when files have mutual dependencies between them. Understanding, testing, reusing and adding new features becomes harder due to complex dependency among the members of component. In the following section, we explore these specific structural problems.

1.2. Problem Statement

In order to understand large software systems, we analyzed the structural quality of some existing software systems in terms of their dependencies. One of the software systems that we studied was Mozilla, version 1.4.1, an open source browser project. Mozilla was targeted to become the browser for Apple OSX.10 but Apple decided to build its own, Safari, based on the KHTML rendering engine[1] [65]. This decision was based on the size, complexity, and performance of the existing Mozilla base line [66]. The Mozilla project eventually abandoned much of the 1.4.1 code base before continuing with development [66]. The large size – 6,193 source files – and acknowledged problems makes this code an interesting object for study. Can we understand why this version of the product was unsuccessful? Can we find effective ways of improving the system, without detailed knowledge of its low level design details? In an attempt to answer these questions, we performed a type based file-to-file dependency analysis over several of the Mozilla libraries, focusing especially on GKGFX, a major library, within Mozilla with 598 files. We obtained a lot of interesting information about the structure of this code, including Figure 1.1 and Figure 1.2 below.

[1] KHTML is another, competing, open-source browser project.

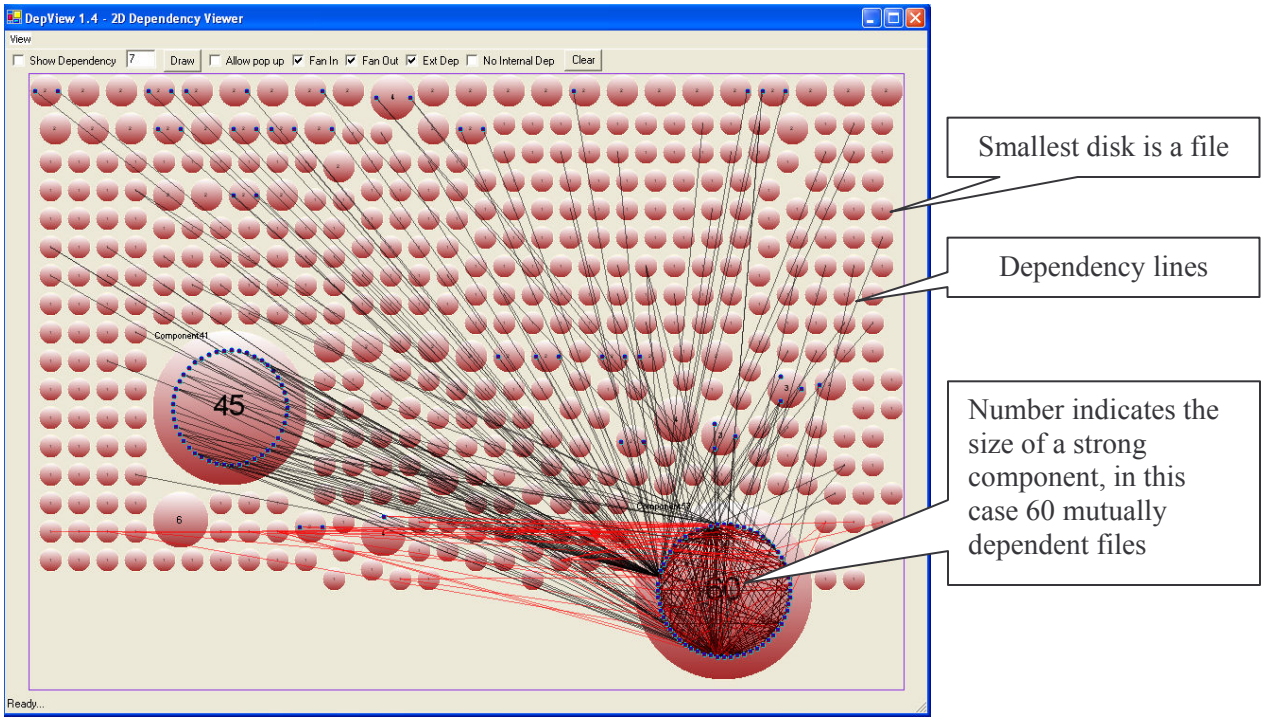


Figure 1.1 – Internal and external dependencies of component #57.

The two figures, Figure 1.1[2] and Figure 1.2, represent dependency relationships within the GKGFX library (Mozilla's NGLayout Project [56]) from the Mozilla project version 1.4.1. In this figure, the smallest disks represent individual source files; all larger disks represent strong components, e.g. sets of mutually dependent files. The number at the center of each circle indicates the size of a strong component (number of files). A line between circles shows dependency among files.

Figure 1.1 shows internal and external dependencies of the largest strong component within the GKGFX library. This figure reveals that the strong component uses services of many individual files and members of other strong components. In addition, Figure 1.1 adds dependencies on files, outside the same strong component, on files inside, indicating services it provides to these files.

[2] These figures were generated by our visualization tool, DepView.

The large disk in the Figure 1.2 represents the same strong component shown in Figure 1.1, a collection of mutually dependent files, 60 in all. Every one of these files depends, either directly or indirectly, on every other. The dependency relationships are shown by dense lines within the disk. Each dot around the circle is one of the 60 files. If any file inside the strong component is changed, it may break the operation or design of any other file in the component and any of the external files using services of this component, as in Figure 1.1, e.g., 60 plus many more.

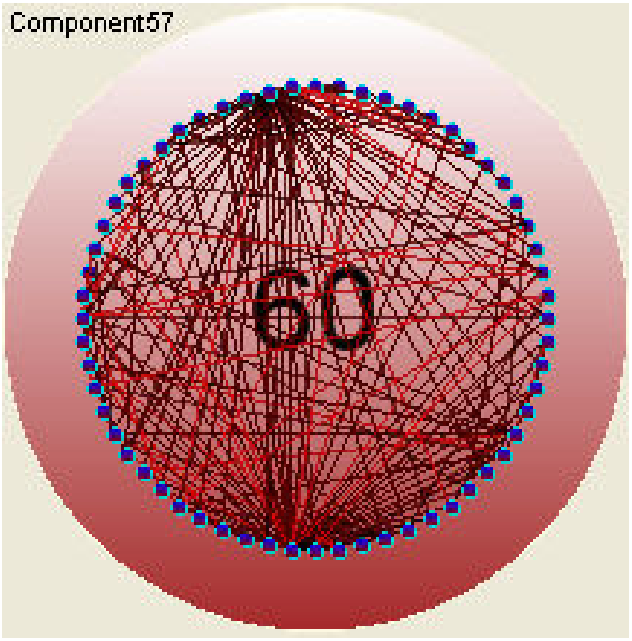


Figure 1.2 – Internal dependencies of component #57 consisting of 60 files.

The complexity of these dependency relationships demonstrates that this component has extremely poor testability characteristics. Should a developer find and fix a defect in one of these files, a huge number of other files – more than 60^3 – need to be retested to demonstrate that the change caused no other breakage. There are dense dependencies not only within the strong components, but also among the strong components. This is an indication of high

[3] A change in a file inside this strong component requires retesting all sixty files inside the component, and all of the many files outside the component, which depend on files within. These dependencies are the ones shown in Figure 1.1.

coupling throughout the GKGFX library. Additionally, due to these dense dependencies, making changes and tracking the effect of those changes is difficult. Therefore, extensibility - new feature addition – becomes difficult.

The figures above reveal particular issues with the development of large software systems in general. If dependency between components is dense, that causes several undesirable attributes. First, it is very hard to reuse files from the component that, because they depend on so many other files. Extracting them is very difficult, because each extracted file depends on so many other files. Second, it is hard to test files in the component effectively because every time a test uncovers a defect, which we fix, we have to retest all the previously tested files in the component because of their mutually dependencies. Finally, it is very difficult to understand the behavior of these systems because of their dense relationships.

In this section, we presented a few of the results of our analysis of the open-source Mozilla project, version 1.4.1. We concluded that the source packaging of its GKGFX library makes it difficult to test and understand its behavior. In the next section, we present some of our views, and the views and conclusions of other researchers, concerning these issues.

1.3. Other's Statements Relating to Problems in Large Development Software.

It is a natural consequence of development that as a project gets larger, dependency among its components gets denser and grows more complex. This dependency is necessary to provide services from one component to another; on the other hand, excessive dependencies make a system inflexible and fragile. The project becomes difficult for developers to understand, test, maintain and reuse.

It is very important to provide timely feedback to software engineers and project management about the state of a software development project, emphasizing these dependencies. This implies that monitoring the state of a large software development project is important and will be a major focus of this research. Most of our work is concerned with

dependency structure of large software systems. Early detection of structural defects will avoid delays, difficulties and costs associated with fixes made later in the project lifecycle. Higuera and Haimes [54] reported that:

> *"Many of the most serious issues encountered in system acquisition are the result of risks that either remain unrecognized and/or are ignored until they have already created serious consequences".*

Source code itself carries valuable information that we can monitor frequently. Software source is always accessible to its developers and managers, and carries up-to-date information, unlike project documentation, which may be out of date or may not exist. Moreover, source code provides quantitative information that can be turned into qualitative symptoms of several types of important problems. This can be used to provide timely feedback to software engineers and project management about the state of the software development project [40].

Software systems can be extremely complex. Developing large complex software systems is difficult, not just due to structural complexity, but because features of that complexity are essentially unique. When there are common implementation details in a software system the common structures are factored out into a single service. Other complex systems, VLSI chips, for example, use repeated structures, so understanding a modest number of relatively small cells may translate into understanding a major subsystem. Here is what Fred Brooks, Kenan Professor of Computer Science, University of North Carolina, Chapel Hill [1] has to say about software complexity:

> *"Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the*

> *statement level). If they are, we make the two similar parts into one, a subroutine, open or closed. In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound."*

And later in the same reference [1], he says:

> *"Much of the complexity in a software construct is, however, not due to conformity to the external world but rather to the implementation itself – its data structures, its algorithms, its connectivity."*

This complexity, coupled with organizational factors, has been responsible for a number of noted software disasters: Therac-25 X-Ray machine malfunction resulted in the deaths of several patients, due to a race condition[4], 1985-87 [3], the Denver Airport Baggage System failure[5], 1995 [4], Ariane 5 crash, due to arithmetic errors coupled with specification and design errors:

> *"Very tiny details can have terrible consequences", "That's not surprising, especially in complex software system such as this is", Jacques Durand, head of the Ariane 5 project, in Paris.*

1996 [5], and Mars climate orbiters[6], 1999 [6], to cite a few. Complexity causes not only malfunctions in operational systems, but problems with the development process resulting

---

[4] This system is complex
[5] Problems with both mechanical and software complexity.
[6] Data in English units instead of metric in software application code.

in cost and schedule overruns and project cancellations. The Standish Group published a widely cited report claiming these survey[7] results, 1995 [7]:

1. 15.5% of responders reported cost overruns of under 20%. The rest were higher.

2. 13.9% reported time overruns of under 20%. The rest were higher.

3. 31.1% of all projects were cancelled.

We have been examining several large systems: the open-source Mozilla and KHTML projects, and the libraries MFC (Microsoft Foundation Class library, part of the Visual Studio Software Developers Kit), and STL (Standard Template Library, part of the C++ standard library). In addition, we analyzed our own tool implementations to verify our new methods, algorithms and tools. This gives us a mix of open-source, commercial, and expert developed code, on which to test our ideas. As you will see, the results are quite interesting. In the next section, you will find the summary of our goals and accomplishments.

1.4. Goals and Accomplishments

The goal of this research is to understand how to detect structural problems in large software development projects. Secondarily, we seek to devise algorithms and methods to diagnose structural flaws. Finally, another goal of this research is to provide tools needed to support analysis and monitoring of static structure.

Our primary focus is for systems that are so large that no one person can understand the entire semantics of the project. That drives us to use methods that do not require semantic analysis[8]. Lastly, we explore possible corrective procedures and simulate their implementation, and observe resulting improvements.

The number of source files, in these projects, is too large to pay individual attention to each file. We need a way to rank files based on their impact on system quality. We have several questions, which may help us to identify these files or groups of files. Which files

[7] Sample size of 365 respondents, representing 8,380 applications [7].
[8] Most of our analysis to-date is based on static type and function-based dependencies.

contribute most to large strong component size? Can we order the risk of files by using each file's interrelationships with other files in the system? How does internal quality of a file, and the files on which it depends, affect overall system quality?

Another aspect of software implementation is its malleability. It is easy to make changes to a small part of a software system, but much harder to understand the impact of such changes on the system as a whole. This has two potential difficulties: a change may improve the functioning of a small part, but in fact have undesirable repercussions on the larger system. In addition, change makes establishing reusable components more difficult, both because the components may change and become incompatible with other users in the larger system, or because the users change and can no longer correctly use the component.

Our goal is to enable a Project Manager to visualize his large code base and determine where corrective action is needed and continually monitor the development progress of his system. The static dependencies we have been discussing are visible on a micro scale. Each developer knows what other files her code depends on, but may not be aware of indirect dependencies and other files in the systems that depend on her code. The dependencies on a macro scale are invisible to humans, due to the overwhelming complexity of real large projects[9].

Without the help of analysis tools, it is difficult to understand a large project, evaluate its quality, and track progress effectively. Therefore, we generated tools that can handle analysis of large-scale software systems[10]. Chapter 2 covers generated tools and interpretation of extracted textual and visual data by them. Chapter 3 documents an empirical study of a large open-source project, which illustrates the value of these tools in understanding large software systems.

---

[9] A project developing 5 million lines of code in two and a half years needs about 350 developers, from an example used in CSE784 – Software Studio, details are in AppendixA.2 at page 166.

[10] We applied our dependency analysis tool, DepAnal, to the entire Mozilla system, all 6193 files. This processing consumed about four hours on modern desktop computer. An earlier version of our code required more than 24 hours for this analysis.

These observations led us to consider ranking files by their risk level. Files will be ranked, according to their risk contribution to the entire system. Files with high risk ranking then become the target on which developers focus first, in order to alleviate structural problems. In Chapter 4, we introduce a software product risk model by considering dependency relations among files and files' internal metric properties. If a file has poor internal quality and the system has many files that depend, directly or indirectly, on this file, its quality is a risk factor for the system. The system risk would be smaller if very few other files depended on this file. This idea is formulized in our risk model.

Additionally, this research focuses on the ability to identify components for potential reuse. We describe a model that indexes software components according to their potential for reuse. This reusability index ranks source code, in existing systems, based on its place in the structure of the system and its internal metrics. This enables developers to evaluate a file for reuse before looking at its code. Section 4.4 explains the details of our reusability index model and its application.

While developing the risk model, we studied the relative frequency of required consequential changes in files in the project, called Change Impact Factor (CIF). The product risk model uses change impact factor for every dependency relationship between files in a project. But, initially, we could supply only rough estimates for the values of these parameters. So we designed and executed an experiment to measure the CIF factors, as functions of time, for a real project. As part of this experiment, we developed a measurement process that can be applied to other projects, as well. In this way, a more accurate assessment of risk is obtained, in real time, as a project unfolds. Chapter 5 presents details of the experimental design, its application, and its results.

After identifying potential dependency problems, we also explore the effects of modifying different dependency types to improve the structure of a large system, without needing a detailed understanding of its internal semantics. We simulate the effects of these

changes to determine their value, in improving system structure. Chapter 6 presents the details of this study.

1.5. Method Statement

In this section, we describe study methods that we used to pursue this research.

1.5.1 Type Based Dependency Analysis

We focus on file level dependency information, since files are the unit of testing and configuration management. We are not interested in portraying type-to-type or function-to-function dependencies for the reason that we are dealing with large numbers of source files, every file can define several types, and this would increase the volume of analysis information to the extent that it would be difficult to draw conclusions about it. Note, however, that our dependency analysis, in fact, uses this information, extracted from source code.

File dependency information can be obtained quickly, using our analysis tools. Therefore, this information is always available, unlike project documentation, which may be out of date or may not exist.

Briefly, the dependency model can be described as follows, first we collect declarations of types, functions, and global variables, and then, we find invocations of these items across the files. Finally, we determine the dependency among files based on the collected declaration and invocation data. The dependency model used throughout this research is given below.

Dependency Model - file A depends on file B if:

- A creates and/or uses an instance of a type declared or defined in B
- A is derived from a type declared or defined in B (inheritance)
- A is using the value of a global variable declared and/or defined in B
- A defines a non-constant global variable modified by B
- A uses a global function declared or defined in B
- A declares a type or global function defined in B
- A defines a type or global function declared in B
- A uses a template parameter declared in B

These rules intentionally do not acknowledge dependency of a base type on its derived types even though it is possible that a derived type modifies protected data members of the base. Doing so, we believe, would identify potentially many false-alarm dependencies in well-designed systems. It would be interesting to compare analyses of a major system with this assumption and with a model in which the base is declared to depend on all derived types if it provides protected data.

1.5.2 Qualitative and Quantitative Measures of System Quality

We need quantitative inputs about a software system under study to evaluate the quality of system. With the data extracted from dependency analysis, there are two possible ways to proceed. One is to focus on mutual dependencies obtained from the dependency graph as strong components; the other examines dependencies among individual files. We show data gathering and processing flow we use during our analysis of software, in Figure 1.3.

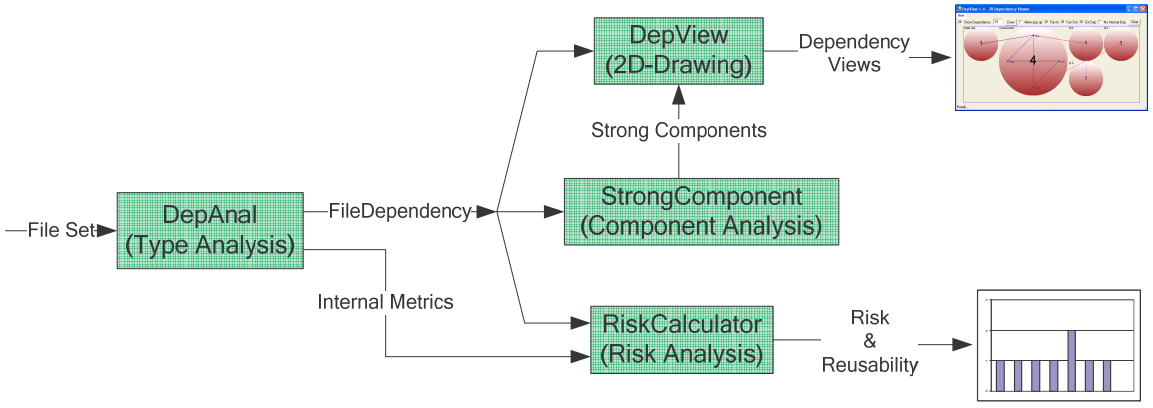


Figure 1.3 – Data Flow – During analysis and visualization of software system's quality

We first find dependencies between source files, and record this information, along with each files' internal metrics, using DepAnal. After obtaining this textual information, we

visualize dependencies, using DepView, and run our risk model-based analysis tool, RiskCalculator, to localize potential problems.

The level of detail, at which information is presented, is very important. Too much detail weakens comprehensibility. However, we need detailed enough information to understand the quality of the static structure, and need detailed enough information to locate the origin of structure problems. The two dimensional drawings in Figure 1.1 and Figure 1.2 immediately disclose the web of dependencies and size of interdependent file groups, within the file set analyzed. In addition, risk analysis merges file internal metric information with dependency information and provides an effective level of detail. Moreover, risk analysis enables us to identify files that may adversely affect the quality of the system.

Type Analysis is carried out by our dependency analyzer, DepAnal, a file-to-file static dependency analyzer for C/C++ source files, section 2.3.1 provides more detail. Two-dimensional Visualization is obtained by our tool DepView. Product Risk analysis is performed by using information generated by DepAnal in conjunction with our parsing tool called Matrix Maker. Additionally, we included a MatLab generated linear system solver[11] to evaluate the matrix equations that describe our risk model.

As shown in Figure 1.3, Product Risk Analysis uses internal metric information, along with file-to-file dependency data. During risk analysis, we use function size and cyclomatic complexity, but other potentially useful quantitative metrics could also be used - details are provided in section 4.1.5.

1.5.3 Finding Mutual Dependencies

After obtaining file level dependency information, we build a dependency graph, analyze its strong components, to find mutual dependencies, and then perform a topological sort of the components. This last sorting step is useful for visualization and is also useful in

[11] A stand-alone executable file.

developing test plans. In a classical test process, we start with the files that depend on no others, and then continue by testing only those files that depend on already tested files. For systems with strong components, this is not possible, due to mutual dependencies. Thus, the number and size of strong components gives important insights regarding how well a software project is packaged. The dependency density within strong components discloses how strongly coupled files are, and the strength of this mutual coupling is a compelling measure of test risk, because it measures potential for the need to retest large numbers of files.

1.5.4 Visualizers Providing Comprehensible View

Our DepAnal results are in text format. Drawing conclusions from these text files is almost as hard as reading source code. We developed the 2D dependency viewer, DepView, to obtain comprehensible views of large software systems. The 2D interactive views of dependency information discloses qualitative information about the system in an easily understood fashion – see Figure 1.1 and Figure 1.2.

1.5.5 Monitoring Development Manually

Our risk model depends on three things: dependency relationships, analyzed by DepAnal, internal metrics, analyzed by a program called Analyzer, and the probability that a change in some file will cause changes to be required in other files in the system. The first two items are analyzed by our tools, but the third is not.

In order to estimate the probabilities for change in files, due to changes in other files in the same system, we designed an experiment to explore propagation of changes throughout a project's development lifetime. DepAnal was redesigned and implemented from scratch, and each change carefully recorded. Then the data, so obtained, was analyzed to determine the change impact factor values (CIF).

1.5.6 Sample Analysis: Partial Analysis of Dependency Analyzer

We studied our own project, which has 30 source files. We know our project in detail, such as which files are hard to understand and which files need to be re-factored. However, we were not sure about the size of the strong component dependency between its files. The results we obtained, shown below in Figure 1.4, are encouraging by illustrating the effectiveness of the approach even for relatively small size software projects. Charts below expose high-level structural information. Performing a product risk analysis provides a finer level of detail, so that we can identify high-risk files.

Topological sort of the files
<-Most Independent
ITest.h
regexpr2.h
regexpr2.cpp
restack.h
syntax2.h
FILEINFO.CPP
FILEINFO.H
NAV.CPP
NAV.H
TOK.H
SEMI.CPP
SEMI.H
Utilities.cpp
Utilities.h
IncludeMngr.cpp
IncludeMngr.h
Grammar.cpp
Grammar.h
ScopeInfo.cpp
ScopeInfo.h
Collector.cpp
Collector.h
DepRecorder.cpp
DepRecorder.h
DepFinder.cpp
DepFinder.h
Main.cpp
reimpl2.h
syntax2.cpp
TOK.CPP
Most Dependent ->

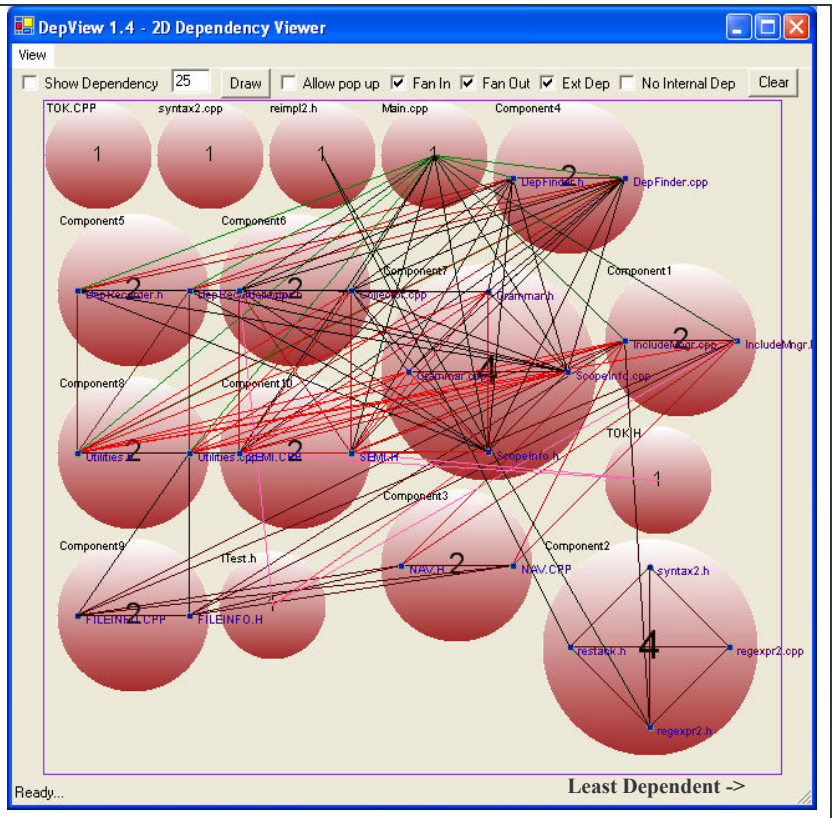


Figure 1.4 – New Design DepAnal Ver 1.7.a's internal dependency structure. Consists of 30 files

Figure 1.4 illustrates file level dependency structure of DepAnal. Before seeing this picture, we were not aware that DepAnal has a strong component with size of 4 files. The largest strong component carries more than 25% of the new source files in the project[12]. This view demonstrates detailed information without using our developer knowledge of this project.

[12] Some of the files taken, without development, from other libraries and projects.

As a developer of DepAnal, we know our code in detail; however, we were not aware of the existence of this component and this dense interaction between files.

In Figure 1.5, below each dot indicates dependency between two files. The count of dots shows the degree of communication density. A dot's distance to diagonal indicates whether a file's communicates with a file in its neighborhood or not. Moreover, if a dot is below the diagonal, it indicates existence of a mutually depended group of files.

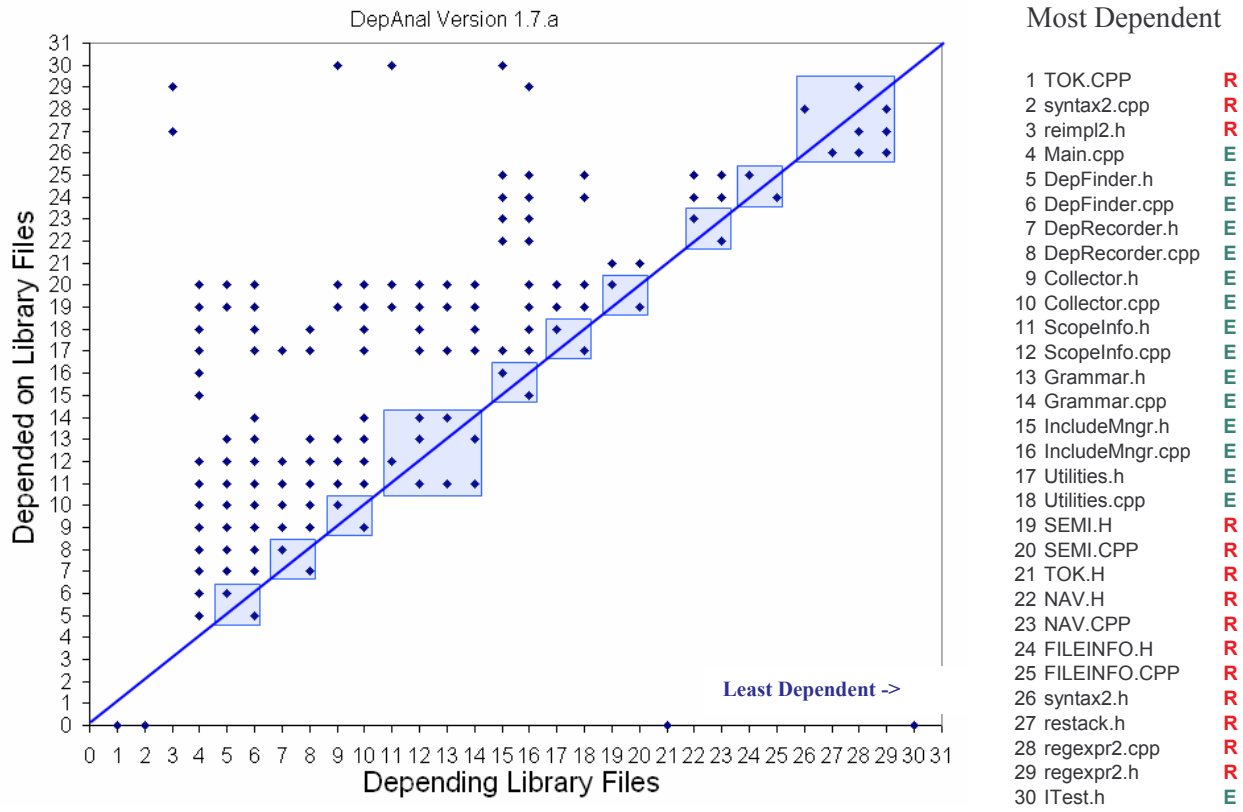


Figure 1.5 – Expansion of Strong Components – New-Design DepAnal Ver. 1.7.a

In Figure 1.5, we see strong components expanded into their individual files, each dot represents a dependency relationship. For any dot, the file vertically below it (x) depends upon the file horizontally to its left, on the ordinate (y). Any dot under the diagonal indicates the existence of a strong component. Each rectangle represents a strong component as shown in

Figure 1.4. The table on the right in Figure 1.5 shows the topologically sorted files. R indicates the file reused, E indicates file was created for this project.

The methods used to obtain this information have the capability of analyzing large-scale software, as described in a footnote on page 10.

1.6. Results and Contributions in Brief

This section briefly describes the thesis' results and contributions.

- Developed methods to uncover existing structural problems in software from source code.
- Developed tools to provide immediate feedback to software developers and managers about structural state of software development project, even for every large projects.
- Developed source file ranking algorithms using notions of product risk, importance, and testability of a file.
- Introduced a model that indexes software components according to their potential for reuse.
- Designed and conducted an experiment to evaluate change impact factor between source files. For this purpose we redesigned DepAnal, and during that implementation we monitored and recorded each change.
- Applied our tools on industrial projects to observe and report on the applicability and quality of estimation.
- The study also enables the identification of components, which need individual attention and suggest possible ways to avoid impending problems before they become chronic.
- The study enables a software manager to monitor a software project rapidly without waiting for documentation files to be produced, directly obtaining structural quality information from source code.

- Our empirical study has demonstrated that useful information about significant problems in both large and small systems can be identified without a detailed knowledge of the entire code base.
- The product risk model predicts that as the density of dependency relations increases in strong components of the dependency graph, Risk factor grows and becomes unbounded at critical densities.
- We explored the effect of different dependency types over dependency structure of a large system without a detailed understanding of its internal semantics.
- We have applied the model to a library from the Mozilla open-source project. The model predicted that most of the development risk is in about 10% of the library files. That good news was probably unknown to the Mozilla designers.
- We conducted statistical analysis of file properties versus change potential from Mozilla change database, using Multiple Linear Regression.

1.7. Literature Review

In the bibliography, we cite over 70 papers that are related to our work or have provided insight or inspiration for this effort. Here, we review some of them that have been particularly useful, and cited in this chapter.

1.7.1 Dependency Algorithms

Vassilios Tzerpos [10] developed a file to file dependency analyzer by looking at #include statements so that "*newcomers can grasp the overview of the system much faster, designers can inspect the quality and modularity of the system structure, and programmers have much clearer view of which part of the system they are actually developing*". In the same way, one of our goals is to find dependencies between source code files based on static type analysis. Unlike Tzerpos [10], we ignore include-based dependencies, as developers are occasionally careless about the files they include, causing dependency false alarms. Unnecessary

dependencies affect build efficiency but do not affect design or compilation breakage due to change. Instead, we focus on type-based file dependencies. Note also, that analyzing type dependencies affords the opportunity to base risk analysis and corrective actions on the types of dependencies encountered.

Kazman, S.J. Carrière [52] identified the necessity of tools in order to extract architectural information from source code, and explain how they achieve this by a workbench, Dali, a set of analysis tools. They have written scripts to extract "file depends on file" relations from make input file (makefiles). In our case we do not need any other information except source files themselves to extract dependency information. In addition, like #includes, make files sometimes carry incorrect dependency information, reducing the accuracy of the analysis.

Ferenc et al. [53] explain the reverse engineering framework, Columbus, which supports project handling, data extraction, representation, storage, filtering and visualization. It is a framework to help developers to comprehend the system under development. File to file dependencies are not their tools' direct output, but can be derived from its resulting output. This work focuses on specific type dependencies, while our focus is more specific towards source file dependency and its visualization. Files are the unit of testing and configuration management, and so file dependencies are a more appropriate measure of the system structure than its internal type details – we use these internal type details, but focus on the file-to-file dependencies they determine as a measure of system quality.

Martin Fowler [50] talks about dependencies (Coupling), and says they are necessary but should be reduced. One way to reduce dependencies, he suggests, is to use interfaces. And he stresses the need to avoid cycles[13]. He is concerned with a macro view of systems, similar to our study, and sees excessive dependency as a structural design problem. He has one basic rule that is "*to visualize high level dependencies and then rationalize them, separating the interface*

[13] Cycles are what cause mutual dependencies, e.g. strong components.

and implementation to break dependencies." This is one of several techniques, which can be applied to remedy structural design problems – we have already shown that type of dependency can be one of the effective directions to pursue.

Robert Martin [47] also studied general dependencies between modules. He classifies good (interface) and bad (concrete class) dependencies and considers the affect of dependencies on reuse. He introduces two metrics for instability and abstraction based on types, to define groups of files, which cannot be reused without one another, named Class Categories. We use, in our study file-based instability; Martin uses class-based instability.

Yijun et al. [25] use, parsing technologies CPPX or Datrix, which are similar to Columbus [53] for system analysis. File-to-file dependencies are not their tools' direct output, but can be derived from it. What they do is to extract a "code dependency graph" from the output of the parsers. However, we developed our own parser and dependency analyzer, and compiled into one standalone compact application without the need of other tools.

Rotschke and Krikhhaar [57] describe a tool with the goal of extracting automated UML information between the members of source files. Their tool has, however, an intrinsically different purpose, which is to understand architectural quality of software. For the large systems, UML discloses detailed relationships between pieces of software, which will be too much to comprehend. Granularity of our tools is file level, which, we believe, provides the right amount of detail for human consumption to understand structural quality of software. Similar to [57], Yu and Rajlich [28] consider hidden dependencies and change propagation. For data collection and analysis they consider type dependencies but not file to file dependencies.

To find file-to-file dependencies, using source files as input, we have developed model and implementation for C/C++ projects [59]. The model can be extended to object oriented programs like Java, C-Sharp etc.

1.7.2 Refactoring Software Systems

Yijun et al. [25] present a graph algorithm to remove unnecessary dependencies among files. They do this by reorganizing the header files using a class partitioning process. If a file depends only on some portion of a class in a header file, they distribute class into two or more classes, and as a result, they will have new header files. By eliminating unnecessary dependencies, they speed up compilation. However, they change the semantics of the code in a way that will cause a lot of retesting. Our study is different from theirs; we are not generating new code or partitioning classes into new classes. We use analysis and simulation to estimate the effects of constructive changes in Chapter 6. We have a similar goal to reduce dependency among source files, however our method is more applicable in our context, which focuses on multiple sources of dependencies, not just class-based.

1.7.3 Analyzing Quality of Source Files

Jungmayr [8] has explored analysis testability using a static dependency model, which was the inspiration of ours. He has proposed a definition of testability relating to the direct and indirect fan-out of components[14], proposes a metric methodology, without endorsing specific metrics, and reports on several experiments[15] using this approach. We differ in several ways from his work 1) by considering importance, 2) in distinguishing types of dependency relationships, 3) and in providing what we believe to be a more realistic[16] weighting of indirect dependencies. Inspired by a discussion by Jungmayr [8] on testability, we developed a file rank algorithm that ranks each file based on its testability – a function of its internal quality and the testability of the files it depends upon, and its importance – a measure of the number of files that depend on it. Just a note our work can be viewed as a further exploration of the same ideas.

---

[14] We have shown [11] that fan-out is highly correlated with change for the large Mozilla project.
[15] Jungmayr does not provide any technical details in [8] and we are unable to locate any concrete descriptions by him on these experiments.
[16] The merits of this weighting are argued in Chapter 5

Furthermore, unlike Jungmayr, we classify and treat differently dependency types, e.g., mutual, global, callback and simple type usage dependencies. One reason for doing this is that only dependencies based on simple type usage can be manipulated without breaking code, simply by rearranging code packages. All the other types are breaking changes. That is, we must change some aspect of the design to modify the dependency structure for these types.

Inoue et. al. [45] proposed a usage-based file rank procedure. Their goals are to retrieve components from a storage repository. Our ranking procedure is risk based with the goal of identifying components that have high risk of propagating changes. The methods of our study and the former have some similarities but the algorithms and final results are quite different. Our study uses a two-level structure with Test Risk and importance as the bases for ranking.

In our study, a file with high rank – one that needs to be corrected and tracked – has high importance and poor testability. The rank is essentially based on both micro qualities – those of the file being ranked – and macro qualities – how it relates to other files in the code base. Its importance is much like the importance of a web page. Many search engines on the internet use page ranking algorithms, as in Google, named PageRanks [41][42][43]. Mostly these rankings are determined by the number of unique external visitors. Similarly, another study was conducted to find impact analysis of publications, named “influence weight” [44]. And most recently, Component Rank method [45], based on abstract use relationships, for ranking software components to retrieve components from a storage repository, propagating significance throughout the use relations.

We have explored the use of a software metric set supporting examination of large systems and found a few to be useful, based on an empirical study supported by Multiple Linear Regression analysis. We have published a paper documenting these results [11] and another which uses a neural network [62] . Our methods, described here, use a function size metric and the cyclomatic complexity metric, demonstrated by Capiluppi and Ramil [9] to be effective, for the analyses of large systems.

Strong components, mutually dependent files, introduce the possibility of a chain of forced consequential changes when a single component member file is changed, perhaps to repair a latent defect or improve system performance. Lehman [55] states that when a system grows in volume and complexity, it may arrive at a point such that any further change to the system causes, on the average, one extra fault, at which point, the system becomes unstable or unmanageable. One of the goals of this research is to identify those characteristics to minimize such consequential changes. We have shown, using a risk model discussed below, that test risk becomes unbounded when density of mutual dependencies increases beyond a critical point[17].

We develop a file-rank algorithm, similar in concept to the page-rank algorithms used by some search engines, to identify particular risk areas, localize them to specific files, and suggest means for diminishing their risks. We have written research papers about this "Software Development Risk Model" [60][61]. This algorithm is novel, and uses direct and indirect dependencies to characterize testability and importance. When important files have poor testability properties, we have shown that larger than normal numbers of changes are likely.

Change impact analysis can be used to estimate the effect of proposed changes to other parts of the software. We have developed procedures to calibrate the change impact factor between source code files. Michelle L. Lee [71] considers, in her dissertation, the impact of change on types, global functions and global data, such as how many classes are going to be affected by a change. In this dissertation, we are interested in a coarser level of impact analysis, that of file-to-file change impact, as in the product risk model.

[17] The value of critical dependency density is a function of probability of an original change in a file causing consequential changes in dependent files.

1.7.4 Internal Metrics of Files

For purposes of testing and change control, it is important to find ways to estimate the effect of a change in one file on other files, which use the services of the changed file. Dependencies among source files reveal important information about possible impacts of a new change over a set of dependent files. However, not only file-to-file dependency information but also internal implementation quality of files should be considered. Capiluppi and Ramil have reported an analysis of cyclomatic complexity [9] [51], as it relates to what they term "release-touches", the number of releases in which a file has changed. They have shown that cyclomatic complexity is related to frequent change. And they stated "*... the source files which are subject to the higher change rate include a large portion of highly complex functions*". Based on these findings we chose to include cyclomatic complexity as part of our measure of file quality.

Parallel research has been carried out by Ping Yu, et. al. [40] and Basili et. al. [46]. Using regression analysis, they studied the relationships between 10 object-oriented metrics and the fault-proneness of a class. At the beginning of their research, they developed several hypothesis about the effect of metrics suits on fault-proneness, and they try to find out whether their hypothesis are supported by the subject system, which is written in Java with 123 classes. Some of the results that they report, Fan-in, Lines Of Code (LOC), and Number of Methods per Class (NMC) have statistically significant effects on fault-proneness. Most of their metrics are internal to a class, and their unit for this analysis is the class.

These four papers, along with our own, which relates metric values to observed changes in the Mozilla project [11], are important for this work because we use fan-in, fan-out, and complexity metrics, as well as others, as the basis for our characterization of large systems and file-rank algorithm is also utilizes these measures.

Software source code itself carries valuable information for monitoring the state of a project quickly; additionally software source code is always accessible and carries up-to-date

information, unlike project documentation. As stated in [40] "*Software metrics provide quantitative information that can be used in many ways to make assessment of the software products and the development process, to help engineers in coding practices and project manager in decision making… metrics data provides quick feedback to software engineers…*"

Our work uses a two level measure of software risk: internal metrics, like those discussed in this section, are used to define a notion of risk for a file in isolation. We combine that with test risk derived from the files it depends on, and a measure of a file's importance, based on the number of files that depend on it, to evaluate software risk of each file in a development project.

1.7.5 Visualizing Software Projects

Bassil and Keller surveyed and analyzed many software visualization tools, they stated in their paper [49] "*Today's software systems are increasingly large and complex, and their development and maintenance typically involves the collaboration of many people. This makes the tasks of programming, understanding, and modifying the software more and more difficult, especially when working on other people's code*", in their survey, one of the conclusions is "*the more the visualized software systems is large, the more important it is to visualize it graphically and the less useful it is to pass directly to the source code.*"

Another publication of Bassil and Keller [48] covers what are the most desired and used properties of Software Visualization (SV) tools. It states, "*concerning code analysis aspects, it seems only a low number of these aspects are supported by current SV tools*". And "*Calculation of metrics were the least supported, but were sometimes desired*". They stress the necessities of software visualization tools.

While we have worked on analysis of large software systems, the complexity of the systems that we study brought us to the same conclusion. For this reason, we developed a 2D file level dependency viewer (DepView, covered in section 2.3.4).

In the next chapter, we discuss elementary structural properties of software systems, and illustrate these properties with an analysis of the source code of our research tools.

Chapter 2

Analysis of System Structure

We have studied several open source software projects, mainly focusing on the Mozilla project, discussed in Chapter 3. Because it has very well maintained release information, code base controlled by release and each release is accompanied with a change history. Beside Mozilla, we have looked at KHTM, MFC and STL library files. However, we are not showing KHTML and STL analysis. These studies were carried out with static source code analysis. In the following sections, we will discuss system structural properties we believe to be important, and tools we developed to analyze these properties for large software systems.

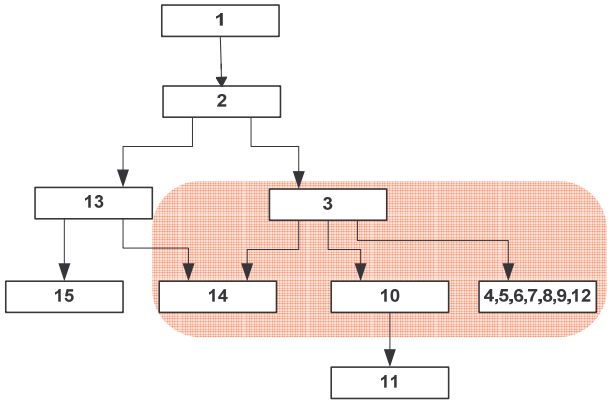
2.1. Basic Models

First, we present some hypothetical structural problems and desirable structures. In this section, we present the charts that we use to analyze a software project; these are all very small examples for illustration. In a later section, we will use the same charts for large software projects.

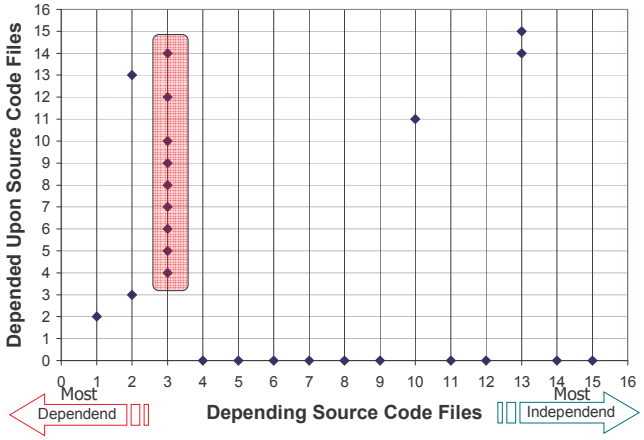
2.1.1 Problem: Large Fan-out

Depending on scores of other files - large fan-out - may indicate a lack of cohesion – the file is taking responsibilities for too many, perhaps only loosely related, tasks and needs the services of many other files to manage that.

The chart in Figure 2.1, below demonstrates the case of large fan-out along with a topologically sorted diagram of the files. Any file depends directly only on the files indexed by points in the diagram vertically above it.



Structure Chart – Large Fan-out



After topological sort

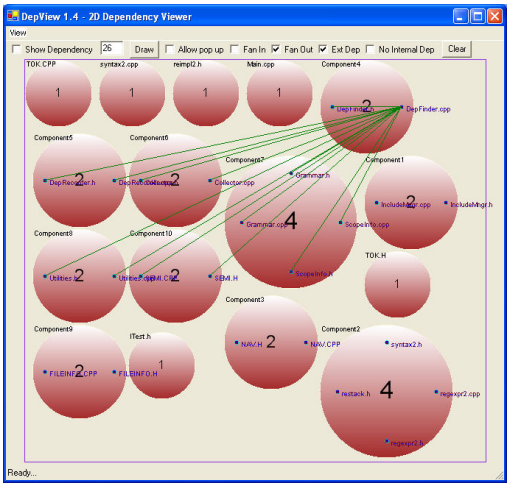
Figure 2.1 – Basic examples – large fan-out

The topological sort order of the files is [15-14-13-12-11-10-9-8-7-6-5-4-3-2-1]. Each of these numbered files depend only on files with lower numbers, but do not necessarily depend on every file with lower number.

We see in the chart above, file number 3 depends on nine other files directly, and depends on one other file indirectly, not shown by the chart for ease of interpretation. In the later phases of development, if a change is requested in file 3, it is hard to retain information about what each of the nine files are used for. Even the developer of that file can easily forget the details of those 9 files. Another drawback is, if a change occurs in any of those nine files, file 3 has to re-tested to make sure those changes do not break its compilation or operation. When operation of code in file

3 is defective, the problem can be in file 3 or in any of the 9 files, it depends on. Therefore, fan-out should be limited, excessive fan-out reduces reuse; makes code harder to understand and modification more difficult.

Below in Figure 2.2, we see a dependency view of DepAnal. The picture only shows the fan-out of dependency of DepFinder.cpp, as we see it depends on many other files. This will make testing of this file harder. Change in depended files likely force DepFinder.cpp to change, this will require frequent testing and reduces the manageability of the file.



On the left, we see that DepFinder.cpp uses services of many other source files.

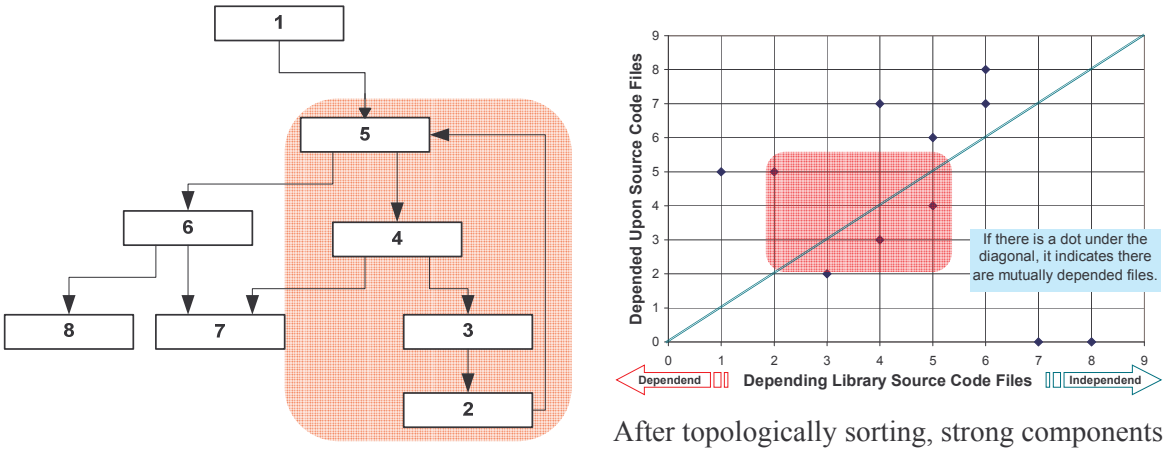
An error fix or update of DepFinder.cpp will require understanding of all depended files. This reduces flexibility of accepting new changes.

Figure 2.2 – Example of excessive fan-out, dependency picture of DepAnal

If someone wants to use DepFinder.cpp, he needs to add scores of other files used by DepFinder.cpp. Consequently, reusability of DepFinder reduces. This is an undesirable property and increasing the size of the project.

2.1.2 Problem: Large Strong Components

A strong component, within a dependency graph, is a set of mutually dependent files. When topologically sorted, a strong component will have dependency relations that appear below the diagonal of the dependency matrix, as shown in Figure 2.3.



Structure Chart – Strong component

After topologically sorting, strong components are expanded

Figure 2.3 – Basic examples – strong component

The existence of strong components, in a set of files, makes it impossible to carry out an ideal testing process on the set, as discussed below.

Ideal testing process:

- Test those files with no dependencies, and then test all files depending only on files already tested.

For testing, a strong component must be treated as a unit, because they cannot be put into a total order. The larger a strong component becomes, the more difficult it is to adequately test.

Change management becomes tougher, due to consequential changes that may occur when we fix latent errors or performance problems. The chart shows circular dependency among the files 5, 4, 3 and 2. There is no topological sort possible among these files; this will cause difficulty picking a file to start testing. In addition, this makes code hard to understand and any change of a file in a strong component requires all the files in the strong component to be re-tested. If the size of a strong component increases, it makes testing, maintaining and adding new features much harder to accomplish. This is a serious sign of structural problems.

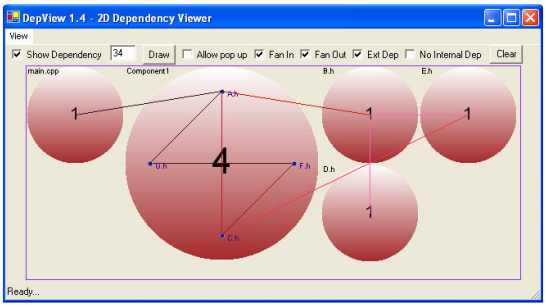


Figure 2.4 – Example of strong component, a strong component with four files

This is the topological sort of the files [8-7-6-5-4-3-2-1]. The order given is the best we can achieve for testing. Numbered files depend only on files with lower number, but do not necessarily depend on all files with lower number. Files 2, 3, 4, and 5 cannot be ordered.

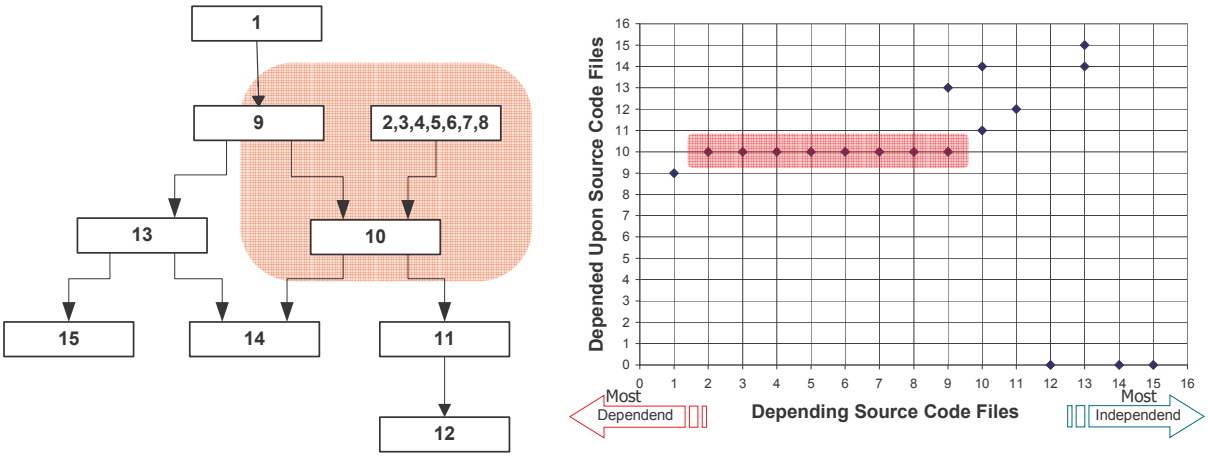
From the re-use perspective, a developer would not want to include a lot of files in his project just for a couple of features; since it will complicate testing, increase the size of the project, and reduce comprehensibly. As a result, large strong components reduce software reuse.

2.1.3 Problem: Large Fan-In

High Fan-in is not inherently bad. It implies significant reuse, which is good. However, poor quality of a widely used file will be a problem.

High fan-in coupled with low quality creates a high probability for consequential change. By consequential change we mean a change induced in a depending file due to a change in the depended upon file

The chart in Figure 2.5, below demonstrates the case of a large fan-in along with its topologically sorted diagram of the files. Any file is used only by the files indexed by points in the horizontal line passing through the used file on the ordinate.



Structure Chart – Large Fan-in

After topological sort

Figure 2.5 – Basic examples – large fan-in

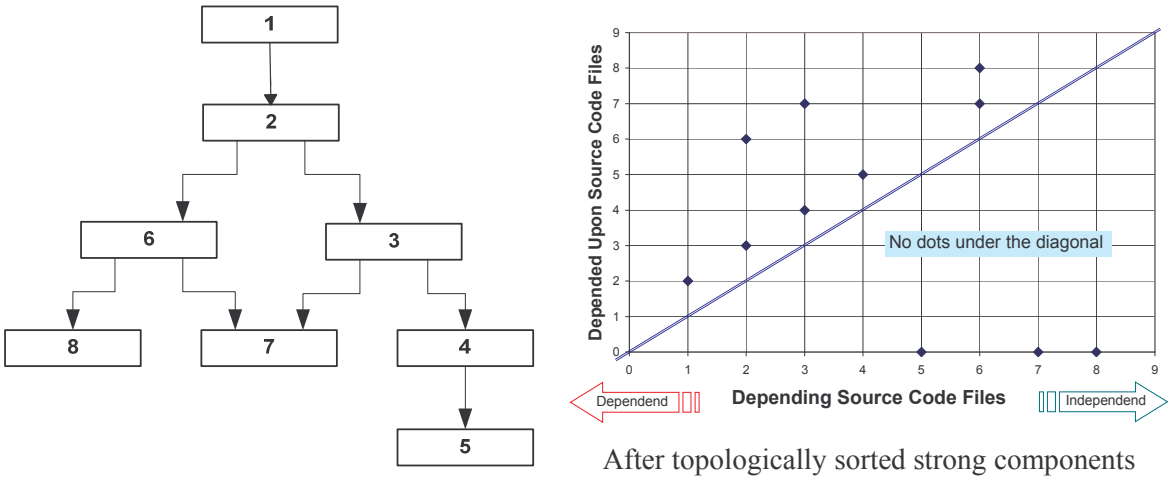
File 10 is used by 8 files, this shows file 10 is highly used, this is a good thing, however if a change occurs in file 10, all the depending files have to be re-tested to make sure that the introduced change does not break their functionality. Sometimes a change can cause a chain of secondary changes to depending files. Excessive fan-in is risky as in the case of excessive fan-out.

This is the topological sort of the files [15-14-13-12-11-10-9-8-7-6-5-4-3-2-1]. Numbered files depend only on files with lower numbers, but do not necessarily depend on every file with lower number.

2.1.4 Desirable Dependency Structure

Each component (file) depends only on its close neighbors. All files have low fan-in and fan-out. There is no call back to upper level components, or deep call forward.

Figure 2.6, below demonstrates the desirable dependency structure on the left along with its topologically sorted dependency diagram of the files on the right.



Structure Chart

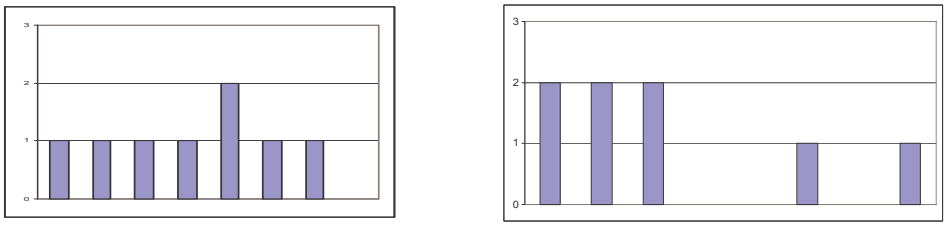
After topologically sorted strong components expanded

Figure 2.6 – Basic examples – desirable dependency structure

With this structure, testing can be organized in a straightforward way – test all files that depend on no others; then test files that depend only already tested files. There is no circular dependency, and each file depends only a couple of files, which enables developer to comprehend, to reuse, and to test the project easily. Moreover, a new change can be easily accommodated without requiring heavy testing or without causing a chain of changes.

Ideal structure has cohesive components with no mutual coupling. This is the topological sort of the files [8-7-6-5-4-3-2-1]. Numbered files depend only on files with lower numbers, but do not necessarily depend on every file with lower number.

Figure 2.7 shows the fan-in and fan-out histogram of basic example above in Figure 2.6.



There are no large fan in

There are no large fan outs

Figure 2.7 – Sample desirable fun-in and fan-out sizes

The histogram in Figure 2.7 indicates that there are no scores of fan-in or fan-out, which is desirable from the perspective of good dependency structure.

2.2. Dependency Analysis Tool, DepAnal

In Chapter 3, we embark on a study of the structure of large systems, focusing on Mozilla, Version 1.4.1. Before doing that, however, we will illustrate the structural ideas we have been discussing in this chapter, by examining our own DepAnal tool – a system considerably simpler than Mozilla - to help us interpret some of the ideas presented earlier in this chapter. To do that, we discuss several aspects of its code structure and its design.

We analyzed our static type dependency analyzer, DepAnal, in which we know every line of the code in detail. We would like to find out whether our analysis techniques are disclosing the same level information. Moreover, can we get insight without reading source code? Shaded area in Figure 2.8 shows the role of DepAnal in our analysis technique.

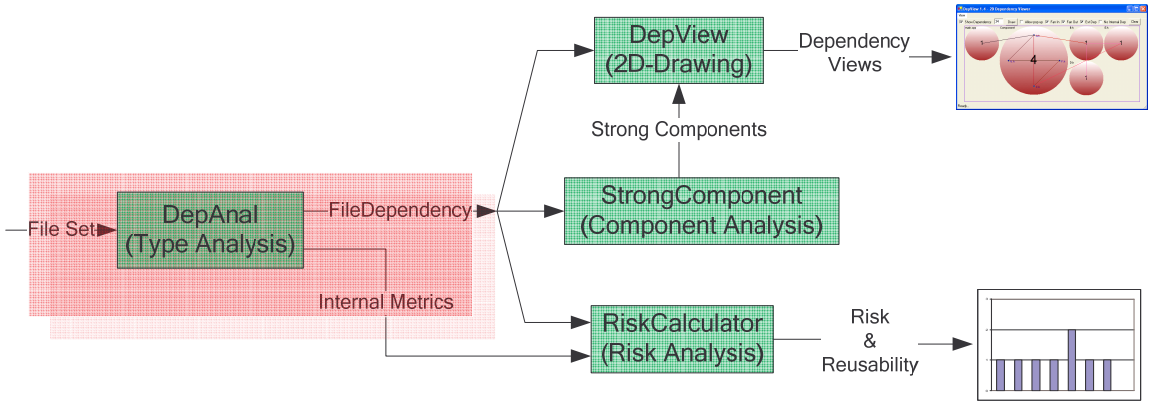


Figure 2.8 – Analyzing DepAnal itself.

We get insight both of mutual dependencies obtained from the dependency graph as strong components, and on individual files. DepAnal has 30 source files, 14 files out of 30 are reused files. We know our project well, which files are complex and hard to modify. However, we were not sure about the order of high risk files, the size of the strong components and their interaction with other files: The results we obtained, shown below in Figure 2.9, are encouraging. It

illustrates the applicability and helpfulness of the approach even for relatively small size software projects.

Topological sort of the files	
<-Most Independent	IncludeMngr.h
ITest.h	Grammar.cpp
regexpr2.h	Grammar.h
regexpr2.cpp	ScopeInfo.cpp
restack.h	ScopeInfo.h
syntax2.h	Collector.cpp
FILEINFO.CPP	Collector.h
FILEINFO.H	DepRecorder.cpp
NAV.CPP	DepRecorder.h
NAV.H	DepFinder.cpp
TOK.H	DepFinder.h
SEMI.CPP	Main.cpp
SEMI.H	reimpl2.h
Utilities.cpp	syntax2.cpp
Utilities.h	TOK.CPP
IncludeMngr.cpp	Most Dependent ->

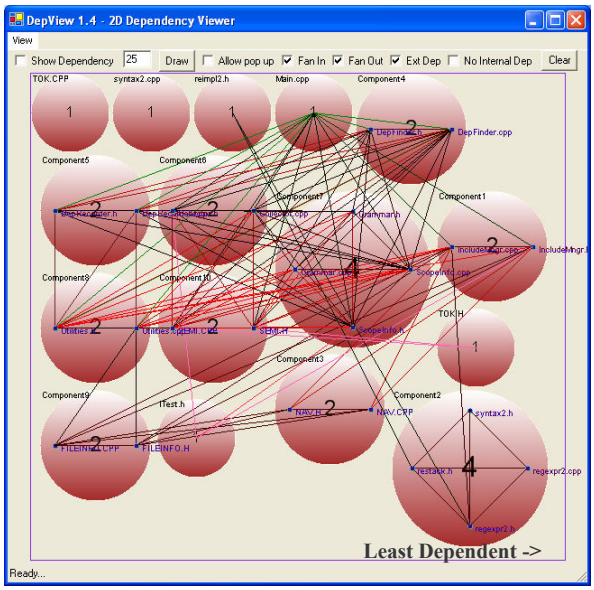


Figure 2.9 – DepAnal Ver 1.7.a's internal dependency structure. Consists of 30 files

At Figure 2.9, we see DepAnal's file level dependency structure, before seeing this picture, generated by DepView, we were not aware that DepAnal has a strong component with size of 4 files, which are files developed specifically for DepAnal. For this small size project, the largest strong component carries more than 25% of the developed source files in the project. This view demonstrates a potential problem without using our developer knowledge of this project. Even knowing our code well, we were not aware of the existence of this component and this dense interaction between files.

Figure 2.10, below shows each dependency between two files, which are obtained after expansion of topologically sorted strong components. The closer the dots to the diagonal is the

better the structure, indicating local communication, no deep call forward and no deep call backward.

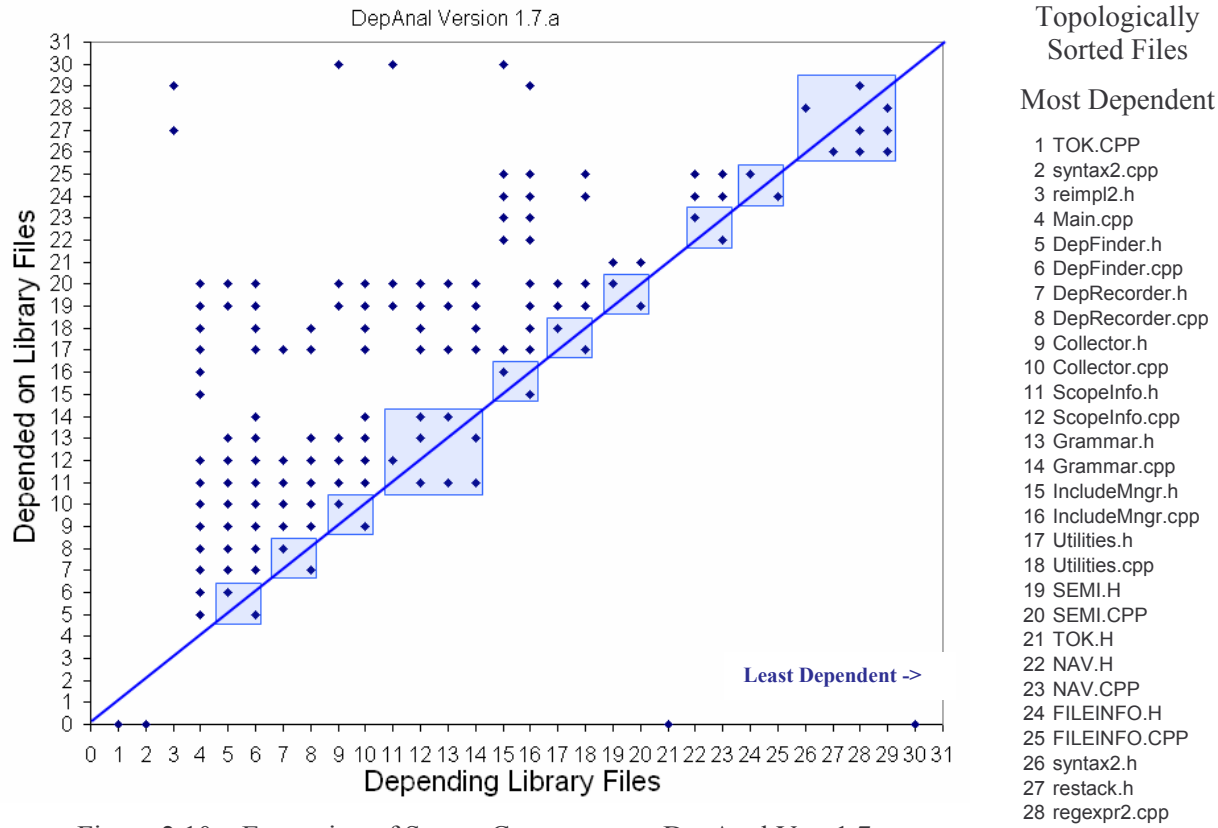
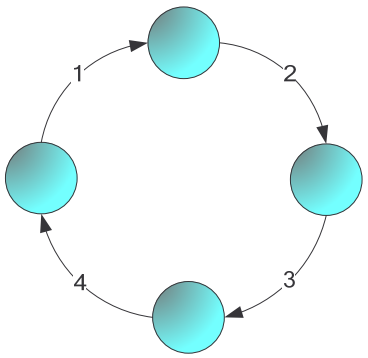


Figure 2.10 – Expansion of Strong Components - DepAnal Ver. 1.7.a

At Figure 2.10 above, we see strong components expanded into their individual files, each dot represents a dependency relationship between two files. For any dot, the file vertically below it (x) depends upon the file horizontally to its left, on the ordinate (y). Any dot under the diagonal indicates the existence of a strong component. Each rectangle represents the strong component as shown in Figure 2.9.

One other important information Figure 2.10 reveals is the number of dots inside the rectangles. The more dots (dependency relationship) the higher the risk. We show in section 4.1.8, that as the dependency density increases, in a set of mutually dependent files, the system gets closer to, or reaches, the point of unending changes.



Strong component with 4 files

We know the largest strong component in DepAnal consists of 4 files. In order that 4 files belongs to one strong component, we need at least 4 dependency lines among them. However, there can be other dependencies between the members of a strong component, as in the case of DepAnal's strong component, which has 8 dependency lines as shown by the block in Figure 2.10. There are twofold more (4-extra) dependencies between members of the files. This reduces the flexibility of the files for change, as any change may induce other changes to occur.

Without the help of tools, it would be impossible to analyze large quantities of source files in a timely manner and obtain accurate results manually. In order to overcome these limitations we developed several tools to analyze source code and visualize the results to enhance our insights about the project under examination.

Figure 2.11 shows fan-in histogram of DepAnal. Some of the files have high reuse, such as Utilities.h/.cpp and Semi.h/.cpp. These files are providing common services needed by most of the files in the system. As developers, we expect these files to have good implementation quality, since change in those files requires testing of many files and has potential to cause cascading of changes in depending files.

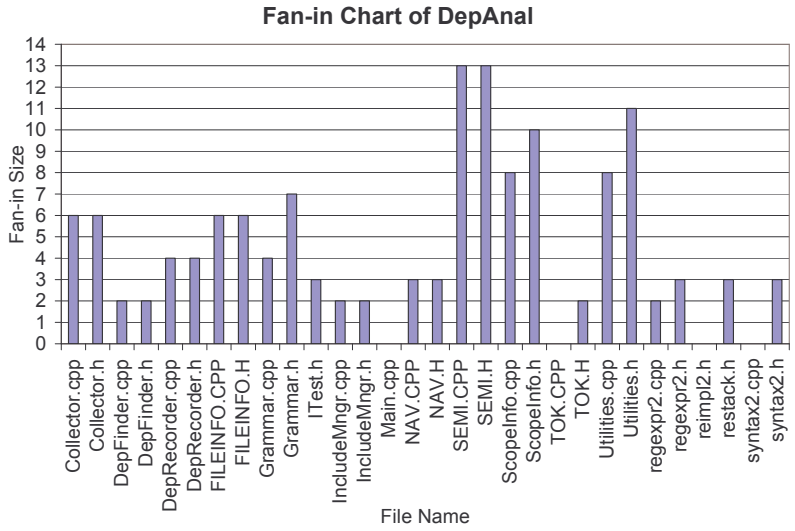


Figure 2.11 – Fan-in Chart of DepAnal Ver. 1.7.a

In Figure 2.12, we see the fan-out chart of DepAnal, some of the files use services of many files in order to accomplish their jobs. For example, Main.cpp uses too many files to accomplish its assigned task; it would be difficult to understand the reason those 14 files are needed. Some of them have reasonable fan-out size. As stated earlier, large fan-out reduces comprehensibility of a file. Depending on many files may cause frequent change in the file with large fan-out due to changes in depending files. Additionally, in order to make sure no adverse effect transpire due to changes in depended files, frequent testing is unavoidable.

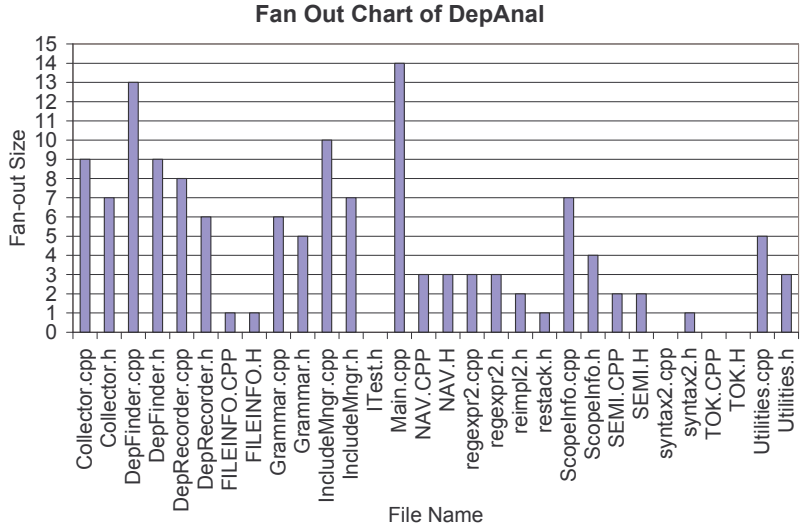


Figure 2.12 – Fan-out Chart of DepAnal Ver. 1.7.a

2.3. Analysis Applications

We have analyzed several open source projects. In order to accomplish this we provided analysis tools capable of analyzing very large systems and characterizing their behavior so as to make visible the web of dependencies that every large system possesses. These dependency relationships have always been visible in the small – in local areas around one specific part of the code base. However, dependencies are invisible in the large – far too complex for a human to understand without tools. Our goal is the construction of a robust tool set that makes these relationships visible in clear descriptive ways, with analysis code that is robust, able to handle very large systems, and with a friendly enough user interface that researchers outside our group can use them. We have used these tools to help us build and interpret risk models, file ranking, alpha estimation, and reusability index. Furthermore, the analysis does not depend on semantic understanding of large parts of the system. For the large systems we study, a single developer can only understand a small part of the complete system.

Some of the major tools that we developed for analysis are shown in Table 2.1, below.

Tool Name	**Short Description**
DepAnal	Dependency Analyzer, Finds dependencies between C/C++ source code files based on static type analysis.
DepAnalUI	This is a user interface for all the developed tools to integrate the management.
StrongComponent	Extracts mutual dependency groups and their topological sorts by using the dependency information generated by DepAnal.
Analyzer	Analyses several internal metrics of software source file, such as each function's cyclomatic complexity, total line count.
DepView	Dependency Viewer, provides two dimensional (2D) interactive views of dependency relationship file and strong component level.

MatrixMaker	Creates importance and test risk matrix by using dependency and metric information.
RiskCalculator	Calculates the risk rate of files by using matrices generated by MatrixMaker.
ChangeLogger	Logs each change record with brief info, cause, date, dependency and metric information.

Table 2.1 – Selected developed tools for analysis

This tool set enables a software manager or developer to constantly monitor structural and internal qualities of files. Unwanted dependencies can be spotted shortly after they occur, and the software manager can request corrective action.

2.3.1 DepAnal

Dependency Analyzer, DepAnal is an automated file level dependency analyzer. It finds dependencies among C/C++ source code files based on type, global function and global variable invocations and declarations. It ignores include-based dependencies, as developers are occasionally careless about the files they include. Unnecessary dependencies affect build efficiency but do not affect design or compilation breakage due to change. All of generated data are presented as direct dependencies. Hence, the transitive closures of the dependency graph are not shown. Analysis is carried out this way because, for large systems, the transitive closure becomes very dense and hard to interpret.

There is one main application file, DepAnal.exe, which finds dependencies between the files by scanning through source codes in files. A window based user interface we generated to manage DepAnal and accompanying analysis application, called Dependency Manager. Dependency Manager organizes the order of the applications' invocation. Some of the processes it spawns are Analyzer.exe, which is generating metric information, about each source file, and

StrongCompAnal, which uses DepAnal's output as input to find strong components (A group of mutually depended files targeting a particular goal).

The plots and graphics, shown above were derived from a static dependency analysis using DepAnal. The analyzer uses a production-based grammar analysis, greatly simplified from the grammar required to specify the entire C++ language. The results of the analysis are used to build a dependency graph for the analysis set, which is then processed to find strong components, and prepare a topological sort, needed for some of our visualizations. Additionally, the output also used during Product Risk analysis Reusability Index calculation.

The analysis finds dependencies based on the following model: file A depends on file B if;

1. A creates and/or uses an instance of a type declared or defined in B
2. A is derived from a type declared or defined in B (inheritance)
3. A is using the value of a global variable declared and/or defined in B
4. A defines a non-constant global variable modified by B
5. A uses a global function declared or defined in B
6. A declares a type or global function defined in B
7. A defines a type or global function declared in B
8. A uses a template parameter declared in B

These rules intentionally do not acknowledge dependency of a base type on its derived types even though it is possible that a derived type modifies protected data members of the base. Doing so, we believe, would identify potentially many false-alarm dependencies in well designed systems. It would be interesting to compare analyses of a major system with this assumption and with a model in which the base is declared to depend on all derived types if it provides protected data.

2.3.1.1 Architectural View of Dependency Analyzer (DepAnal)

DepAnal's goal is to find dependencies between source code files based on static type analysis. Dependency relationships between the files are determined by the model described

above, as in [8][25][26][27] DepAnal makes three passes over each file in the project, as shown in Figure 2.13.

There are three passes over the source code set. We can summarize the job of each passes as following.

- First pass prepares source files for analysis,
- Second pass collects user-defined types, global functions and global variables,
- Third pass finds dependencies between source files by finding invocations of items collected in second pass.

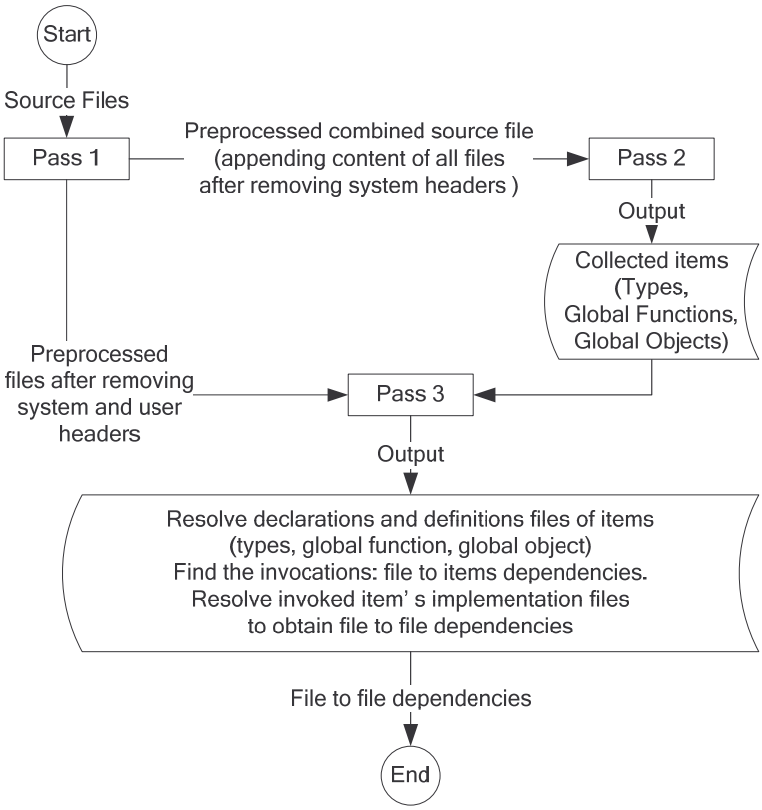


Figure 2.13 – DepAnal data flow diagram

End outputs shown in Figure 2.13 are dependency information between the source files together with each files internal metrics.

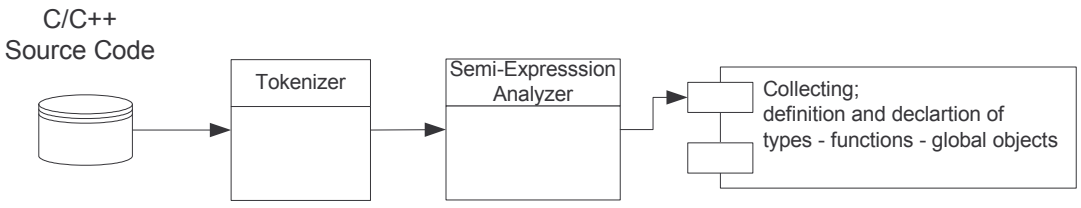


Figure 2.14 – Collecting data from source code

Figure 2.14 illustrates the process of collection of user-defined types, global functions and global objects. We use the same process for collecting invocations

The internal architecture: The core task is to assemble useful information from collected data in a representation that gives easily comprehended views of the current state of an analyzed project. DepAnal's outputs are all text based. Charts showing the system's state of health are prepared using Microsoft Excel.

The goal is to build a tool that can be used to constantly monitor evolution of the state of large software systems and provide guidance about where detailed quality analysis and re-factoring are needed.

DepAnal tool does not identify unused code. Its parser is not a full implementation of a C++ recognizer, but rather an ad-hoc implementation of the rules described in section 2.3. We have checked manually its results on modest size projects and run it many times on our own code, as it evolved, and believe that the results are accurate, within the limitations described in this paragraph.

We also developed two adjunct tools that extract additional information from the project.

2.3.2 Strong Component Analyzer:

StrongComp[18]. builds a dependency graph from the data provided by DepAnal and analyzes its strong components, that is, sets of files that are mutually dependent as defined by

[18] The first implementation of this tool was implemented by Srinivas Neerudu, now with Microsoft in Redmond, Washington.

Lakos [58]. It then performs a topological sort of the strong components to show an ordered flow based on dependency. Finally, it expands the strong components, within the sorted component order, to arrive at a representation of all the files as well ordered as is possible when there are mutual dependencies. This provides a candidate for testing order of the files that attempts to minimize re-testing when latent defects are found and repaired.

2.3.3 Size and Complexity Analyzer:

Analyzer counts the number of lines of source code in each function and analyzes each function's cyclomatic complexity, measured by the number of regions enclosed by the control flow graph of the function. Anal also evaluates the total line count and sum of the complexities of all of the functions in each file.

2.3.4 DepView

All the results, obtained from running DepAnal on software projects are represented as text, and interpreting these text files is almost as hard as reading source code. We felt the need of another way of representation, which would disclose qualitative information about the system in an easily understandable fashion. We developed the 2D dependency viewer, DepView, to obtain comprehensible views of large software systems.

Dependency Viewer, DepView helps us to see dependencies between files and strong components, it gives another insight about how densely parts of the component interconnect, using its 2D graphical interface. DepView provides mainly two kinds of view; one is component wise dependencies, which gives the big picture of project; another is file level dependencies, which is much denser. However, with file wise we can focus on a file and visually see dependencies between it and other files.

Each circle represents a strong component, and at the center of circle, there is a number, which indicates how many files are parts of this strong component, Size of the bubble is

proportional to the number of files in strong component. We assume that a file itself is a single component with size of one.

Figure 2.15 illustrates how well the project is packaged into modules.

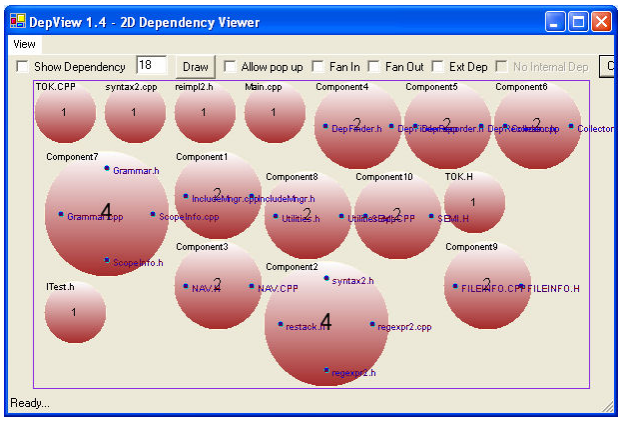


Figure 2.15 – DepView of DepAnal, components and files

On the left, it shows DepAnal's files and strong components, it does not show dependency information in this view. Even though it does not show dependency lines, it gives insights about strong components, as size of bubble proportional to number of files in strong component. The larger the strong component, the harder to adapt to a change, the harder to test.

Figure 2.16 illustrates dependency relationship of component #6. It can be easily seen how many files are using the services of the component, and how many files are being used by the component to accomplish its tasks.

On the right it shows dependencies of Component 6, which consists of two files, Collector.cpp/.h. This is useful information to get more detail about interaction among source files.

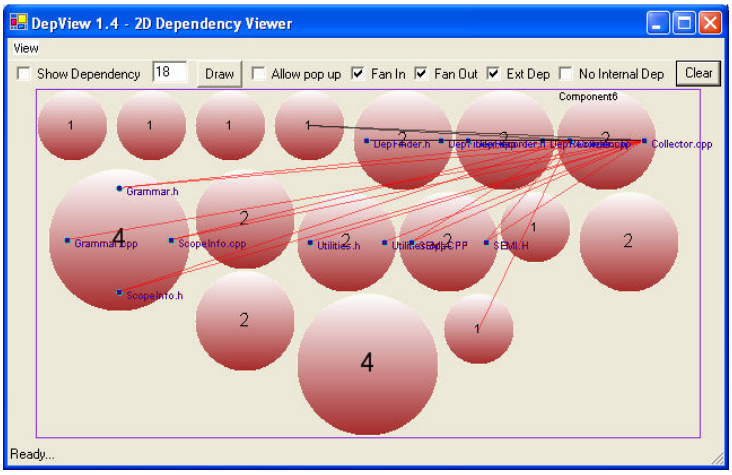


Figure 2.16 –DepView, dependencies of component 6

In addition, generating frequent DepView images enable us to monitor the evolution of dependencies among source files and monitor growth of strong components. This information can be used to help avoid excessive dependencies.

As summary:

- Visualize the static structure in one picture
- Visualize the web of dependencies
- Realize mutually depended files and size of the mutually depended files
- Reusability of the files in the system (if a file is a member of a large strong component or depend, on transitively many files, it is not a good candidate for reuse)
- Whom a file provides services
- From whom a file gets services

2.3.5 Dependency Analyzer User Interface

Dependency Analyzer is console application, which needs "settings.txt" to acquire information about the project to be analyzed. To make use of DepAnal more user-friendly graphical user interface developed which helps managing settings.txt and execution order of developed analysis tools. There are three groups of information, titled Project Settings, Environmental Settings and Applications as shown in Figure 2.17.

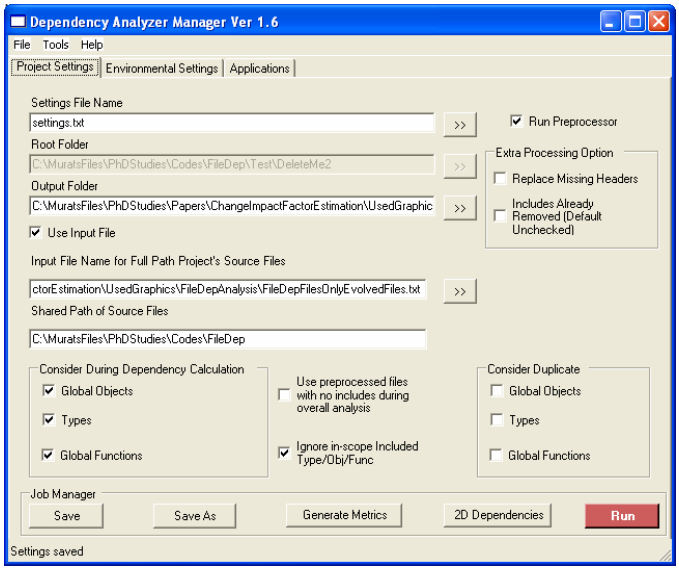


Figure 2.17 – Settings for project to be analyzed and dependency options

Project settings contain the information regarding analysis project at hand, such as source directory, output directory, which dependency types are to be considered. Environmental settings contain additional preprocessor options specific to analysis project at hand, include and library path information. Applications are the developed tools for the analysis.

2.3.6 Change Logger

Change logger is used to collect detailed change, dependency and metric information in a database. All the information is dated to monitor evolution of change impact factor (details are in Chapter 4 and Chapter 5). Figure 2.18 shows the screenshot of Change Logger application, which records occurred change information together with reason of the change, if it is a consequential.

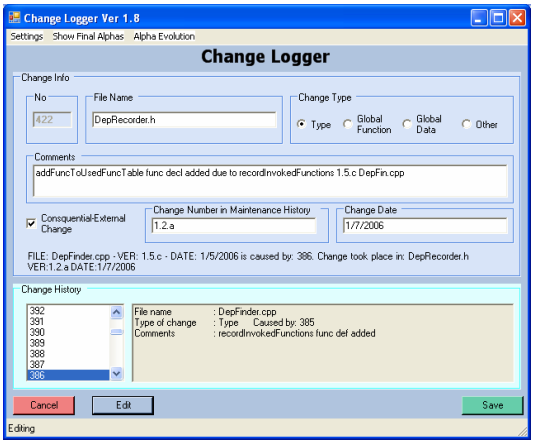


Figure 2.18 – Change Logger, records change information for change-impact-factor (CIF) estimations

Each change carries the following details: File name where change occurred, what is the change about, what other change drive this change, change date, change number, type of change. Additionally, while recording each change entire project is analyzed by DepAnal, current dependency and metric information also recorded into database. It also calculates the alpha values within given any two dates.

2.3.7 Matrix Maker

Matrix Maker is a C# application using DepAnal's output to create matrixes for risk analysis in Chapter 4. It generates importance, testability matrixes and their corresponding results arrays, accepts dependency and internal metric information to create matrixes. The outputs of Matrix Maker are used by MatLab Risk Calculator application to find out Risk values of each source files.

Figure 2.19 shows the screenshot of Matrix Maker application, which generates text based matrix files to be used during product risk calculation

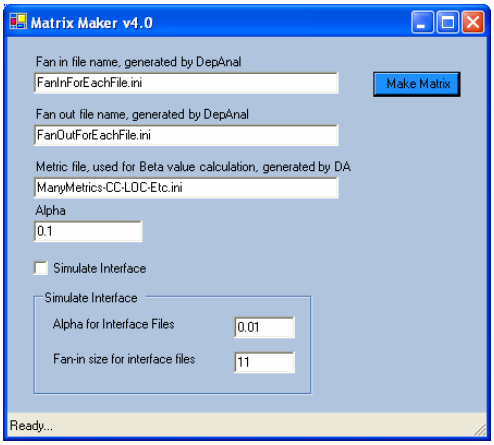


Figure 2.19 – Matrix Maker – creates matrix for risk analysis

To simulate the effect of interface insertions, it provides an option to use special alpha value for the files, which are used by certain number of files. This feature is used mainly for simulating constructive change covered in Chapter 6.

Moreover, Table 2.2 shows other small size handy applications for analysis.

Averager	Finds average change occurred to files based on metric sizes.
CleanFiles	Removes non C/C++ files from list of files to be analyzed.
DirWalker	Walks recursively directories and grabs all source files

FindExistingFiles	Going back from Mozilla 1.4.1 libraries to previous version of Mozilla to collect if the same file used in those versions.
MergeMetricsWithHistory	It merges DepAnal generated metrics with change history by release.
RemoveDupFiles	If same files exist in different folder it identifies and prevent same file to be processed by DepAnal several times

Table 2.2 – Helper tools for analysis

2.4. Summary

Without a detailed knowledge of the entire code base useful information about significant problems can be identified. Screening static structure provides both quantitative and qualitative information regarding structural problems, showing how pieces are interconnected with each other.

In order to get insight about a software project, assistance of analysis tools is a great help. Especially, in the case of large software. Analysis tools provide quick, accurate information directly from source code, which carries always up-to-date information. Therefore, we have developed and applied these applications. Moreover, we analyzed the entire Windows build of the Mozilla project, version 1.4.1, released in October 2003. There are 6193 files in that build, which means that this build is indeed a large project, and we show in our analyses, in Chapter 3, is subject to many serious structural problems.

Chapter 3

Empirical Study

In this chapter, we focus on analysis based on static type dependency between source code files. All of our data are presented as direct dependencies. That is, we do not show the transitive closures of the dependency graph. Analysis is carried out this way because, for large systems, the transitive closure becomes very dense and hard to interpret. Note that we do account for these transitive dependency relationships in our Product Risk Model, discussed in Chapter 4. Our primary interest is evaluating the quality of a system's structure and implications of the structure for project management, maintenance, and testing. We present and interpret results of an empirical study of C/C++ projects and how to asses the quality of software systems from analysis of their source code.

3.1. Empirical Study of the Open-Source Mozilla Project

All of our findings are based on a static dependency model outlined in the Chapter 1 and Chapter 2. We present several different views of the dependency data and draw some conclusions about what such data can disclose concerning a project's implementation.

Mozilla is a very large project developing browser tools for many different platforms. It consists of many thousands of files, and so is a typical example of the large systems we wish to explore [12]. The Windows-based version of this software was chosen for analysis, as we are familiar with that as a programming environment and have all the tools to execute the various builds required for this study. We have examined the entire Windows build as well as several constituent libraries and adjunct tools, 6193 files in total, generating builds for each before proceeding with our analysis.

The analysis results are presented for several data sets, in five views:

1. Fan-in: the number of files that depend on a file, for each file in the analysis set, and related fan-in density histogram.
2. Fan-out: the number of files that a file depends on, for each file in the analysis set and related fan-out density histogram.
3. Strong components: groups of files that are all mutually dependent and related strong component density histogram.
4. Topological sort of the strong components.
5. Expansion of all strong components within the sorted data.

We examine each of these views and interpret their data with respect to measures of project implementation strengths and weaknesses they reveal. Type dependency fan-in and fan-out have been discussed before [13][14][15] with results presented similar to those shown here. We focus explicitly on the structural aspect of program implementation at the file level.

3.1.1 Mozilla Data Collection

We downloaded version 1.4.1 of the Mozilla Win32 configuration [16] [12]. This included the entire build, which makes many executables and libraries. We were able to build all the libraries and executables in about a week's effort, using the information provided on

www.mozilla.org. This involved making a few recommended changes to makefiles[19], setting environment variables, and settings in for the Visual Studio C++ compiler, used for all the builds for this empirical study.

Note that our analysis pertains only to the Mozilla source code, but we wanted to ensure that we analyzed exactly those files used to create individual executables and libraries. It took some time to understand the required directory structure, make modifications to that to suit our analysis, and then make trial builds, but the process went surprisingly smoothly.

We built some simple parsers to find all the files included in a specific build, based on compiler output. This included all common code and header files. The statistics for this process are shown in Table 3.1.

Number of executables:	94
Number of dynamic link libraries:	111
Number of static libraries:	303
Number of source files for Win32, v 1.4.1	6193

Table 3.1 – Summary of generated outputs and files from Mozilla built

The information provided on the Mozilla web site was very well prepared, easy to digest, considering the size of this large project, and straightforward to use. We chose this project because of the quality of its tools and the fact that it has a very large code base.

The analysis tools developed for this research were able to digest the entire code base of 6193 files and perform all the analyzes in approximately 4 hours after configuring the settings on a PC with 1 Gigabyte of random access memory, running Windows XP Professional, with Pentium IV Processor.

[19] A configuration file used by the make utility defining the location of source files, how they will be compiled and linked to create the executable program. www.mccabe.com/iq_research_iqgloss.htm

3.1.2 Fan-in Data Extracted from Mozilla GKGFX Library

Figure 3.1, below, shows fan-in for each of the files in the Mozilla GKGFX library. This plot analyzes all of the dependencies on each of the 598 source code files in the library from within the library. When we analyze the entire build, many of these fan-in numbers become larger.

Fan-in is the number of files that depend on a file. A file with large fan-in is desirable from the perspective that it demonstrates high reuse of the types defined in that file. For instance, we would expect to see high fan-in for some utility library routines. However, should that file have less than desirable quality attributes one would expect to see a high probability of change, not only for that file, but also for many of the large number of files that depend upon it [17].

Figure 3.1 and Figure 3.2 illustrate histogram of fan-in value and its corresponding density values respectively.

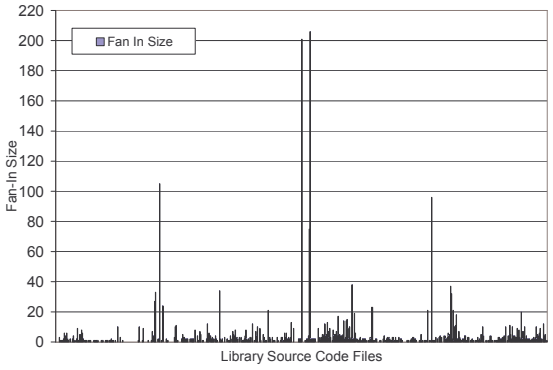


Figure 3.1 – Mozilla GKGFX Library Fan-in

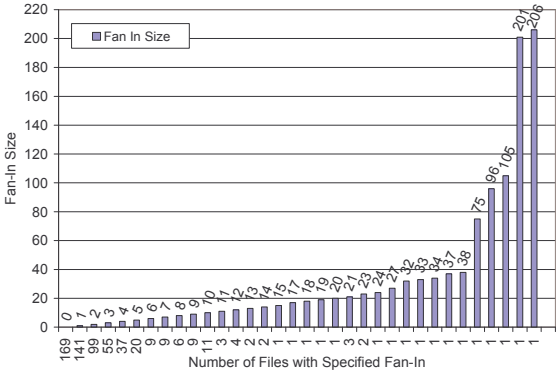


Figure 3.2 – Fan-in Histogram for GKGFX Library

There are scores of files, shown in Figure 3.1, with very large fan-in. All of these should be important targets for quality analysis, in order to effectively manage the change process during development. High fan-in coupled with low quality creates a high probability for consequential[20] change [18]. We have also looked at the wealth of change data provided by Mozilla's associated change data log to understand this process better.

[20] By consequential change we mean a change induced in a depending file due to a change in the depended upon file.

In Figure 3.2 we show fan-in density for the same library – simply a histogram for the data in Figure 3.1. This plot shows that some of the source code files have high fan-in, characteristic of a widely used library. A library with this profile should be given high priority for analysis by the test team and quality analysts.

We explored the relationship between high fan-in files and high risk[21] files. We analyzed Mozilla's GKGFX Library, and selected 100 files with highest risk value and 100 files with highest fan-in value. And matched these two groups, asking the questions, how many of the highest fan-in files are common with highest risk files. We observed that 45 out of 100 highest risk files are also in the highest fan-in files, and 5 out of top 10 highest risk files are again in the highest fan-in files. This showed us high fan-in files are likely to have high risk values.

3.1.3 Fan-out Data Extracted from the Mozilla GKGFX Library

Fan-out for the GKGFX library is shown in Figure 3.3, below. Fan-out is the number of files that a file depends on. A file with large fan-out may be symptomatic of a weak abstraction. We expect that a source file may carry out its assigned tasks with the aid of a few trusted delegates and perhaps a few references to commonly used utilities. However, depending on scores of other files may indicate a lack of cohesion – when the file is taking responsibilities for many, perhaps only loosely related, tasks and needs the services of many other files to manage that.

Figure 3.3 and Figure 3.4 illustrate histogram of fan-out values and its corresponding density values respectively.

[21] Risk is defined in Chapter 4

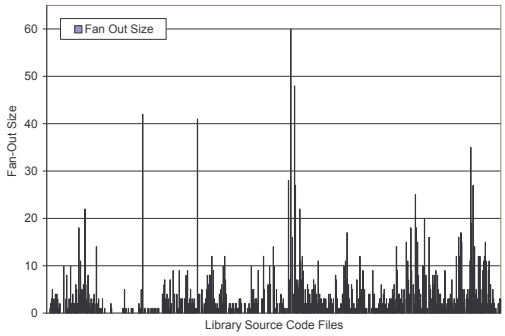


Figure 3.3 – Mozilla GKGFX Library Fan-out

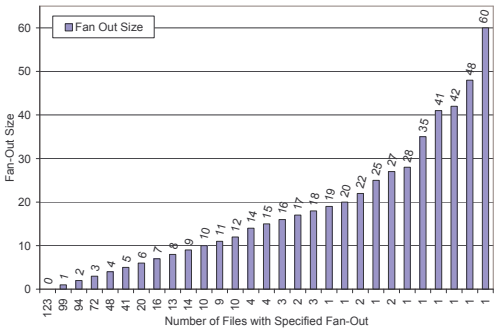


Figure 3.4 – Fan-out Histogram for GKGFX Library

Figure 3.4 shows a Fan-out histogram for the data in Figure 3.3. There are a significant number of files with large fan-out. If one follows the classic test model, testing code that only depends on already tested code, this profile suggests difficulty scheduling testing for this library. Automated test schedule planning tools can provide significant help for this, but, we show below that there may still be persistent problems creating a satisfactory test sequence for libraries with many high fan-out files

Also we explored the relationship between high fan-out files and high risk files. We analyzed Mozilla's GKGFX Library, and selected 100 files with highest risk value and 100 files with highest fan-out value. We matched these two groups, observing that 71 out of 100 highest risk files are also in the highest fan-out files, and 7 out of top 10 highest risk files are again in the highest fan-out files. This showed us high fan-out files are likely to have high risk values as similar to high fan-in. And only 22 files are common between highest fan-in and fan-out files out of 100.

3.1.4 Strong Components in the Mozilla GKGFX Library

A strong component of a libraries dependency graph is a set of source code files that are mutually dependent. Any given file from a strong component depends, either directly or indirectly[22], on every other file in the component. There can be no complete dependency ordering

[22] Type-based dependency is a transitive relationship. For reasons discussed earlier, we chose to show only direct dependencies.

within a strong component, so there is no way to prepare a classic testing schedule based on testing only code that depends on already tested code. Essentially the strong component must be treated as a unit. The larger strong components become, the more difficult it is to adequately test.

Figure 3.5 shows a strong component histogram for the GKGFX library. There are many strong components of modest size, and one huge component, consisting of 60 files. Circles are in topological sorted order; the order given is the best we can achieve for testing. Each file or strong component is drawn from upper left corner (most dependent) to bottom right corner (most independent). Above files depend only on files to their right or files below to them, but do not necessarily depend on every file on its right or below. To save space on the screen when we draw a small component after a large strong component we follow the following simple rule. If there is a space, we draw just next to large component starting first available space closer to top. If there is no space at its right, we start at top left most available space to draw our next component or file. Therefore, to find out whether a circle is above or below, just compare upper left corners of the bounding box containing the circle.

The dependency coupling that forms strong components may be due to the use of non-constant global data [19], to callbacks that provide notifications to a caller distant in the dependency tree, or to mutual dependencies on types defined across the strong component. Whatever the source, they indicate problems with testing and possibly with change management, due to consequential changes to fix latent errors or performance problems [18].

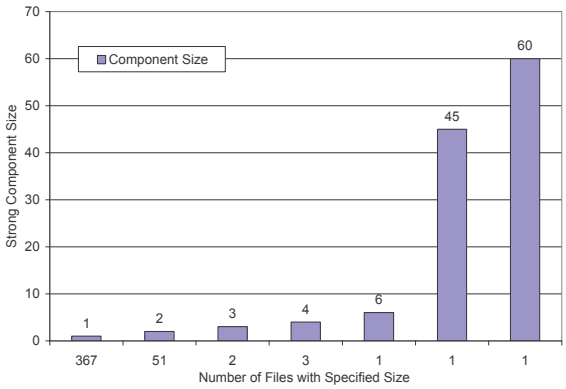


Figure 3.5 – Mozilla GKGFX Library Strong Components Histogram

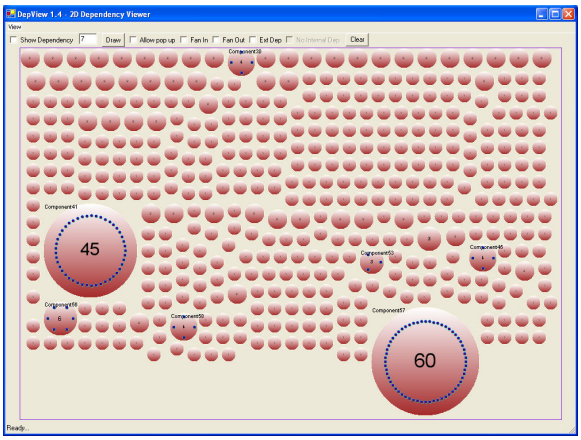
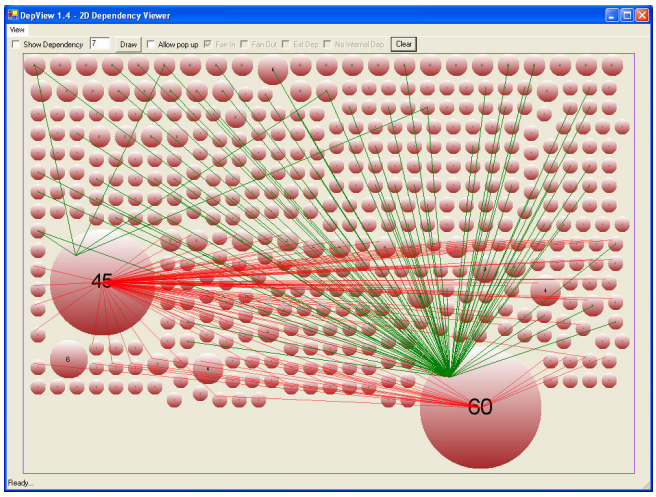


Figure 3.6 – Mozilla GKGFX Library Strong Components by DepView

Another issue that this plot illustrates is the lack of well defined modules. The dependency model we use for this analysis recognizes mutual dependencies between declaration and implementation of a type, global object, or global function. So we would expect, for non template-based source code, to see most files appearing in strong components of size two, or a few more perhaps, reflecting the design of a module with declarations of all types provided by the module in a header file and implementations in a corresponding implementation file, ideally of the same name. Here, we see that most of the files in this library do not fall into the classic module structure.

Figure 3.7 focuses in dependencies among strong component for the two of the largest components. If file dependencies were shown, we would see many dense lines than illustrated in Figure 3.7.



Each circle represents a strong component; number on the circle shows how many files are in that strong component. In the figure, the largest strong component consist of 60 files, lines from center of the circle show fan-outs and lines coming to the left corner of the circle show fan-ins to this component.

Figure 3.7 – Dependencies of only two of the largest strong components with other components.

If strong component size gets larger, it reduces the ability to adapt to new changes, since change may give rise to further, consequential, unexpected changes. This reduces the gain from change. If the component gets large enough, the software library may reach the point where change is no longer feasible, due to testing effort and consequential change. This is how an un-maintainable legacy system is born.

As we stated above, we only show external dependencies among components, besides this there may be a large number of dependencies between the members of a component.

We can draw the following conclusions about this library from Figure 3.8 and Figure 3.9. There are dense dependencies not only within the strong components, but also among the strong components. This is an indication of high coupling between many of the GKGFX files. High coupling naturally causes mutually dependent components, which are undesirable, because then, there is no effective file order for testing, as discussed in Chapter 2. Presence of the very large set of mutually dependent files, defined by this strong component, indicates difficulties in carrying out a classic testing program for this library. The figures above show that many files use many of other files to accomplish their tasks, this makes it difficult to understand their functionality. Consequently, it is harder to test and maintain those files than files accomplishing their task with

the help of few other files. Additionally, due to dense dependencies making changes and tracking the effect of those changes is difficult, therefore extensibility (new feature addition) degrades.

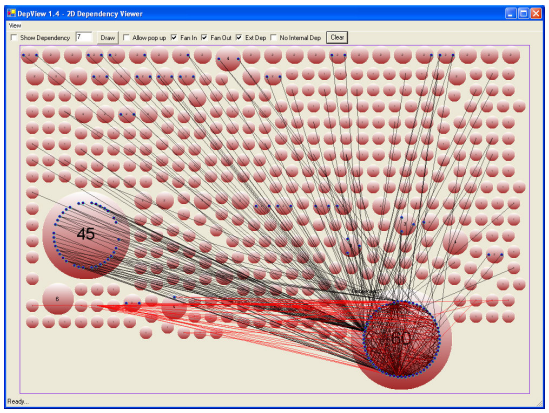


Figure 3.8 – Internal - External dependencies of Component #57 consist of 60 files.

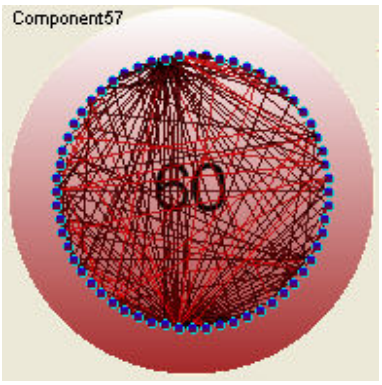


Figure 3.9 – Internal dependencies of Component #57 consist of 60 files.

If we focus on the internal dependencies between the members of strong component 57, we see that files are densely connected to each other as illustrated in Figure 3.8. If we add to the view the external fan-out dependencies of strong component as well, it will reveal that if any other depended-upon component changes; Component 57 also needs to be tested to make sure component 57 still performs according to its requirements. This view is shown in Figure 3.10.

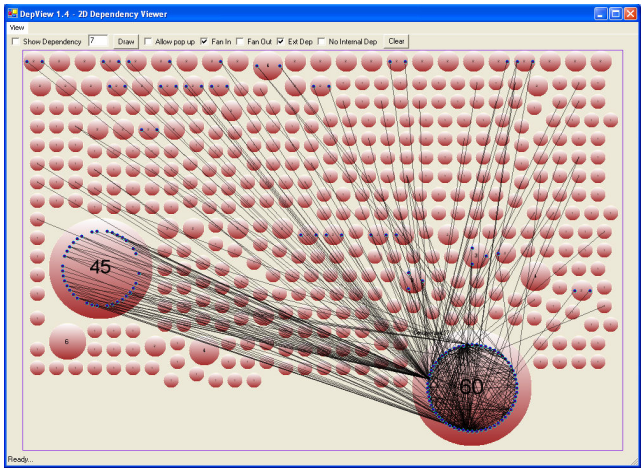
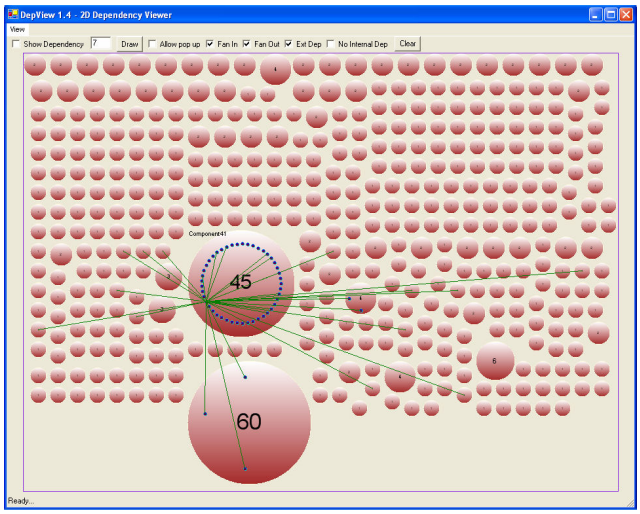


Figure 3.10 – External dependencies to Component 57

In GKGFX, looking at intermediate analysis results of DepAnal, we find there are only 15 files with template type definitions, but observe that in Figure 3.5 there are more than 300 components with only a single file. Some of these may be test drivers, but only 11 files have main functions, so a quality analysis would conclude that module definitions need attention for this library.

In Figure 3.11, we see Mozilla GKGFX library. As earlier stated, that dimension of the circle is proportional to size of the strong component. Component #41 is the second largest strong component with 45 source files. In the figure, it shows dependencies (Fan-out) of one of the files in that component. As we see, it depends on not only the files inside the associated strong component but also the files, which belong to other strong components. If any change occurs to depended files or depended components, this file needs to be tested to make sure, introduced change does not have an effect on the functionality of that file.



In this figure, smallest circles represent individual source files; others are strong components, which are sets of mutually dependent files.

The number at the center of each circle indicates the size of a strong component (number of files).

A line between circles shows dependency among files.

Two possible navigation levels exist in DepView. One is focusing on strong components; the other is focusing on individual files.

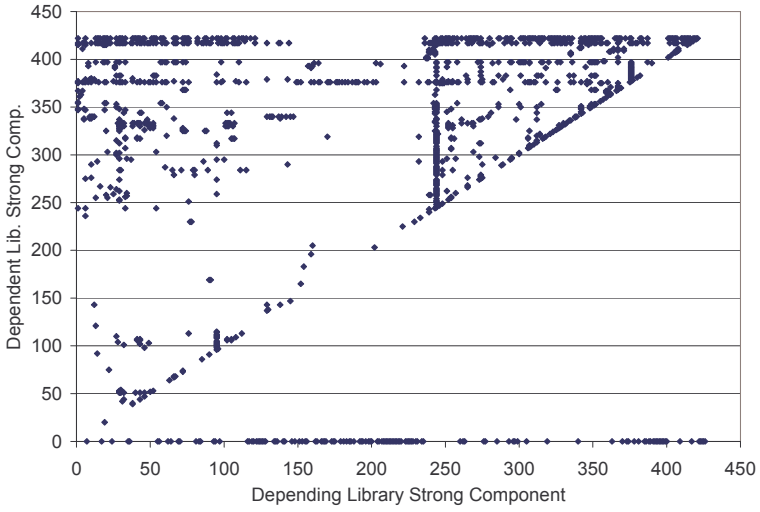
Figure 3.11 – A strong component member file' s fan-out to other files in GKGFX Library

In Figure 3.9, above we see the internal dependency relations within the component #57; our risk analysis, discussed in Chapter 4, shows this component is responsible for a major part of test risk for GKGFX.

Note that we have studied many of the libraries contained in the Mozilla project, as well as analyzed the entire Windows build of Mozilla, version 1.4.1. The results we obtain from all of the library parts and the whole project are very similar to those we find for GKGFX. We cite GKGFX often, because it is, in fact, representative of the project as a whole. (See Figure 3.14)

3.1.5 Topologically Sorted Dependencies for Mozilla's GKGFX Library

As mentioned earlier in this section, a classic testing strategy organizes source files into a topologically sorted dependency order, starting with files that depend on no other. We test those and continue by testing files depending only on previously tested files. This is not possible in the presence of strong components. However, we can condense all files in each of the strong components, and provide a topologically sorted view of the components, as in Figure 3.12, below. This provides us with a testing schedule that is as close as we can get to a classic test order.



To interpret this diagram, select any mark on the plot. For that mark, the strong component vertically below it, on the abscissa (x), depends upon the strong component horizontally to its left, on the ordinate.

Figure 3.12 – Topologically Sorted Strong Components before Expanding

Because all the elements of this diagram are strong components[23], their dependency graph can be topologically sorted – we have condensed away all of the cycles within that graph. This is apparent from the diagram, as all elements, except the ones lying on the abscissa[24], are all above the diagonal. Any file depends only on the files indexed by points in the diagram vertically above it. Therefore, all components depend only upon files to their right in this order[25].

The dense horizontal lines represent components with high fan-in. Each of the many distinct abscissa values depends on the corresponding single strong component on the ordinate. Similarly, a dense vertical line represents a strong component with high fan-out. The single strong component on the abscissa depends on the many unique corresponding strong components on the ordinate. Thus, structural problems for the library, as a whole, are evident in this diagram.

In Figure 3.13, we show the data from Figure 3.12, with all of the strong components expanded into their individual files. There, of course, will no longer be a topological order throughout, because individual files from a strong component cannot be put into sorted order, due to their circular dependency relationships.

Approximately half the files in this library (Figure 3.13) cannot be put into a classic testing sequence. This indicates a high probability of repeatedly testing a given file.

The top row shows a utility file, since many files depend on it. These kinds of files increase reuse, however files with high fan-in have low changeability, because they are being used by many others. If any change is made, the developer needs to make sure that the new change does not introduce any breakage to all these depending files, which increases testing effort.

High-fan out files use many other files to accomplish their defined tasks. This reduces the comprehensibility of the file and makes the job of the developer harder because interpretation and change now involve many files. Additionally it reduces reusability, since in order to reuse one file, developers have to include depended files into the project.

---

[23]. Many, of course, composed of only a single file.
[24]. By convention, we have plotted all components that depend on no other component on the abscissa.
[25]. This order is not unique. There may be many such orders, and some may be more useful than others. We are currently pursuing this idea, with interesting results we expect to publish subsequently.

Approximately half the files in this library cannot be put into a classic testing sequence. This indicates a high probability of repeatedly testing a given file. If the file belongs to a strong component and any other file in that component is changed, rigorous testing dictates that it be retested. This makes a compelling argument in favor of continuous regression testing using test harnesses, as proposed in [20][21] , and [22] .

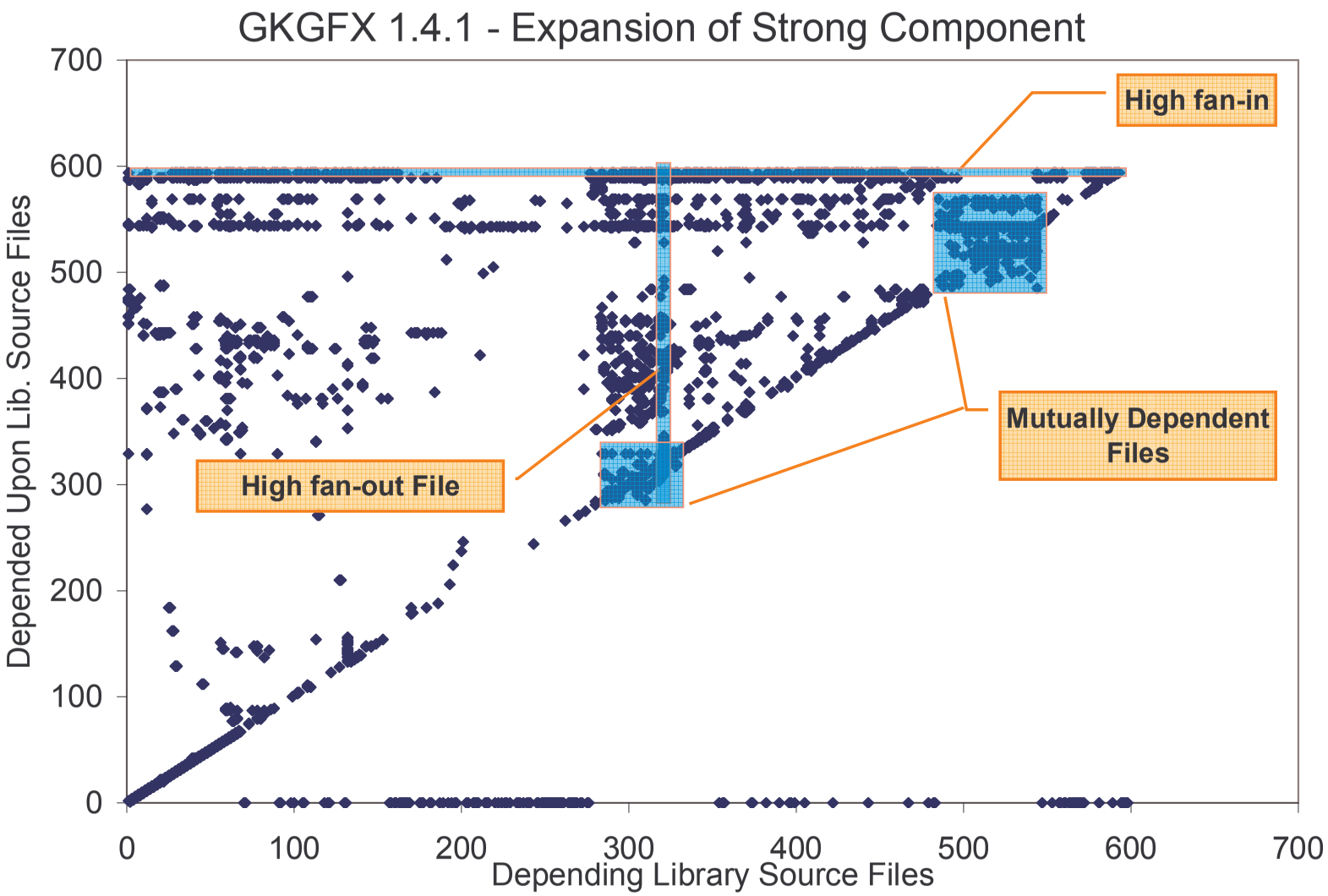


Figure 3.13 – Topologically Sorted Strong Components after Expanding

Ideally, we would like to see the data in Figure 3.13 clustered just above the diagonal. That would indicate that most dependencies were local, e.g., nearest neighbor in the sense of the sorted dependency graph. We also would like to see only a few dependencies below the diagonal, representing necessary mutual dependencies, perhaps due to event callbacks or intimately related class structures, as in the mutual dependency between a graph class and its node class.

Figure 3.14 shows the expansion of strong components after topological sort for entire Mozilla version 1.4.1 with 6193 files.

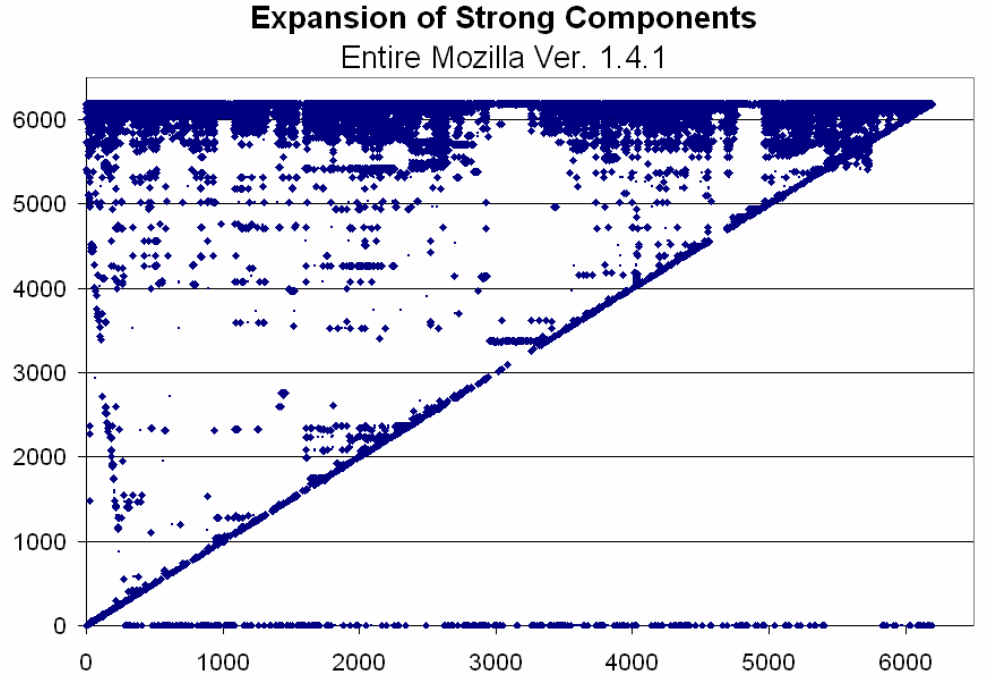


Figure 3.14 – Expansion of Strong Components after Topological Sort, Entire Mozilla

The purpose of this analysis is to demonstrate that our tools are capable of analyzing more than six thousands files within 4 hours. The figure also shows that it is difficult to interpret the structure of the entire system, except in very broad terms, due to the density of its dependency relationships. This is why we focus mostly on individual libraries, in this chapter.

For comparison, we show below, in Figure 3.15 a similar plot for expansion of strong components for Microsoft Foundation Class Library (MFC) module [23].

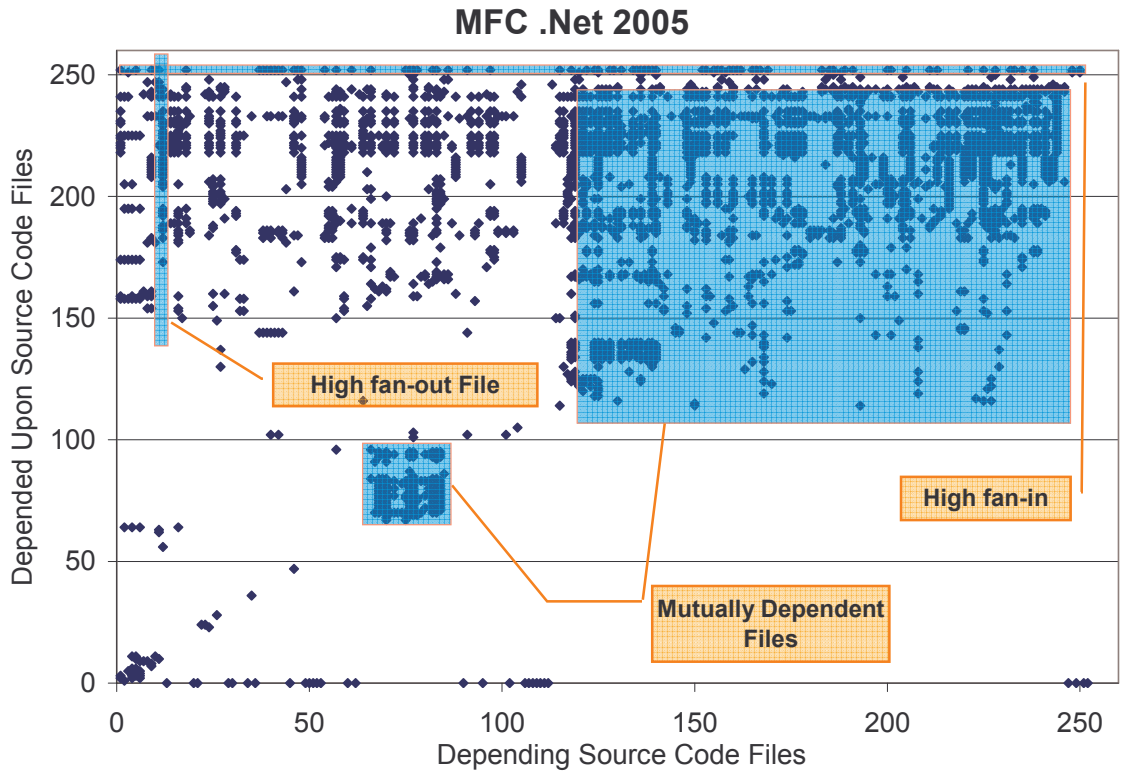


Figure 3.15 – Expansion of Strong Components after Topological Sort, MFC

We see similar undesired structural property in MFC as in GKGFX library. There are many files with high fan-in and high fan-out, and large strong components. Figure 3.16 is DepView of the MFC; strong components and dependency among strong components can be seen clearly.

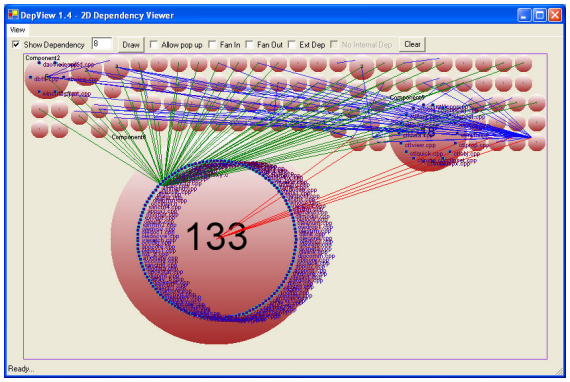


Figure 3.16 – Dependencies between components of MFC

Figure 3.17 shows the fan-in chart of MFC. Tooltip shows the value of fan-in size along with file name, as we see wincore.cpp has high fan-in value of 104.

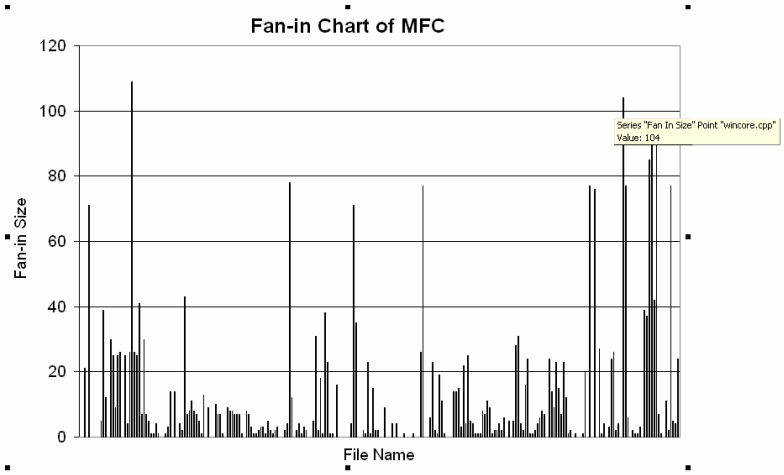


Figure 3.17 – Fan-in chart of MFC

In Figure 3.18, we show the fan-out chart of MFC. As stated earlier, high fan-out degrades the comprehensibility of a file. We see some of the files are using services of many files.

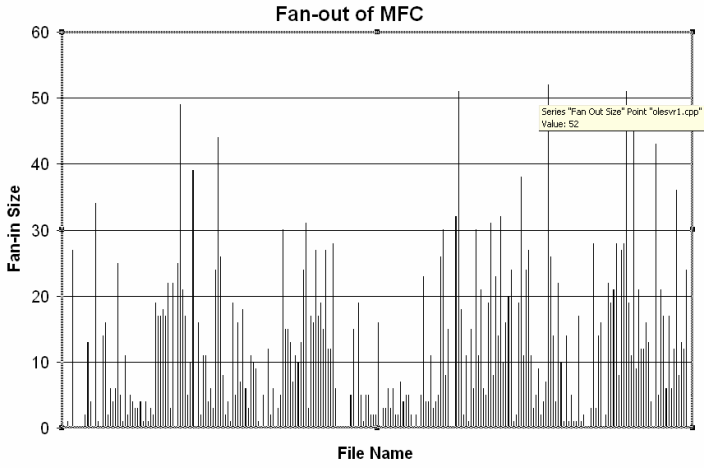


Figure 3.18 – Fan-out chart of MFC

As we see figures above, all these information can be extracted without semantic analysis. And all disclose different aspect of structural quality of software project.

3.2. Summary

In summary, static structure provides both quantitative and qualitative information regarding structural problems. Type-based dependency analysis is a useful tool with which to direct implementation and testing of large projects, and even for not-so-large projects. We can draw conclusions about:

- Quality of abstractions used in the project, based on fan-out of individual files.
- Potential for consequential change when files with high fan-in have poor quality, as indicated by internal metrics.
- Difficulty preparing effective test plans when files have high fan-out, especially in the presence of mutual dependencies.
- How well the project is packaged into modules.

The empirical study has demonstrated that useful information about significant problems in both large and small systems can be identified without a detailed knowledge of the entire code base. In the following chapter, we present how we can identify precisely which files are responsible for weak structural quality.

Chapter 4

Software Product Risk Model

This chapter focuses on both the risks associated with complex software structures and the ability to identify components for potential reuse. We developed a file-rank procedure that orders the entire system's file set by increasing risk, the product of importance and test risk, both defined in this chapter. This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve testability and product risk. Another contribution, discussed in this chapter, is a model that indexes software components according to their potential for reuse. This reusability index provides help to developers by ranking source code in existing systems, based on its place in the structure of the system and internal metrics. This enables developers to evaluate a file for reuse before looking at its code.

As we have shown, in Chapter 3, development of large software systems creates many, often thousands, of source code files with complex inter-dependencies. Clusters of mutually

dependent files introduce the possibility of a chain of forced consequential changes when a single cluster member file is changed. We have applied this model to a library from the 1.4.1 release of the open source Mozilla project, to the well known Microsoft Foundation Class (MFC) library (MFC analysis is in section 6.1.2 at page 128), used to develop windows applications, and to our own analysis software, all with interesting results.

4.1. Risk Model

We observe, in this chapter, that clusters of mutually dependent files introduce the possibility of a chain of forced consequential changes when a single cluster member file is changed, perhaps to repair a latent defect or improve system performance [55]. The model shows that density of dependencies within such clusters plays a crucial role in this behavior. Increasing density leads to increased risk of essentially unending sequences of change, known as thrashing. Our model is derived from a notion of test risk, based on the work of Jungmayr[8], combined with a measure of importance, for each file. We develop a file-rank procedure, which orders the entire system's file set by increasing risk, the product of importance (a measure of the number of files that depend on it) and test risk (a function of its internal quality and the testability of the files it depends upon), both defined in the following sections.

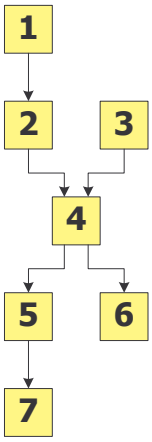
This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve project risk. We have applied this model to a library from the 1.4.1 release of the open source Mozilla project, composed of 598 files of source code, with the results, presented in Section 4.2.

The contributions of this research will, we believe, be useful for any of the disciplines that depend on large complex code bases. Computational Biology, Aerospace Systems, and Medical Imaging Systems, among many others, depend on large software toolkits, analysis systems, and display technology. Because much of the current work in these areas is new research or advanced product development, the codes that support those disciplines are continuously evolving and new software tools appear frequently.

The methods of this research provide direct support for management of large developing code bases. Not only are weakness discovered, but the model provides direct prescriptive guidance to improve the quality and reduce project risk of these systems.

4.1.1 Dependency Structure

In Figure 4.1, each square represents a source file and the arrow between the squares shows dependency relationship between two source files. If the arrow points from file A to file B, then file B provides services to A and A depends on B. In this example, files 6 and 7 are the most independent files since they do not use any other files' services. It is straightforward to test them, at least in terms of these structural relationships. However, this does not imply that these files are unimportant. On the contrary, files 6 and 7 provide services to many files above them, so their importance in this example is high.



In this sample project, file 1 has high test risk, due to its dependence on all the other files except file 3, either directly or indirectly. Any change to these files will require file 1 to be retested. However, its importance is low, in that no other files depend upon it for services. The opposite is true of files 6 and 7. Files 2, 3, 4, and 5 are intermediate cases that we will analyze below.

Figure 4.1 – Simple dependency between files

To discover the state of software system, we develop a file-rank [45] procedure, which orders the entire system's file set by increasing risk. The risk of a file is the product of its importance and its test risk, which are described in the following sections.

4.1.2 File Importance

Here we define importance from the perspective of change impact. Importance of a file is based on the number of other files that directly or indirectly depended upon it. The degree of importance is based on the likelihood of a chance in this file causing change to the files that use its services (or Change Impact Factor, CIF value covered in Chapter 5). Importance, I, can be greater than or equal to 1. File 1 has importance 1 ($I_1 = 1$), since no other files depend on it - it can be changed without worrying about anything other than its internal implementation. If we select another file, which is being used by other files, it will have higher importance, since any change applied to that file may affect the files using it.

Figure 4.2 shows the formula of importance calculation of file i, and the arrow indicates the direction of the dependencies.

$$I_i = 1 + \sum_{AllCallers} \alpha_{ij} I_j$$

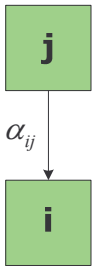


Figure 4.2 – Example of importance of a file and formula of importance calculation.

Here we use coefficient alpha (α_{ij}), which indicate change impact factor between file i and j. α_{ij} is the likelihood of a consequential change in file j when a change occurs in file i. If there is no risk that a change in file i will affect file j, then $\alpha_{ij} = 0$, and there is no contribution, from that file, to the importance of file i.

The smaller (closer to 1) the importance value for file i is, the better, in terms of impact of modifications to this file on the remaining files in the system.

Table 4.1 demonstrates step by step calculation of importance of file 1 through file 7.

$I_1 = 1$
$I_2 = 1 + \alpha_{21} I_1$ $I_2 = 1 + \alpha_{21}$
$I_3 = 1$
$I_4 = 1 + \alpha_{42} I_2 + \alpha_{43} I_3$
$I_5 = 1 + \alpha_{54} I_4$ $I_5 = 1 + \alpha_{54}(1 + \alpha_{42} I_2 + \alpha_{43} I_3)$ $I_5 = 1 + \alpha_{54}(1 + \alpha_{42}(1 + \alpha_{21}) + \alpha_{43} I_3)$ $I_5 = 1 + \alpha_{54} + \alpha_{42}\alpha_{54} + \alpha_{21}\alpha_{42}\alpha_{54} + \alpha_{43}\alpha_{54}$
$I_6 = 1 + \alpha_{64} I_4$ $I_6 = 1 + \alpha_{64} + \alpha_{42}\alpha_{64} + \alpha_{21}\alpha_{42}\alpha_{64} + \alpha_{43}\alpha_{64}$
$I_7 = 1 + \alpha_{75} I_5$ $I_7 = 1 + \alpha_{75} + \alpha_{54}\alpha_{75} + \alpha_{42}\alpha_{54}\alpha_{75} + \alpha_{21}\alpha_{42}\alpha_{54}\alpha_{75} + \alpha_{43}\alpha_{54}\alpha_{75}$

Table 4.1 – Calculation of importance, I of files in Figure 43.

α_{ij} is the impact strength, which indicates the affect on upper level files of changes in called files. If it is certain that a change in file 2 will cause a change in file 1, $\alpha_{21} = 1$, and the importance of file 2 is $1 + \alpha_{21} = 2$, e.g. the number of files changed when file 2 changes. If α_{ij} evaluates close to 1, it indicates that upper level files will be affected significantly by changes occurring in lower level files which provide services, so importance will increase rapidly. If α_{ij} is close to 0, it indicates upper level files will not be affected much by changes occurring in low level files and the lower level files are not so important[26].

[26] This might be due to the used file offering an interface it implements. If all callers bind to the file's types using the interface contract, it is far less likely that changes in the called file will result in consequential changes in the callers, than if they bind directly to the concrete types within the file.

Let's assume, for a moment, that all α values are identical and elaborate on the importance of file 7. Consider the importance equations, shown below, for the path from file 1 to 7. The effect of file 1 on importance of file 7 is α^4, this indicates that a change in file 7 is less likely to require one or more changes in file 1 than in files 3 or 2. Since the effect of change in file 7 will likely be handled by other files before reaching file 1. Consequential change becomes progressively less likely. Before reaching upper level files it has to pass through many other files. Consequently, change is most likely to affect immediate callers.

$$I_1 = 1 \;\Big|\; \begin{matrix} I_2 = 1 + \alpha I_1 \\ I_2 = 1 + \alpha \end{matrix} \;\Big|\; \begin{matrix} I_4 = 1 + \alpha(I_2 + I_3) \\ I_4 = 1 + \alpha(1 + \alpha + 1) \\ I_4 = 1 + 2\alpha + \alpha^2 \end{matrix} \;\Big|\; \begin{matrix} I_5 = 1 + \alpha I_4 \\ I_5 = 1 + \alpha + 2\alpha^2 + \alpha^3 \end{matrix} \;\Big|\; \begin{matrix} I_7 = 1 + \alpha I_5 \\ I_7 = 1 + \alpha + \alpha^2 + 2\alpha^3 + \alpha^4 \end{matrix}$$

We see, from the equations above, that file 5 contributes by α to importance of file 7. This shows that the most likely file to be effected by change in file 7 is file 5.

4.1.3 Brief Discussion of Alpha Value Calculation

As we see above, each dependency has an associated pair of values, α_{ij} and α_{ji}. That is, given a dependency of file i on j, a change to i may affect j and vice versa. If file i needs additional services, that may cause file j to change, and a change in j, perhaps to fix a latent error may require a change in its caller i, perhaps due to a changed formal parameter. α values may be different for each file dependency.

Alpha values depend, in part, on a project's development process and on the skill of its developers. A skilled developer often uses techniques to ensure loose coupling between components, while a less skilled developer may design in a way that causes many concrete bindings. As a result, average α values differ for each project. Since the values of α_{ij} have to be measured for a given set of developers and project environment, it would be desirable to

implement an, at least partially, automated process for estimating the α_{ij} s as part of a configuration management process[27].

4.1.4 File Testability, T

Testability is the degree of relative effort required for a file to be tested based on number of files it is using and its strength of interconnectedness with them, as well as internal implementation quality. "*A lack of testability contributes to a higher test and maintenance effort*" [8]. Testability or Test Risk of a software file is an important issue in assuring that required functionality is implemented without errors. Testing a file that uses services of others is harder than testing a file that performs its required task without depending on other files. In Table 4.2, Test Risk of file 6 and 7 are the lowest rank. The smaller T (close to 1) is, the more testable the file.

Below, we introduce implementation quality (β), which is described in section 4.1.5

$$T_n = \beta_n + \sum_{AllCalled} \alpha_{mn} T_m$$

The magnitude of Test Risk metric, T_n, varies according to the depended upon files' internal structure, as represented by β_n, and the project's dependency structure. β_n is the test risk of file n in isolation. T_n is the test risk accounting for retesting necessary when one of the file's dependent files changes and it must change.

Table 4.2 demonstrates step by step calculation of testability of file 1 through file 7.

[27] This is a research topic being addressed by another member of our research group.

$T_1 = \beta_1 + \alpha_{21} T_2$ $T_1 = \beta_1 + \alpha_{21}\beta_2 + \alpha_{21}\alpha_{42}\beta_4 + \alpha_{21}\alpha_{42}\alpha_{54}\beta_5 + \alpha_{21}\alpha_{42}\alpha_{64}\beta_6 + \alpha_{21}\alpha_{42}\alpha_{54}\alpha_{75}\beta_7$
$T_2 = \beta_2 + \alpha_{42} T_4$ $T_2 = \beta_2 + \alpha_{42}\beta_4 + \alpha_{42}\alpha_{54}\beta_5 + \alpha_{42}\alpha_{64}\beta_6 + \alpha_{42}\alpha_{54}\alpha_{75}\beta_7$
$T_3 = \beta_3 + \alpha_{43} T_4$ $T_3 = \beta_3 + \alpha_{43}\beta_4 + \alpha_{43}\alpha_{54}\beta_5 + \alpha_{43}\alpha_{54}\alpha_{75}\beta_7 + \alpha_{43}\alpha_{64}\beta_6$
$T_4 = \beta_4 + \alpha_{54} T_5 + \alpha_{64} T_6$ $T_4 = \beta_4 + \alpha_{54}\beta_5 + \alpha_{54}\alpha_{75}\beta_7 + \alpha_{64}\beta_6$
$T_5 = \beta_5 + \alpha_{75} T_7$ $T_5 = \beta_5 + \alpha_{75}\beta_7$
$T_6 = \beta_6$
$T_7 = \beta_7$

Table 4.2 – Example of testability, T of files in Figure 43.

For simplicity, assume α values are identical and β s are all 1[28], and let's calculate the testability of file 1. As we see in Figure 4.3, file 1 depends on all the other files except file 3, therefore internal complexity and dependency density of file 3 does not affect file 1. If we look at the effect of file 7 on testability of file 1, it has coefficient of α^4, clearly it has little impact, assuming α is small compared to 1. However, testability of file 2 has coefficient of α, which has the highest impact. Once file 2 is changed, file 1 has to be re-tested again to make sure change in file 2 does not cause breakage in file 1.

[28] The lower bound on β is 1 as shown in section 4.1.5.

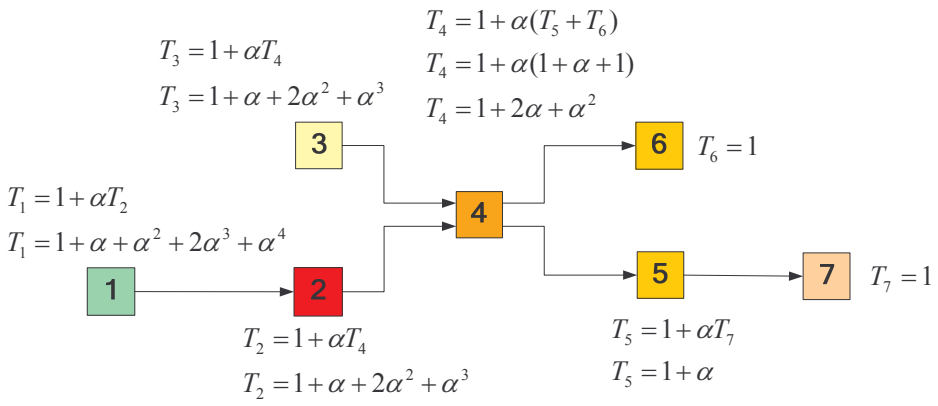


Figure 4.3 – Calculation of Test Risk of files, assuming β is 1 and α values are identical.

Without considering circular dependency, immediate dependencies have strong effect on testability of a file, and dependencies that are more distant have lower effect on testability. Nevertheless, it is never zero.

4.1.5 Implementation Metric Factor, β

Test Risk of a file depends not only on its internal implementation quality, but also on the quality of the files that it depends on. For this reason, metric factor, β, of many other files in the project may affect the test risk of any specific file. A number of metrics may be chosen to evaluate β. For this research, we use average lines of code per function and average cyclomatic complexity per function. For our own work, we take 50 lines of code and cyclomatic complexity of 10 as upper bounds of desirable values for these metrics. We use these bounds to normalize the metric factor, as follows:

$$\beta_i = 1 + \frac{1}{N}\sqrt{(\frac{m_{1i}}{M_1})^2 + (\frac{m_{2i}}{M_2})^2 + ... + (\frac{m_{Ni}}{M_N})^2}$$

$$\beta_i = 1 + \frac{1}{N}\sqrt{\sum_{j \in (1,N)} (\frac{m_{ji}}{M_j})^2}$$

Lowercase m is the measured metric, uppercase M is boundary value metric. The smaller β value is the better. In Appendix 1, we study relationships between code metrics and change count histories for a large project. This way we can use appropriate metrics during the calculation of β value.

There are other potentially useful quantitative metrics that can be utilized during β calculation, such as

- Number of function declarations
- Strong component size
- Number of global object declaration count
- Total line of code
- Number of change occurred during a change period

4.1.6 Case of Circular Dependency

In the case of circular dependency, testability of each member file of a mutual dependency set affects testability of other members of the set. Also, each file is important for other files, since they either directly or indirectly depend on each other. Figure 4.4 shows the mutually depended files of three and importance calculation of file 1.

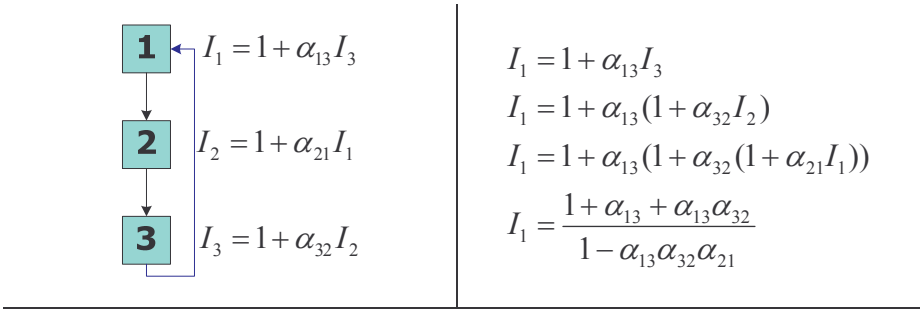


Figure 4.4 – Effect of circular dependency on importance.

Figure 4.5 shows the calculation of testability value for file 1. Dividing a number by a number less than one causes multiplication affect. As a result, it increases the testability value as in the case of importance above.

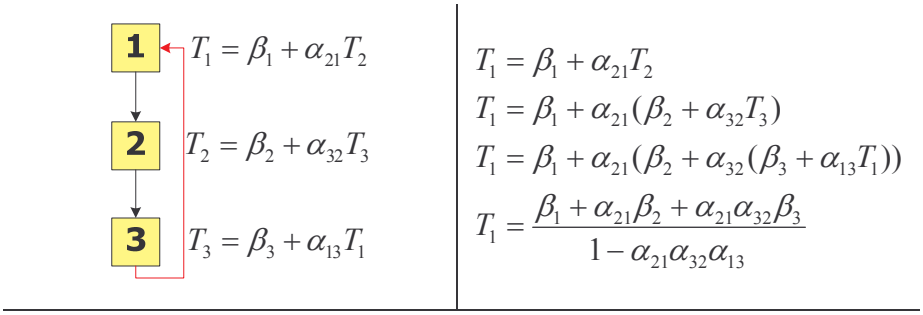
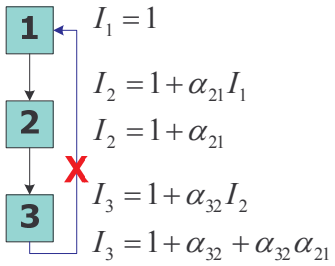


Figure 4.5 – Effect of circular dependency on testability.

In Figure 4.4 and Figure 4.5, we see the effect of circular dependency over Test Risk and importance. As identified, α_{ij} are always less than 1, dividing importance by $1 - \alpha_{13}\alpha_{32}\alpha_{21}$ or Test Risk by $1 - \alpha_{21}\alpha_{32}\alpha_{13}$ makes Test Risk and importance increase. Thus circular dependency increases Test Risk, since a change in any file may affect every file in the mutual dependency set.

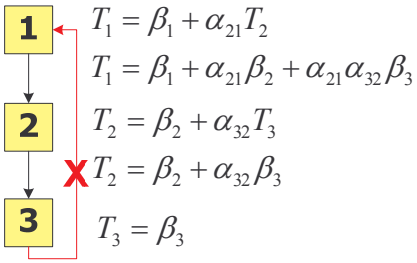


After breaking circular dependency at Figure 4.4, importance value of file 3 reduced

from $I_3 = \frac{1 + \alpha_{32} + \alpha_{32}\alpha_{21}}{1 - \alpha_{32}\alpha_{21}\alpha_{13}}$

to $I_3 = 1 + \alpha_{32} + \alpha_{32}\alpha_{21}$

Figure 4.6 – Importance, after removing circular dependency in Figure 4.4



After breaking circular dependency at Figure 4.5, testability of file 3 reduced

from $T_3 = \dfrac{\alpha_{13}\beta_1 + \alpha_{12}\alpha_{13}\beta_2 + \beta_3}{1 - \alpha_{13}\alpha_{21}\alpha_{32}}$

to β_3.

Figure 4.7 – Testability, after removing circular dependency in Figure 4.5

4.1.7 Representation of Importance and Testability

In order to calculate importance and testability of large software systems, we developed the representation of the Importance and Testability equations, as shown below. Representing the importance and testability as matrices enables us to compute results for large projects. Figure 4.8 and Figure 4.10 show the matrix representation of importance and Test Risk for Figure 4.4.

$$\alpha \times \vec{I} = \vec{1}$$

$$\vec{I} = \alpha^{-1}\vec{1}$$

$$\begin{bmatrix} 1 & 0 & -\alpha_{13} \\ -\alpha_{21} & 1 & 0 \\ 0 & -\alpha_{32} & 1 \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Figure 4.8 – Matrix representation of importance

In Figure 4.9 the arrow shows the dependency direction. In the importance matrix, file i on the abscissa (x-axis) depends on file j on the ordinate (y-axis) and α_{ji} describes how likely it is that change in j would require change in i; $\alpha_{\text{Causing file, consequential change ocurred file}}$

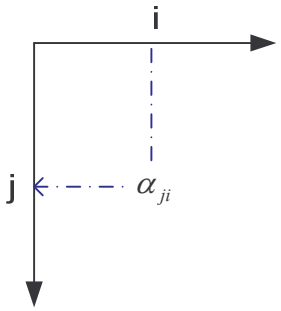


Figure 4.9 – Reading Importance Matrix

$$\alpha^T \times \vec{T} = \vec{\beta}$$
$$\vec{T} = \alpha^{T^{-1}} \times \vec{\beta}$$

$$\begin{bmatrix} 1 & -\alpha_{21} & 0 \\ 0 & 1 & -\alpha_{32} \\ -\alpha_{13} & 0 & 1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}$$

Figure 4.10 – Matrix representation of testability

Reading testability matrix used in product risk analysis

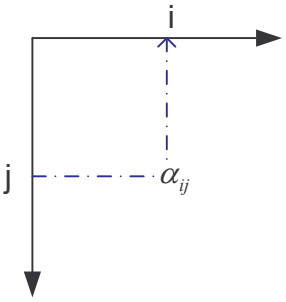


Figure 4.11 – Reading Testability Matrix

In testability matrix, file j on the ordinate (y-axis) depends on file i on the abscissa (x-axis) and α_{ij} is the value how likely change in i would require change in j;

$\alpha_{\text{Causing file, consequential change ocurred file}}$

Formulated as;

$$\alpha_{ij} = \frac{\text{Number of consequential changes in j due to i}}{\text{Number of changes in i}}$$

When there are more than a single cyclic path there is a critical value for α_{ij} at which the solution for importance and Test Risk becomes singular, e.g., the risk becomes unbounded. This indicates that a change made on a component with unbounded risk is likely to cause an unending sequence of changes[29].

It can be clearly seen (Figure 4.6 and Figure 4.7) that:

- Circular dependency reduces the software system's testability,
- Less important files are given elevated importance. Consequently,
- Small changes can affect many other files.

Figure 4.12 illustrates this for three mutually dependent files, and we use this example to illustrate finding the critical alpha value along with calculation of importance and testability values.

$$I_1 = 1 + \alpha_{12} I_2 + \alpha_{13} I_3 \text{ and } T_1 = \beta_1 + \alpha_{21} T_2 + \alpha_{31} T_3$$

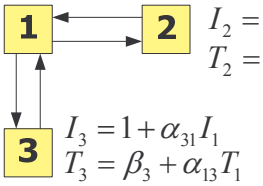


Figure 4.12 – Three mutually depended files.

In Figure 4.12, if for all i, j, α_{ij} are greater than 0.7071, behavior becomes undefined, as the change sequence becomes unbounded.

$$I_1 = \frac{1 + \alpha_{12} + \alpha_{13}}{1 - \alpha_{12}\alpha_{21} - \alpha_{13}\alpha_{31}}$$

and

$$T_1 = \frac{\beta_1 + \alpha_{21}\beta_2 + \alpha_{31}\beta_3}{1 - \alpha_{21}\alpha_{12} - \alpha_{31}\alpha_{13}}$$

[29]. Essentially, our risk model is a Markov process that becomes unstable at the critical value for α_{ij} .

It can be clearly seen in Figure 4.12 that circular dependency increases the software system's Test Risk and file importance. Importance increases since a change in any given file affects all files in the mutually dependent set, including possibly itself. A few more simple cases with increasing numbers of paths show that, as the density of dependency paths increases, the critical value for α_{ij} decreases.

4.1.8 Critical Dependency Density

Software components with high dependency density have several undesirable attributes. First, it is very hard to reuse files from the component because they depend on so many other files that extracting them is very difficult. Second, it is hard to test files in the component effectively because every time a test uncovers a defect, which we fix, we have to retest all the previously tested files in the component because they are all mutually dependent. A change in one may break the design or implementation of many others. Third, as it was demonstrated above, and further elaborated below, as the density of dependencies increases in a set of mutually dependent files we approach a critical point at which a single change causes, on the average, more than one other file to change, causing a chain reaction of changes. This behavior has been noted by others in large complex systems [55], but, until now, has not been satisfactorily explained.

Examining Figure 4.13 through Figure 4.16, we see that there is a critical value of α at which the solution for importance and testability becomes singular, e.g., the risk becomes unbounded. This indicates that a change made on a component with unbounded risk is likely to cause an unending sequence of changes. For the sake of simplicity, here we assume α values are equal among the files and β values are taken 1.

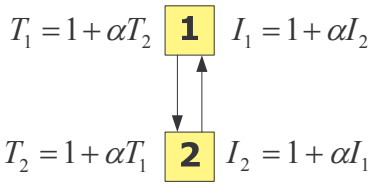


Figure 4.13 – Two mutually depended files, assuming β is 1 and α values are identical.

$$I_1 = I_2 = T_1 = T_2 = \frac{1+\alpha}{1-\alpha^2}$$

No critical value, except alpha is 1.

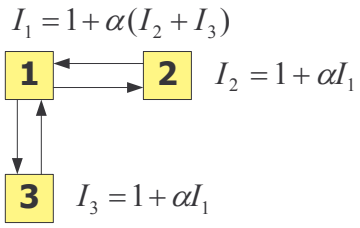


Figure 4.14 – Three mutually depended files.

$$I_1 = T_1 = \frac{1+2\alpha}{1-2\alpha^2}$$

$$I_2 = I_3 = T_2 = T_3 = \frac{1+\alpha}{1-2\alpha^2}$$

If alpha is greater than 0.7071, behavior becomes undefined, as the change sequence becomes unbounded.

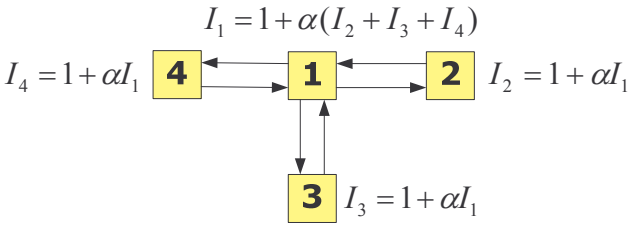


Figure 4.15 – Four mutually depended files

$$I_1 = T_1 = \frac{1+3\alpha}{1-3\alpha^2}$$

$$I_{2,3,4} = T_{2,3,4} = \frac{1+\alpha}{1-3\alpha^2}$$

Critical values is 0.5773

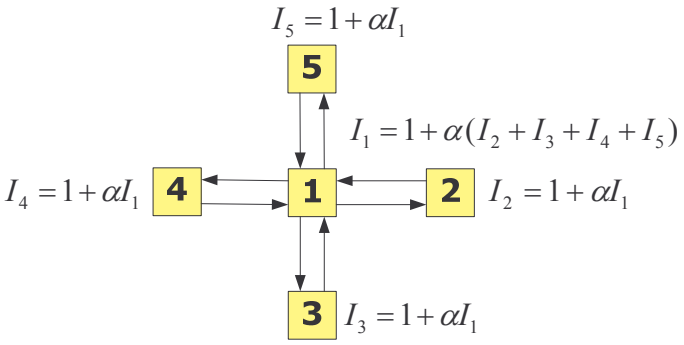


Figure 4.16 – Five mutually depended files.

$$I_1 = T_1 = \frac{1+4\alpha}{1-4\alpha^2}$$

$$I_{2,3,4} = T_{2,3,4} = \frac{1+\alpha}{1-4\alpha^2}$$

Critical values is 0.5

All these critical values are upper bounds for feasible α for the configurations discussed. Since at the critical Alpha value consequential change grows explosively, the higher the value is,

the better. As we see from the figures, if size of the strong component gets larger, critical value of α gets smaller. If critical value of α is closer to 0, this indicates the project is fragile. New changes, fixes, and new feature additions are likely to initiate many other required changes in the project.

The chart, in Figure 4.17 shows strong component size versus critical α value, indicating increase in strong component reduces the flexibility of software project for change.

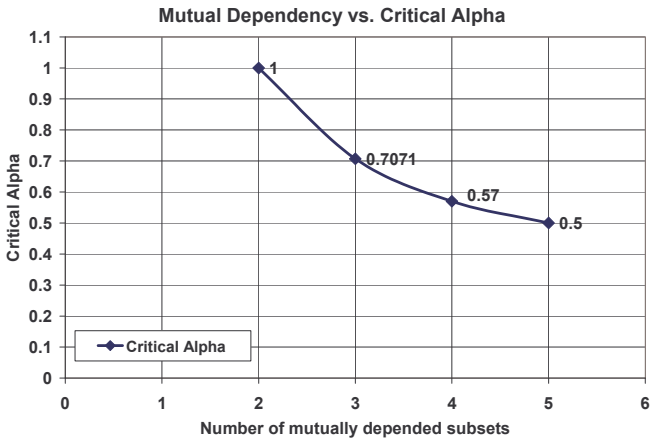


Figure 4.17 – Change in strong component size vs. change in α for Figure 4.13 thru Figure 4.16

If critical α value is large, it implies the project can accommodate changes, fixes, and new features. If not, an original change becomes risky, due to chains of consequential change.

Adding an additional closed path within the mutually dependent set will decrease the value of α.

Recalling that α is the probability that a change in a component will cause a change on a file that depends upon it, this means that as the dependency density increases, in a set of mutually dependent files, the system gets closer to, or reaches, the point of unending changes[30]. Only mutual dependencies can cause singular behavior. Mutual dependency sets with a single closed path are singular only when α = 1, as in the case shown Figure 4.13.

[30]. Note that representing likelihood of consequential change with a single probability across all files in a library is a simplification that allows the construction of risk equations, but diminishes the accuracy the model. Surely α will vary from file to file due to differences in the way it couples with other files. This situation is analogous to biological and cosmological modeling where one makes approximations that permit effective modeling, and then compare the model results with specific measures of reality.

This is a very interesting result. Our risk model exhibits behavior that many developers have observed in practice but understood only as being some unpredictable aspect of system complexity. Using risk analysis, we can estimate how prone a real evolving software system is to unbounded change.

4.1.9 Product Risk Model, R

In order to narrow down our focus to files, which need close attention, we rank files according to internal implementation and external interaction with other files in the project. We call this ranking a software product risk model. Files with high ranks are targets for software engineers to use great care while re-using, enhancing functionality, or fixing latent errors, since any change to that file may force a chain of new changes.

Risk factor is calculated by product of importance and Test Risk metrics.

$$R_i = I_i x T_i$$

$$\vec{T} = \alpha^{T^{-1}} \times \vec{\beta} \qquad \vec{\mathrm{I}} = \alpha^{-1}\,\vec{1}$$

$$R = \alpha^{-1} \times \alpha^{T^{-1}} \times \vec{\beta} \times \vec{1}$$

A file with high Importance and high Test Risk will have a high project risk, while a file with low importance but the same high Test Risk will have lower Risk Factor.

Now, we have a file-rank procedure, which orders the entire system's file set by increasing risk, R_i, the product of Importance and Test Risk. This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve Test Risk.

Risk factor provides feedback about individual files, and also provides insight about the global state of a software project. For instance, if developer needs to test a file, risk factor will give an idea how much time to allocate for that task. Ranking files by Test Risk shows project

management where to focus effort to reduce overall risk by redesigning and re-factoring high risk files.

4.2. Empirical Study of Risk Model on Mozilla Library, GKGFX

We downloaded version 1.4.1 of the Mozilla Win32 configuration [16][12] . This included the entire build, which makes many executables and libraries. We were able to build all the libraries and executables in about a week's effort, using the information provided on www.mozilla.org.

We built some simple parsers to find all the files included in a specific build, based on compiler output. This included all common code and header files.

The information provided on the Mozilla web site was very well prepared, easy to digest, considering the size of this large project, and straightforward to use. We chose this project because of the quality of its tools and the fact that it has a very large code base.

We applied our risk model to Mozilla GKGFX library and it gave us important insights about potential problem files, on which attention should be focused. This information was obtained without diving into implementation details. This attribute is very important for the software project's testers, developers, and managers, since project complexity may make it very difficult for any but the original developer to understand in detail a given set of files.

First, we explored the variation of maximum importance with $\alpha_{ij} = \alpha$, making the simplifying assumption that it is a single probability across all files in a library. Essentially, we are treating α as the average probability of a consequential change in a depending file when we change the depended file. Thus, these results will be qualitatively useful, but not numerically precise. We see, from the plot in Figure 4.18, that Importance grows without bound above α=0.1032. This indicates that changes are very likely to propagate throughout the system since one might expect the value of α to be of the order of 0.1. [55]

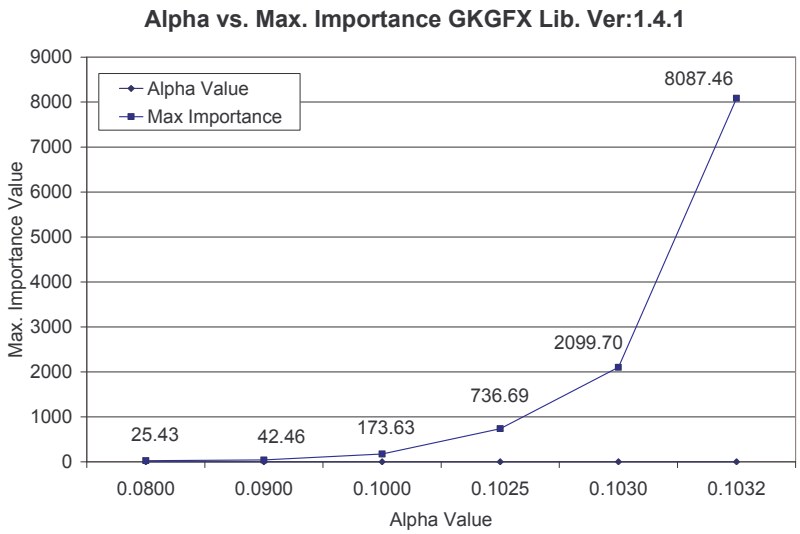


Figure 4.18 – Max Importance vs. Alpha (α) value for Mozilla GKGFX Library Version 1.4.1.

Next, we calculated product risk factor values using average cyclomatic complexity[31] (AvgCC) and Fan-out[32] values for each file in GKGFX when calculating β; upper limits were 10 for AvgCC and 5 for Fan-out. We took these values since it becomes harder to manage a file, which uses several other files' services, accordingly, it is hard to understand and test a file with high complexity functions.

Figure 4.19 shows the risk rate of all the files in Mozilla library, GKGFX, in increasing order, still estimating the alpha value to be 0.1.

[31]. AvgCC = Sum of CC of each functions in a file divided by number of functions in that file.
[32]. Fan-out is a number of depended files whose services are employed by a file.

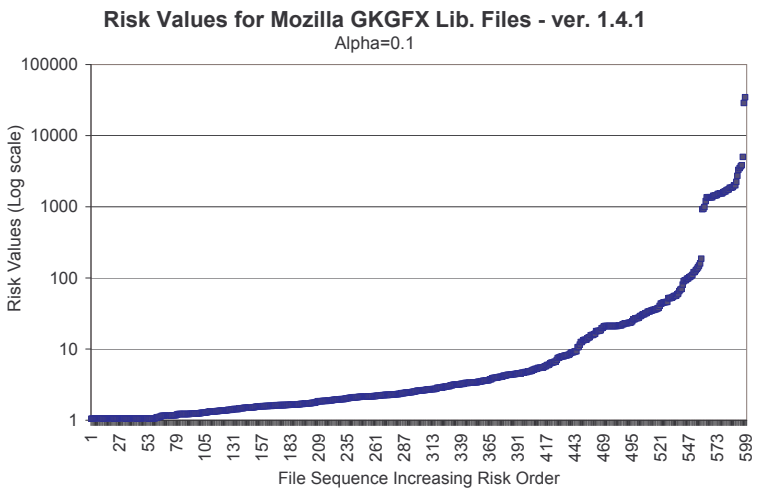


Figure 4.19 – Risk values for files in GKGFX Library

Note that about %10 of the files have most of the Risk in the library code [9] [51]. Interestingly, not all the files with high risk are part of a large strong component (shown in Figure 6.1). This shows that the high-risk files are not guaranteed to be part of the largest set of mutually dependent files.

4.3. Improving the Risk Model

One concern is the values used for alphas. Others have implicitly, or occasionally explicitly, assumed that alpha is either zero or one [8]. Simple constructive arguments, given in Chapter 1, show that this is not realistic, and our model attempts to provide better predictive capability by assuming alpha in the range [0, 1].

In Chapter 5, we develop a process and an experiment for calibrating a project's alpha values. The probability of change in a dependent file due to a change in the depended file will vary, in part, on how well both files are implemented, e.g. on the skill of the project's architect and developers. In order to use the model to provide accurate assessments of risk the project's change history needs to be instrumented to provide project specific alpha values.

4.4. Reusability Index, RI

Development of software systems requires intense labor. The larger the software project gets, the more labor it demands due to development of numerous source code files with complex interdependencies. This makes reuse of previously developed software components desirable to avoid some of the development effort and cost that would otherwise be required. Software reuse is one of the important factors to save development effort to reduce cost [67]. We describe, in this research, a model that indexes software components according to their potential for reuse. This "reusability index" provides help to developers by ranking source code in existing systems, based on its place in the structure of the system and internal metrics. This enables developers to appraise a file for reuse before looking at its code.

Below is our model of a "reusability index" to grade files according to their level of reusability. The purpose of this index is to examine files, in an existing product, for potential reuse. If a file is called by many others in the product, e.g., has a high fan-in, then it has demonstrated its usefulness, at least within that product by this in-situ reuse. If, however, it has a high fan-out, then it depends on many other files, which makes it much harder to reuse. Moreover, if the called files, in turn, call other files, then it is even less likely that the file in question can be effectively reused. For this reason, we expect the reusability to decrease as the closure, $\overline{FO}$, of the files fan-out increases. Finally, if the implementation metric factor, β, of a file is large, that is an indicator of poor quality, which means the component should not be a good candidate for reuse. With this reasoning, we arrive at the model shown below.

$$RI = \frac{FI}{FI + \overline{FO} + \beta} \quad \left| \quad \begin{array}{c} \beta \in (0, \infty), \\ \overline{FO} \text{ transitive closure of fan-out,} \\ RI \in [0,1) \end{array} \right.$$

This Reusability Index (RI) provides help to developers when they need to reuse a candidate source file, giving a value between 0 and 1. If reusability index is close to 1, it shows

this file can be considered for reuse. This will enable developers to evaluate a file for reuse without initially looking at its code. Looking at code to determine potential for reuse is labor intensive, especially for the large projects, and may be almost impossible to accomplish manually due to complex interdependencies.

4.5. Applying Reusability Index to a New Design for DepAnal

We applied this reusability Index model to our own dependency analyzer with encouraging results, as shown below in Figure 4.20. The indexed results agree well with our designer's knowledge of the reusability of our individual source files. DepAnal is a relatively small and simple project, so we can make fairly accurate assessments of the reusability of its parts. It is encouraging to get this kind of correspondence between our automated tool and personal assessments in this simple case.

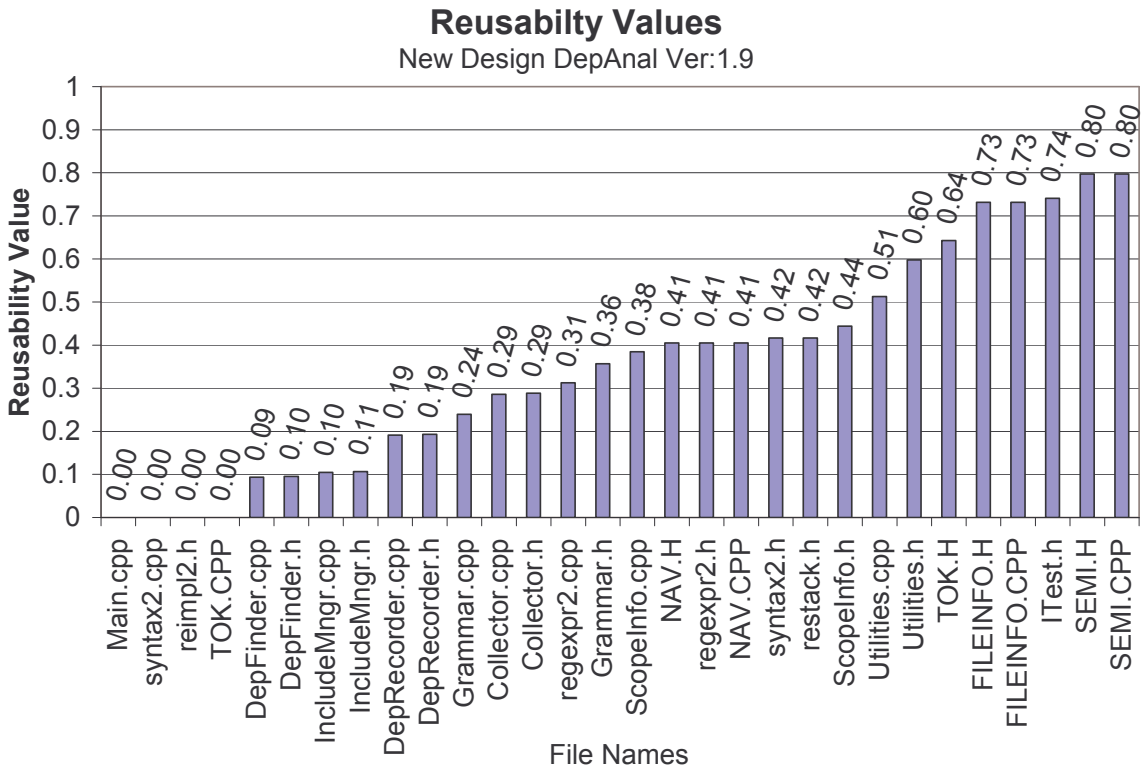


Figure 4.20 – Reusability Index of New Design DepAnal Ver. 1.9

Let's pick a couple of files from DepAnal and elaborate on their index values. Tok.cpp and Main.cpp are test stubs, they are certainly not intended for reuse and their reusability value is zero in accord with indexed results. Nevertheless, Tok.h, Semi.h and Semi.cpp are designed for reuse not only for DepAnal but also for C/C++ source code parsers. Encouragingly, all these files have high reusability index. The rest of the files are ranked about as we, as implementers of the code, expected.

4.6. Summary

In this chapter, we presented a new software product risk model and have shown that the model can be used to predict problem areas, as concentrations of high risk files. The model predicts that, as the density of dependency relations increases in strong components of the dependency graph, Risk factor grows and becomes unbounded at critical densities. We have applied the model to a library from a real open-source project, Mozilla, ver 1.4.1, where the

model predicted that most of the development risk is in about 10% of the library files. This useful information was probably unknown to its developers. In addition, we describe a model that indexes software components according to their potential for reuse. In the following chapter, change impact factor estimation between files is studied. This empirical study provides precise product risk values. Additionally, it enables us to compare using our estimated alpha values and calculated alpha values and their effect on product risk order.

We have shown that, when the density of dependency increases, the ability to accommodate new changes decreases. There is a specific density value at which this behavior becomes unbounded. The Product Risk model demonstrates that removing some dependencies reduces the size of large strong components and reduces the average risk per file. Dependency density can be trusted as a reliable indicator of the quality of the system. Excessive dependency increases complexity of software projects and diminishes ability of the developers to apply changes successfully. Software code becomes difficult to understand and reuse. Developers avoid reusing a component, which accomplishes some needed functionality with many depended components.

Chapter 5

Change Impact Factor Estimation

Change in software is always an essential part of software development and maintenance. Controlled management of change is achieved by being able to estimate impact of changes. This empirical study serves mainly to help understand the impact of a change in a software source file on other source files. We present the design of an experiment to measure these affects, describe its application, and show measured results of the change impact. We monitor evolution of change impact over one project's lifetime.

5.1. Introduction

In this research, we report on measurements of the impact of change in one file on other files, in a small design project, called DepAnal[33].. We will describe this measurement as change impact factor, α_{ij} and define that as:

---

[33]. DepAnal is one of the tools that we monitored its evolution from scratch for this research.

$$\alpha_{ij} = \frac{\sum \text{changes in file j due to a change in file i}}{\sum \text{changes in file i}}$$

Thus, Change Impact Factor (CIF) is the relative frequency of required consequential changes in files in the project. In an earlier research effort [60], we developed a product risk model that uses change impact factor for every dependency relationship between files in a project, but we could supply only rough estimates for the values of these parameters. The goals of this effort are to measure the CIF factors, as functions of time, for a real project, and also, to develop a measurement process that can be applied to other projects, as well. In this way, a more accurate assessment of risk is obtained, in real time, as a project unfolds.

We present the design of this experiment, describe its application, and show measured results of the change impact factors. These results help one to estimate propagation [70][71] of changes and calculation of the magnitude of change, CIF, for a project. The results of the study will improve accuracy of our Risk Analysis model calculation presented in Chapter 4 and a previous paper [60] by using systematically measured change impact factors, derived from an annotated change history. Consequently, all this information provides help to developers and project managers to find parts of their product that are at risk. Not only that, but it also guides them to make effective decisions with regard to implementing new changes and scheduling work activities.

5.2. Background Study

Michelle L. Lee [71] defines the objective of change impact analysis, this way:

"A major goal of impact analysis is to identify the software work products impacted by proposed changes. Evaluating software change impacts requires

identifying what will be impacted by a change and relies on the "impact assessment" to determine quantitatively what the impact represents."

Her dissertation [71] considers the impact of change on types, global functions and global data, such as how many classes are going to be affected by a change. Similar analyses are also found in [73].

In this study, we are interested in a coarser level of impact analysis, that of file-to-file change impact. Our choice is motivated by the conventional process of managing projects by files. Files are the unit of configuration management and analysis. Our risk model is based on file dependencies, calculated from the same kinds of static relationships used in [69],[71],[73] and [74], e.g., type, function, and global data. But, change impacts are empirically determined by carefully monitoring and recording original and consequential changes made to files, during development.

5.3. Change Impact Factor and Risk Model

The granularity of change impact factor in this experiment is focused on software source files. We are interested in determining the degree of interconnectedness between source files, to be able to estimate consequences of a change. The degree of interconnectedness is represented by α .

In Figure 5.1, arrows show directions of change causality and α_{ij} is the likelihood of a consequential change in file j when a change occurs in file i.

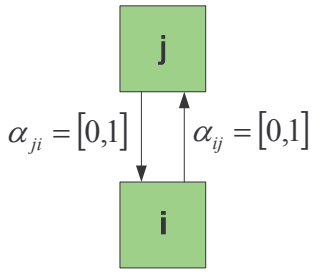


Figure 5.1 – Alpha value representations

Given any two files, i and j, there are two different alpha values between them. One is α_{ij} and the other is α_{ji}. α_{ij} =0 is the lower bound, which value implies that changes in file i are not going to affect file j. α_{ij} =1 is the upper bound, indicating any change in file i is going to affect file j. α_{ij} and α_{ji} are inherently two different alpha values, as we will see in the following section.

An alpha value is the ratio of the number of consequential changes made to a file to the total number of changes in a source (of change) file. The total number of changes is the sum of original and consequential changes. In the Figure 5.2, alpha calculation is carried out by dividing the number of consequential changes occurred in files E, F… X via file D by the total number of changes in file D.

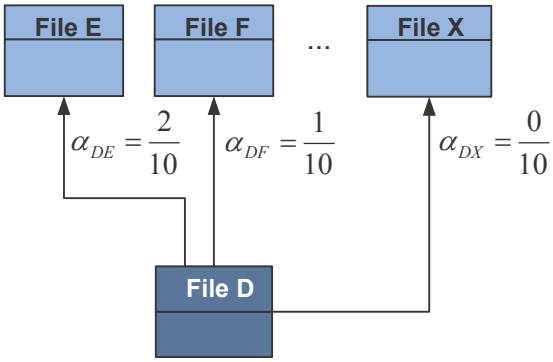


Figure 5.2 – Alpha values between file D and depending files.

In Figure 5.2, a sample alpha value calculation is illustrated. File D is providing services to files E, F and X. There are total of ten changes occurring in file D and two of them required file E to change. The calculation is shown here.

$$\alpha_{DE} = \frac{2}{10} = \frac{Consequential\ changes\ to\ E\ caused\ by\ changes\ in\ D}{Total\ changes\ in\ D}$$

The Product Risk Model ranks files according to internal implementation metrics and external interaction with other files in the project [60]. Risk factor, R is the product of importance and testability for file i, $R_i = I_i x T_i$. Both importance and testability of file i, I_i and T_i respectively, use alpha values during their calculation, as shown in the formulas below. (See Chapter 4, pages 76 and 79)

$I_i = 1 + \sum_{AllCallers} \alpha_{ij} I_j$	$T_n = \beta_n + \sum_{AllCalled} \alpha_{mn} T_m$
Importance of file i	Testability of file n

In Figure 5.3, we show risk, importance and testability values for each file of DepAnal [59], our tool that analyzes static dependencies between files. The benefit of product risk factor [60] is that it provides feedback about both individual files and insight about the global state of a software project. For example, in Figure 5.3, we see the risk contributions of each file to project and see immediately files, which pose high and little risk for the project.

Before testing a file, its Product Risk Factor provides an idea of how much effort to allocate for that task; also, it shows where to focus effort to reduce overall risk by redesigning and refactoring high-risk files.

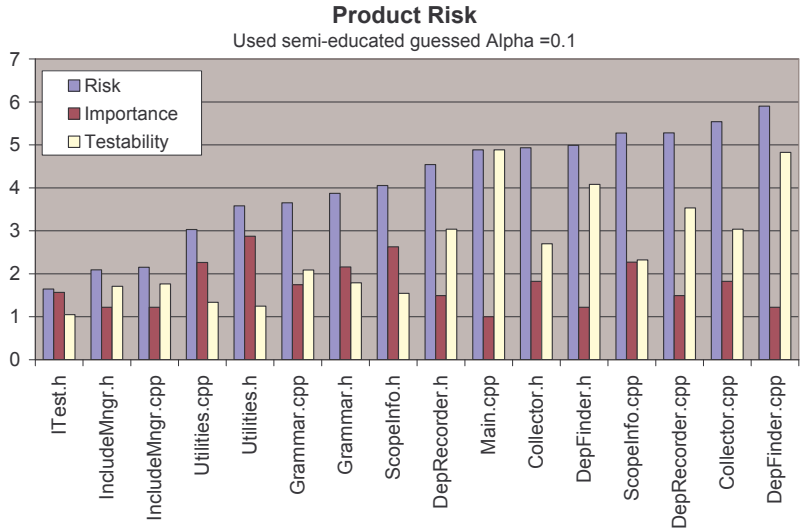


Figure 5.3 – Risk chart of New Design DepAnal [60]

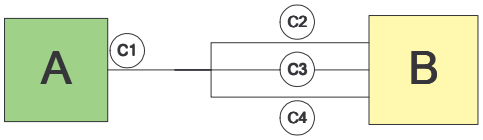
In the earlier research [60], alpha values were modeled as a single constant, 0.1; this was just an estimation. One of the goals of this research is to obtain experiment-based alpha values to strengthen the Risk Model. In addition, this will enable us to compare the results obtained through constant alpha with empirically obtained ones to observe whether differences in alpha value radically affect the ranking of files by risk values. We show, in Figure 5.15, a revised calculation using measured alpha values and in Figure 5.16 using measured $\alpha_{\text{Effective}}$, described in the paragraph, below. We see over 62% of the files stay either in place or at most two-places moved, comparing with file risk order obtained by individually calculated alpha values, as shown in Figure 5.15 and Figure 5.16. Details are presented in Section 5.9, below.

5.4. Experiment Design to Determine Alpha (α)

We designed an experiment to empirically determine alpha values and observer their changes over time. There are two essential points in this experiment design. The first one is to determine what is meant by a "change"; the second is to have a large enough software project to be a reasonable yardstick with which to measure other systems, but small enough to monitor implementation from start to end. Thus, we attempt to obtain a sufficient number of sample data points to carefully represent more general software systems.

By change, we mean a modification/addition/removal of code for any purpose (feature addition, bug removal, commenting, and cosmetic changes) to a file. In addition, making a group of cosmetic changes at once is a change. Each file has its own change history and each change is part of a daily file release in our project. Until first release, the changes made to a file will be counted as one change. First release will be counted as first change. Consequential change is a specific change, required to accommodate a previous change in another file. All other changes are by default original changes, indicating they are not initiated by another change.

One exception with consequential change is that in the same version, if a change requires more than one consequential change to a particular file, only first change to particular file should be recorded as a consequential and rest will be recorded as original changes.



As an example consider the case where a single change in file A causes one or more changes to file B. In Figure 5.4, changes labeled C2, C3, C5 are due to change C1, however only C2 is recorded as a consequential change, the rest are as original change.

Figure 5.4 – Change driving many changes.

Only direct changes due to a source change, not the transitive closure, are counted as consequential changes. Note that a consequential change may cause another consequential change.

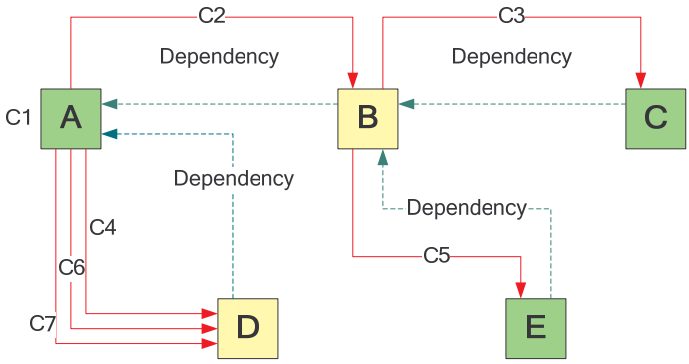


Figure 5.5 shows sample dependency structure and changes. In this figure, C1 is an original change; C2 through C5 would not happen if C1 did not occur. Nevertheless, we record C3 and C5 caused due to C2 not due to C1. C4 is recorded as consequential change too, however C6 and C7 are recorded as original changes.

Figure 5.5 – Sample change flow and dependency between files.

List 1. - Process for Defining Change

1. A file release can exhibit one or more changes.
2. After each successful compilation, all the tests should be exercised to make sure there is no breakage. If the breakage requires a fix in other files, this is recorded as a consequential change(s).
3. Each change is recorded in a maintenance page (comment section within each file) with date and change number. Such as Ver 2.1.a, 2.1 represents the version number of that file and "a" indicates the first change in this version. This is also done for implementation files (.cpp etc). Since our granularity is file level, we do not record changes for modules but always for individual files.
4. If a new function's declaration and definition are added to different files (header and implementation), we record each as a change in the Maintenance History page of the corresponding files. To be consistent we always accept declaration as original change and definition as consequential change.
5. During a fix, or a new feature addition, if several changes are required in the same function, this will be counted as one change, providing previously developed functionality remains intact.
6. During a modification or a fix, if a new global function is created, there will be at least two changes, one is the fix/modification, and the other is the new function. Nevertheless, it is not a consequential change, since both reside in the same file.
7. If a new class-member function is created, there will be total of three changes; declaration and definition of the new function both will be consequential changes, declaration will be consequential change of the fix, and definition will be consequential change of the declaration.

8. Addition of a new member variable is a consequential change of declaration required by implementation.
9. Adding/Removing an existing source file to/from a system is a change
10. Removing an already added file is not a change, providing that it is not supplying any services to others. If it does supply services, it will cause several external changes; therefore, file removal will be a change.
11. While Adding/Removing an existing source file to/from a system, all are original changes. We do not differentiate such as A.h is an original A.cpp is a consequential change.

5.5. Expected Outcome Prior to the Experiment

In the Product Risk Analysis in [60], we showed that alpha values above a critical point may make Product Risk become unbounded. This is due to a change causing a cascade of consequential changes, which, in turn cause other changes in an unstable fashion. Several alpha values were used for Risk calculation. Our experience with Mozilla libraries' Risk analysis showed that alpha values should be of the order of 0.1, to remain stable, and, as a first approximation, we took alpha value to be 0.1 for all files. This analysis result was qualitatively interesting, but not numerically exact.

A study, related to ours, was carried out by Jungmayr [8] were he calculated a system's testability based on interconnectedness; he did not take into account the degree of interconnectedness between source files. In other words, the alpha value in his research was one, since no coefficient is used to adjust the strength of dependency. However, dependency between files does not imply that every change in depended file causes change in a depending file. Most of the time changes remain local to a file.

Excessive consequential change indicates that it will be difficult to make future changes, due to the effects of the change on testing and additional consequential change, an undesirable

property. During our analysis of Mozilla [60], we observed that small increases in alpha value can cause unbounded product risk.

It is the purpose of this experiment to evaluate alpha values experimentally, to avoid the foregoing "ad hoc" adoption of specific values, even though we believe them to be reasonable.

5.6. Empirical Study Process Description

In section 5.4 our experiment design was disclosed. This section covers some practical details of the experiment. The sample data for this experiment came from a reimplementation of our C/C++ file level dependency-analyzer, DepAnal. The analyzer's first external release has 7796 lines of developed code, 5580 of these are code within functions. Implementation took three months, and 503 changes were recorded.

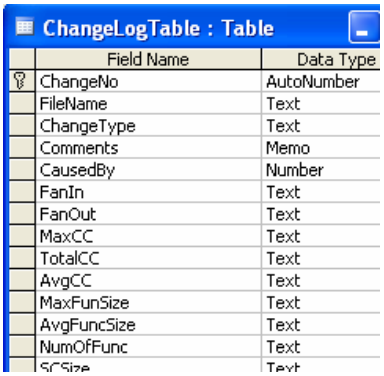
Statistical Information of the Analyzer	
Total code lines	22553
Total developed code lines	7796
Total developed line of code in functions	5580
Total cyclomatic complexity in evolved code	812
Time to first external release (months)	3
Number of changes recorded	503

Table 5.1 – Information Regarding the Experimental Project

Each change is recorded in a maintenance page for each file, where the change occurred. A change record contains the following information:

- Date
- Change number, qualified with internal release number
- Brief information regarding the nature of the change
- Whether it is a consequential change

We also created a Change Logger application, shown in Figure 5.6, to keep data in an organized fashion in order to query later. The Change Logger carries extra information regarding each file, shown in Table 5.2. This extra data is used for exploring correlations between metrics (structural or internal) and changes. This will be a topic of future research.



- Change type
- Number of depending files (FanIn)
- Number of depended on file (FanOut)
- Cyclomatic complexity
- Maximum and average function size
- Total line of code
- Etc…

Table 5.2 – Information in database regarding a file, where change occurred.

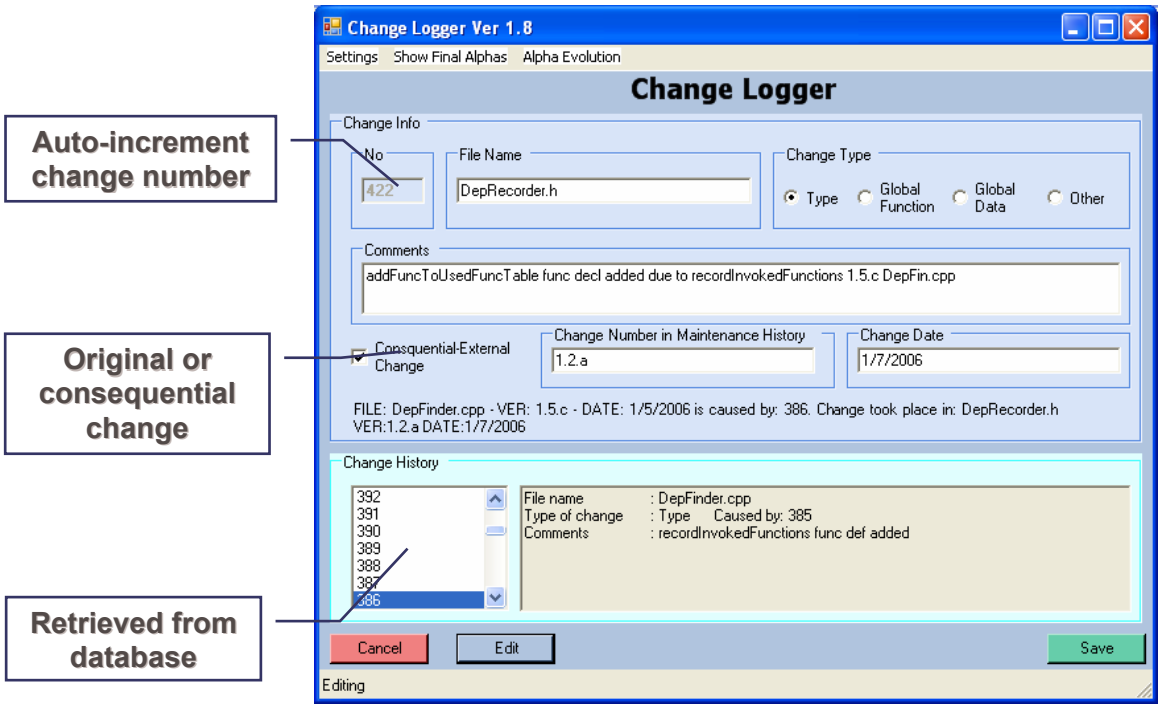


Figure 5.6 – Screen shot of Change Logger

Once a developer implements a change, he has to record it both in the database by using change logger (Figure 5.6) and in the maintenance page, before working on other parts of the software. To record a change, the following information is needed: filename where change occurred, brief textual explanation regarding the change, change number, type and date. If it is consequential change, the developer has to select which file caused the change.

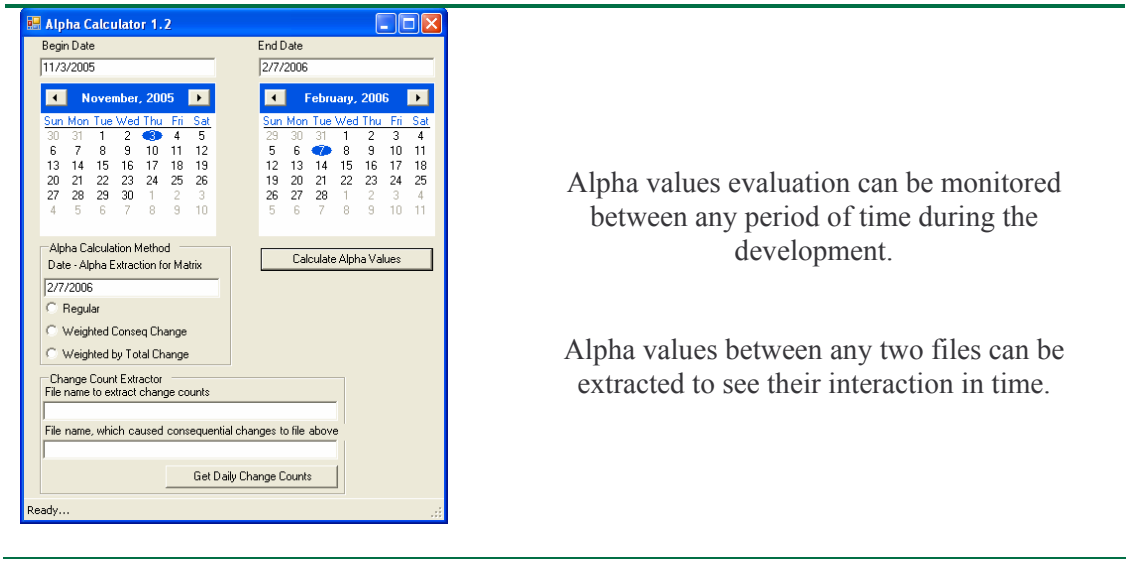


Figure 5.7 – Alpha value calculator

Figure 5.7 above shows Alpha Calculator, which can extract alpha values between anytime during project development. In addition, it generates matrix files to be used for product risk calculation.

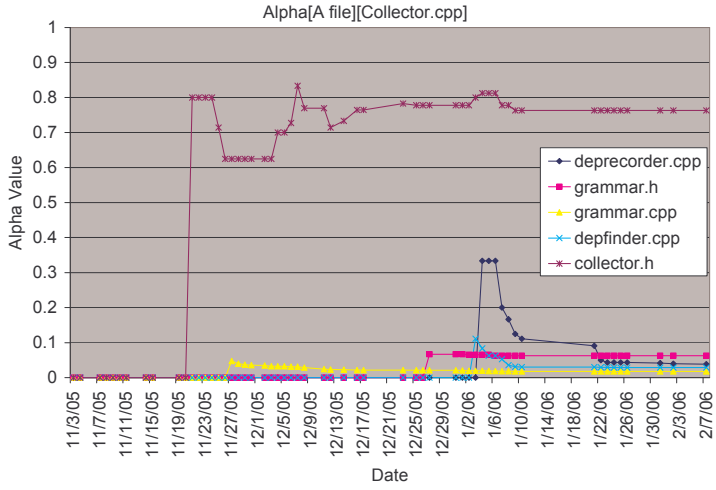
5.7. Our Results

This research provides several graphical outputs. One graphical output type is evaluation of alpha (Change Impact Factor, CIF) values chart for each individual file. Another one is evaluation of project's alpha values chart throughout development. CIF charts for a file have two forms; one form shows the number of consequential changes charged to file A due to changes in

other files. We show this CIF value as α_{XA}. X is files causing consequential changes to file A. Another form of chart is the number of consequential changes caused by file A to other files. We show this CIF value as α_{AX}. Changes are cumulative change counts in a given time interval.

A file's alpha evaluation chart discloses information about how likely this file will be affected by the changes in other files α_{AX}, or how likely changes in this file will affect other files α_{XA}. The chart below (Figure 5.8) shows alpha value of file Collector.cpp, such as $\alpha_{Grammar.cpp,Collector.cpp}$. We read $\alpha_{Grammar.cpp,Collector.cpp}$ as a change occurred in Grammar.cpp how likely will require change in Collector.cpp. File's alpha evaluation charts below do not disclose dependency information.

When we mention consequential change, generally, the scenario is like the following. If file A is using services of file B, change in file B causes A to change. However, some cases it can be just the opposite, such as, while file A using feature of file B, it can encounter a bug, and request file B to change. Another example file A can request new feature addition from file B. In the former case, consequential change is just the reverse direction of the dependency, but in the latter cases in the same direction of dependency.



In this chart we see, first sharp rise in alpha value ($\alpha_{Collector.h,Collector.cpp}$) then becomes stable. In most cases, a module's header and implementation file has higher alpha value compared to other files. This is expected, since they are intended to accomplish assigned tasks together. Until the design matures, frequent changes are normal.

Figure 5.8 – Alpha value evaluation of Collector.cpp throughout the first release.

Mostly changes between modules (header file and implementation file/s) are due to function signature change, adding/renaming/removing member data or function. All these changes are legitimate as are frequent changes between modules at early stages of a development.

$$\alpha_{A,Collector.cpp} = \frac{\sum \text{Consequential changes in Collector.cpp due to A}}{\sum \text{Changes in A}}$$

To be consistent while recording changes, we use the rule that definitions always depend on declaration. As wee see $\alpha_{Collector.h,Collector.cpp}$ is quite high; implying almost any change in Collector.h affects Collector.cpp. Because, any member function addition starts with its declaration then its definition. This means all the function definitions are consequential changes, and as a result high alpha value between header and implementation file are expected.

The lower the denominator is the higher the fraction. If there are not many internal changes recorded in file A, or all the internal change causes changes to the same file, this can

cause α value to be high. Lower alpha indicates files' level of independence from external changes. Therefore, the lower alpha value is the better.

When there is an increase in the alpha value, it indicates that consequential changes are occurring to subject file. Reduced or no change in alpha value indicates no consequential change is occurring to it.

Charts also disclose information regarding file's creation or inclusion time in the project. By looking at time line in Figure 5.8, it is seen that collector.cpp is created at the early stages of the project.

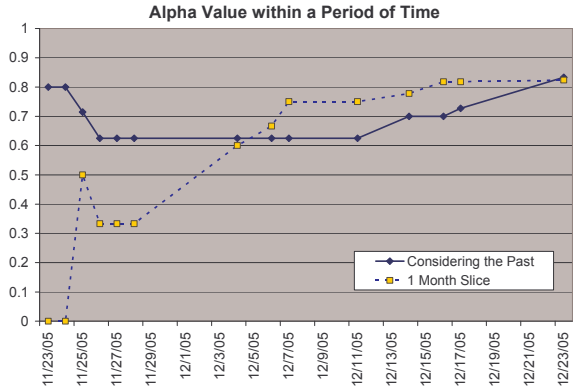


Figure 5.9 – Alpha value evaluation in 1 month period between Collector.h and .cpp

Figure 5.9 shows changes in $\alpha_{Collector.h,Collector.cpp}$ during the time frame of 1 month. The solid line uses all the change history from the beginning of the project. The dotted line uses no change history prior to the initial time of the graph. This figure allows us to observe alpha value in some time interval. See the discussion in the last paragraph of this section, e.g. 5.7.

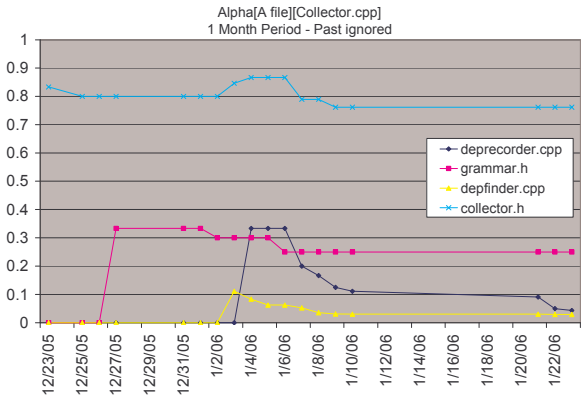


Figure 5.10 – Alpha values evaluation for 1 month period.

Figure 5.8 and Figure 5.10 are both Alpha values for Collector.cpp. In spite of the fact that both figures show the alpha value for the same file grammar.cpp does not appear in Figure 5.10. This is because no consequential change was occurred in Collector.cpp by that file during the time period covered. Moreover, alpha values in Figure 5.10 are different than the values in Figure 5.8 on the same days. That is due to ignoring past changes.

The sliding window time frame is useful for monitoring the evolution of alpha values in a certain time period. Another benefit is to eliminate the effect of history and see the real alpha values during a given period of time.

$$\alpha_{Collector.cpp,A} = \frac{\sum \text{Consequential change count in A due to Collector.cpp}}{\sum \text{Changes in Collector.cpp}}$$

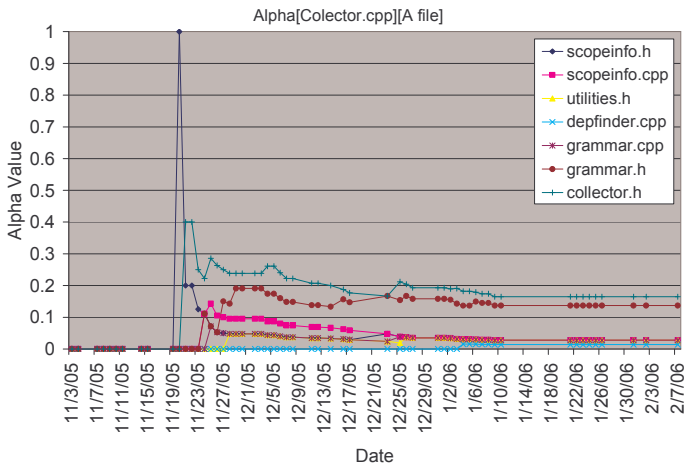


Figure 5.11 – Alpha value evaluation of Collector.cpp throughout the first release.

In this chart, we see how changes in Collector.cpp cause other files to change. As we saw previously, in Figure 5.8, alpha values between a header and its implementation file are larger than other alphas. Typical alpha values for non header/implementation pairs are low, of the order of 0.1, as we have been using, earlier in this research.

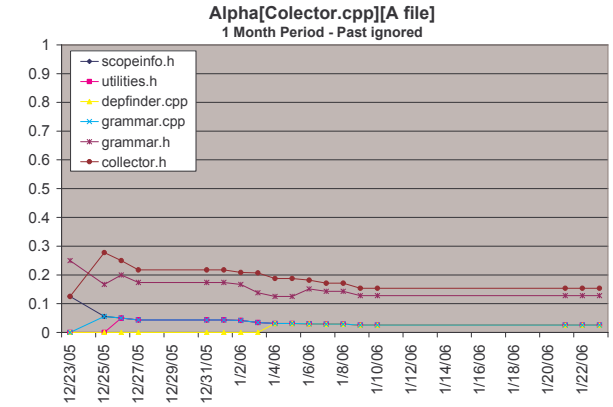


Figure 5.12 – Alpha value evaluation for 1 month period

Figure 5.12 shows Alpha values of Collector.cpp for 1 month. Similar to charts in Figures 5.9 and 5.10, earlier change history is totally disregarded here..

If there is no change in alpha values, (after January 10 in Figure 5.12) there is no change occurring in causing file. It could be an indication that file met its planned functionalities and is fulfilling its requirements or it could be the project manager's decision not to make changes until a particular project release to meet with schedule and budget.

5.8. Computing an Effective Single Alpha Value for a System

In this section, we derive a single $\alpha_{\text{Effective}}$ value, which represents the whole system. It is calculated by summing non-zero consequential change counts occurring in a time unit (day) and averaged over the number of changes occurring on that day, as described in the formula below. All the changes are cumulative. In the formula, m is the number of files in a project, n_i is number of files to which file i causes consequential change.

$$\alpha_{\text{Effective}} = \frac{\sum_{File\,i}^{m} \sum_{File\,j}^{n_i} \text{Consequential change}_{ij}}{\sum^{m} \text{Change}_i} \; if \; \text{Consequential change}_{ij} \neq 0$$

Thus, $\alpha_{\text{Effective}}$ is the relative frequency of consequential changes with respect to all changes.

Figure 5.13 and Figure 5.14 show the evolution of $\alpha_{\text{Effective}}$ in some time interval. Figure 5.13 covers the time frame starting from the beginning of the project up to the time of the project's first release. At the beginning of the project, $\alpha_{\text{Effective}}$ value is low, since very few consequential changes have occurred. Figure 5.14 covers one month time-slice of the development, ignoring the number of changes at the beginning. During the time period covered, files began to use other developed file's services. Due to use of these services, we found latent errors or conceptual misunderstandings that required change. Testing uncovered bugs and the need for additionally functionality. As a result, consequential changes occurred, so larger alpha values are observed in Figure 5.14. We chose to ignore the earlier history here, so that we could clearly see alpha values that represent current activity, not weighted by past change inactivity.

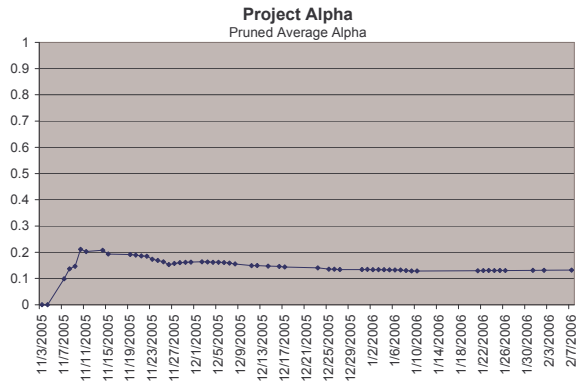


Figure 5.13 – $\alpha_{\text{Effective}}$ evaluation throughout the first release.

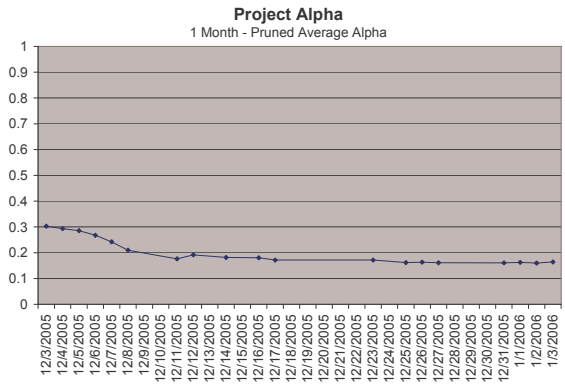


Figure 5.14 – $\alpha_{\text{Effective}}$ evaluation for one-month period.

5.9. Risk Analysis with Measured Alpha Values

Figures below show the product risk [60] of files in our experimental project (DepAnal) calculated using measured alpha values. Figure 5.3 above is also product risk, but the alpha value is an estimated constant, 0.1. Risk values in Figure 5.15 are obtained by individually calculated alpha values, meaning each α_{ij} value (Change Impact Value) used is measured using change history.

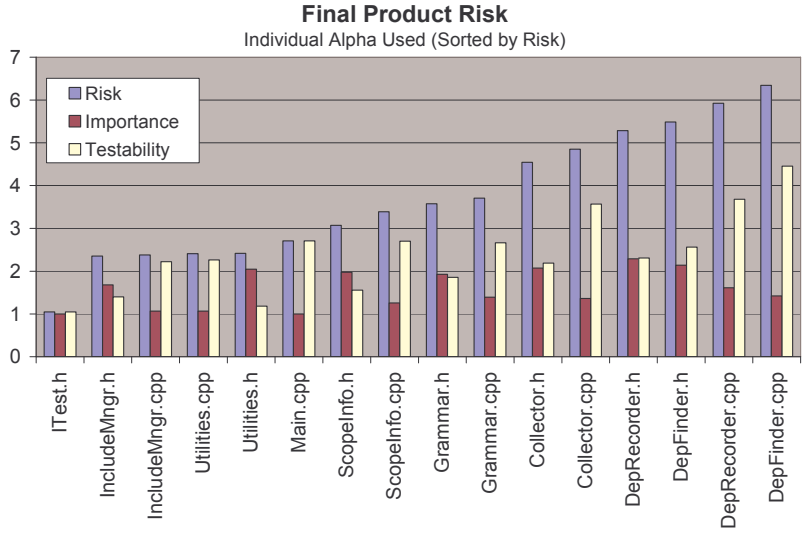


Figure 5.15 – Product Risk with individually calculated alpha

We know that the risk values obtained with individually calculated alpha values are the most precise ones. On the other hand, we would wonder that if we use single alpha value for overall analysis, how closely single alpha represents real risk values. As we saw in previous paragraph, comparison of actual versus estimated constant value was not very precise. For that purpose we calculated the risk values by using $\alpha_{\text{Effective}}$, which is a single alpha measured using change history as in the formula, above in section 5.7. Figure 5.16 shows the risk values calculated measured project alpha value.

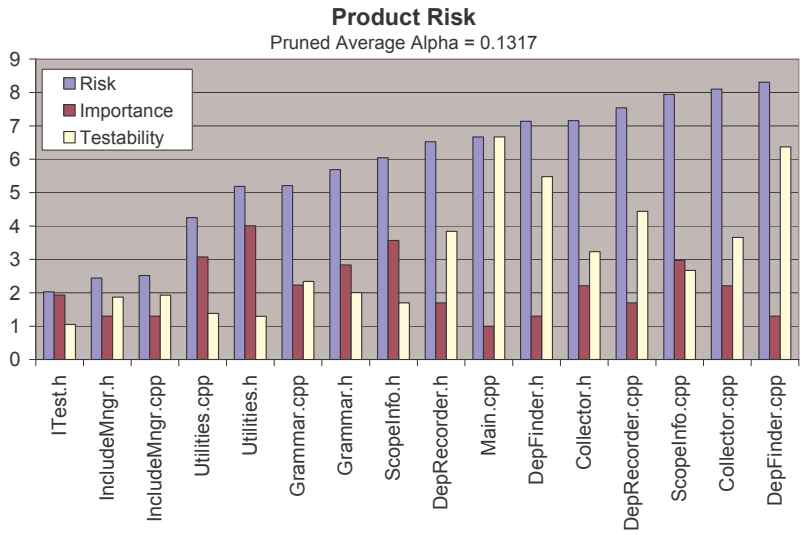


Figure 5.16 – Product Risk using $\alpha_{\text{Effective}}$

As we see in Table 5.3 and Figure 5.16, more than 62% of the files stay in the same order or at most two-places moved as in Figure 5.15. Therefore, the order difference between using single measured $\alpha_{\text{Effective}}$ and individual alphas can be disregarded for this experiment.

Ordering change with regard to individually calculated alpha				
	Estimated Alpha		Calculated Effective Alpha	
Order Change	Count	Percentage	Count	Percentage
Same Place	6	37.50%	7	43.75%
1 Space Moved	2	12.50%	1	6.25%
2 Space Moved	2	12.50%	3	18.75%
3 Space Moved	2	12.50%	1	6.25%
4 Space Moved	3	18.75%	3	18.75%
5 Space Moved	0		1	6.25%
6 Space Moved	1	6.25%	0	

Table 5.3 – Change in risk ordering of files calculated by measured $\alpha_{\text{Effective}}$ and estimated alpha, compared to risk calculated by measured individual alphas.

The effort spent for obtaining individual alpha calculation is not negligible. If alpha calculation is automated, this will be great help to obtain precise risk values, which is an interesting future research area.

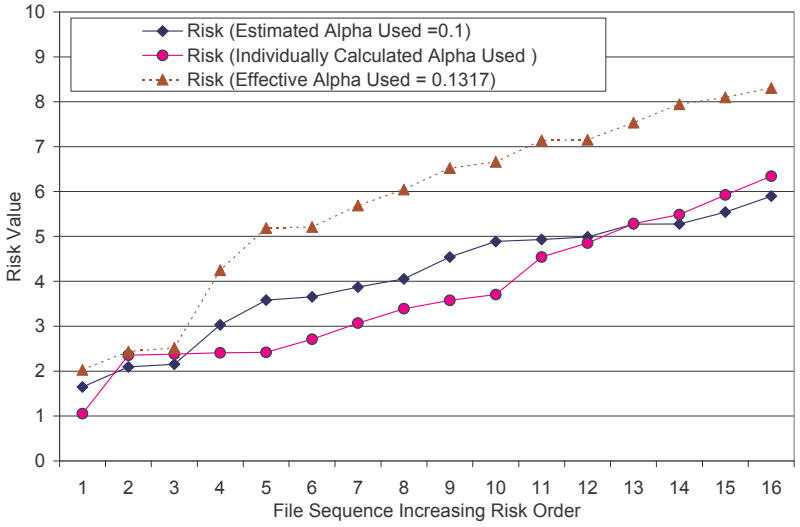


Figure 5.17 – Comparison of outcome of Product Risk with alpha variance

Figure 5.17 shows the comparison of product risk results. Of the three methods of evaluating product risk, the individually measured risk is the most accurate representation, since it is based on real measured change propagation, just as the model uses it. We expected the effective alpha model to approximate the same risk values, but in fact, our ad-hoc estimation of a

constant alpha gave better risk value results. But, the important issue is the ordering of files by risk, as we wish to use the model to decide on which files to focus our corrective actions. From this point of view, using a single effective alpha gives quite reasonable results, as indicated in Table 5.3. There we see that approximately 40 percent of the files keep the same position, compared to individually measured alphas, and 75 percent of the files moved no more than 3 places.

Since the ad-hoc estimation also performed well, with respect to file order, we are lead to believe that there may be an alternate process for computing a single effect alpha that may, in fact, be better. We have not discovered that yet, and so this is a topic for future research.

We believe that our process for measuring and using individual alpha values, is important and useful for evaluating product risk, and can be of very real benefit to project managers and developers.

5.10. Contributions of this Study

The contribution of this research can be summarized as follows. An experimental design was developed, which can be applied to other software development projects, to assess their change impact factor (CIF). Measured CIF values enable one to obtain a precise measure of product risk. This study provides a unique definition of change impact model, which is based on file to file change history. One of the aims of this research is to help large-scale software developers analyze the current standing of their software's structural quality. That is why granularity of CIF plays an important role. Another contribution is the empirical study and its quantitative results for CIF calculation and risk prediction. For example, we witness the evolution of alpha values over a project's lifetime. Additionally, we observe how closely a single project alpha can estimate product risk value, compared with individually measured alphas.

5.11. Concluding Comments

Change in software is an essential part of software development and maintenance. Estimating a proposed change's effect on the later phases of the development helps project managers and developers with decision-making, and predicting future progress. During development, on some occasions, speedy solutions are necessary to meet project schedules. These sudden changes may cure local problems at that time, but in the future, may cause significant quality defects [68]. For example, software can acquire a tendency to consequential changes, or unintended dependencies can arise. Management of change is achieved by being able to estimate the impact of changes; this study serves mainly to support that need. Calculated risk profiles disclose each file's vulnerability to external changes, as well as project's overall vulnerability. Software managers can use these charts to monitor and control the change process. Understanding impacts of a change is one of the methods for guarding against software quality degradation.

Calibrating change impact factor parameter values for a project from change history is applicable to any software development project. These quantitative measurements are superior to estimated alphas. High change impact values are not a desirable property of a file or a project. If a file is inclined to change due to external changes this increases effort required for implementing changes. As a result, it increases bug fix time and new feature implementation time. Knowing the system's sensitivity to change and estimating the effect of a change enables controlled and well-planned change activity.

Using change history enables us to:

- Understand degree of connectedness between source files,
- Provide controlled change activity.
- Monitor software quality
- Understand the evolution of change impact value (CIF) over a project life time

- Determine the quality of software via CIF; high CIF indicates low quality or immature software project.
- Alpha values help to predict the role of change of a file and the project, consequently this helps during;
    - Decision making.
    - Effort estimation.
    - Project scheduling

Chapter 6

System Structure - Simulating Constructive Change

In this chapter, we examine the affect of changes we may make to improve the structure of systems analyzed with the help of DepAnal and DepView. These changes are simulated using our generated structural metrics, by making changes to the discovered dependency data. We are not redesigning Mozilla, for example. That would be impractical. Instead, we are simulating the effects of changes like removing global variables and inserting interfaces between components with high fan-in and their many clients. We also have a more concrete example in our analysis tool, DepAnal, for which we have two versions. We redesigned this relatively small project to help us evaluate the prediction process[34].

Simulation is not perfectly accurate – it provides estimation and in some cases upper bounds on the improvements that can be made. However, it is critically important that a software manager or architect can estimate the affects that proposed changes may make, before embarking

[34] We also undertook this redesign in order to experimentally evaluate the Risk Model alpha factors, for a real project.

on an expensive and time-consuming redesign. For example, we have previously shown that the GKGFX library of Mozilla has serious structural problems. Because it consists of 598 files, it is very desirable to investigate potential improvement before investing in a lot of changes. This estimation is the subject of this chapter.

Dependencies between files are determined by declaration or definition of types, global functions, and global data, as indicated section 1.5.1. We show in this chapter that it is useful to make distinctions between the types of dependencies between files. One reason for doing this is that only dependencies based on simple types and global function usage can be manipulated without breaking code, simply by rearranging code packages. Type to type and global function dependency type can be optimized by moving types or global functions from file to file subject to maximum number of types/global functions per file. This may help, for example, to reduce the size of a strong component that contains these files.

Introduction of a non-constant global data definition always introduces mutual dependencies between every file that accesses that datum. These invoking and defining files, by default, depend upon each other. For instance when a global variable changes (e.g. rename) both files will be affected. If a global datum is renamed or deleted, every file that cites the global variable needs to be updated accordingly.

In addition, the affects of dependencies on types can be reduced and mutual couplings can be diminished by using interfaces to bind components together rather than binding to concrete types. This does not eliminate a dependency, but it reduces the likelihood that a change in a component represented by an interface will affect a component that uses the interface. This is so, because the interface hides all the implementation details of the component that supports the interface. The following sections explore candidate improvement techniques.

6.1. Eliminating Global Variables

Non static global variables are accessible from anywhere in the program. A non-constant global variable has the potential to cause large mutual dependency file sets. We would like to

observe what we may gain by elimination of them, and what we gain by assuming the global variable as constant. Removing global variables is simulated by simply removing their entries in the dependency table. This does not account for finding ways to communicate the same information by less dangerous means. As a result, we are finding an upper bound on the possible benefit. When we put back the needed parts of the communication, some of the dependencies will also come back. In the following, we investigate how much of this potential can actually be realized.

In this section, we explored the affect of a specific change - removing global variables - on GKGFX Library and MFC project's structural quality.

6.1.1 Analysis of GKGFX Library of Mozilla

Here, we explore the Mozilla GKGFX library to discover the change in dependency structure by eliminating global variables. Figure 6.1 shows a dependency view of the GKGFX library with the original dependency analysis on the left, and after removing global object dependencies on the right. We see decrease in size of one of the largest strong components from 60 to 27.

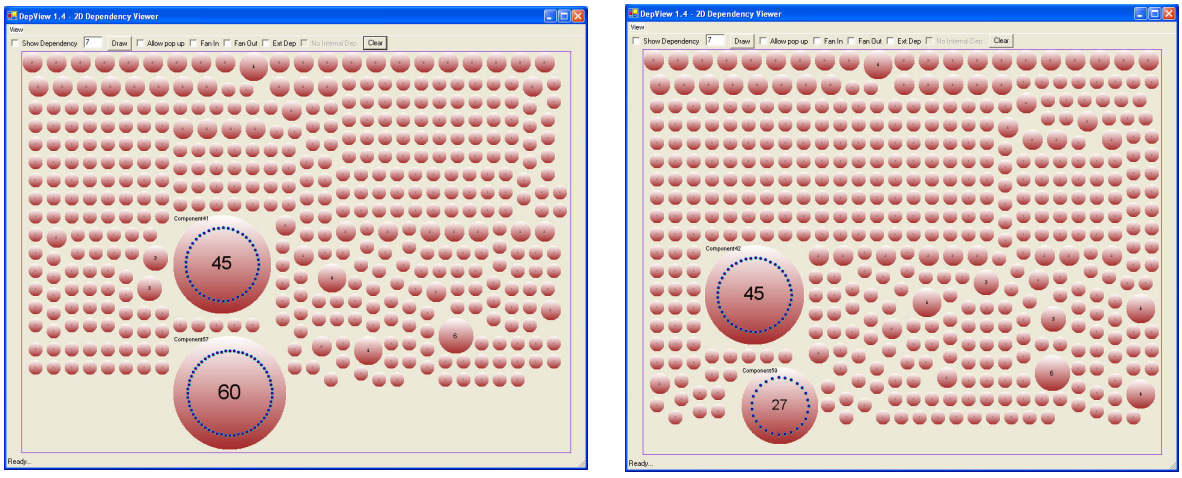


Figure 6.1 – Components of GKGFX Library, on the right after removing global object dependencies

As we know, each circle represents a strong component, and the number on each circle shows how many files are in that strong component. When we remove dependencies caused by global variables, component #57 on the left in Figure 6.1 is broken into pieces, and largest remaining piece has 27 files on the right titled component #59. . This indicates that use of global variables should be avoided as much as possible[35] [63]. And, also indicates that use of global variables increases systems complexity and strong component size.

Component #41 on the left and component 42 on the right in Figure 6.1 have the same member files constructed with global functions and type invocations. When we further analyze this component individually considering only type dependencies, that is, by removing global function dependencies, we get a strong component broken into smaller size as shown in Figure 6.2.

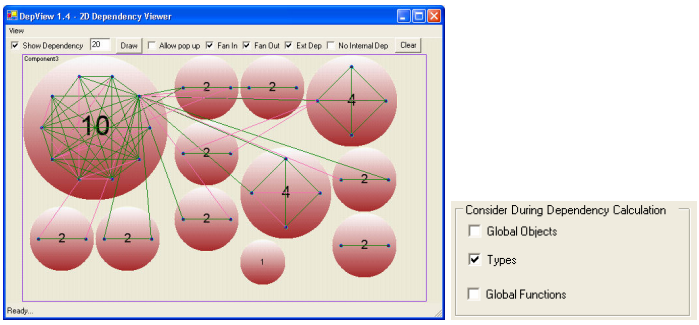
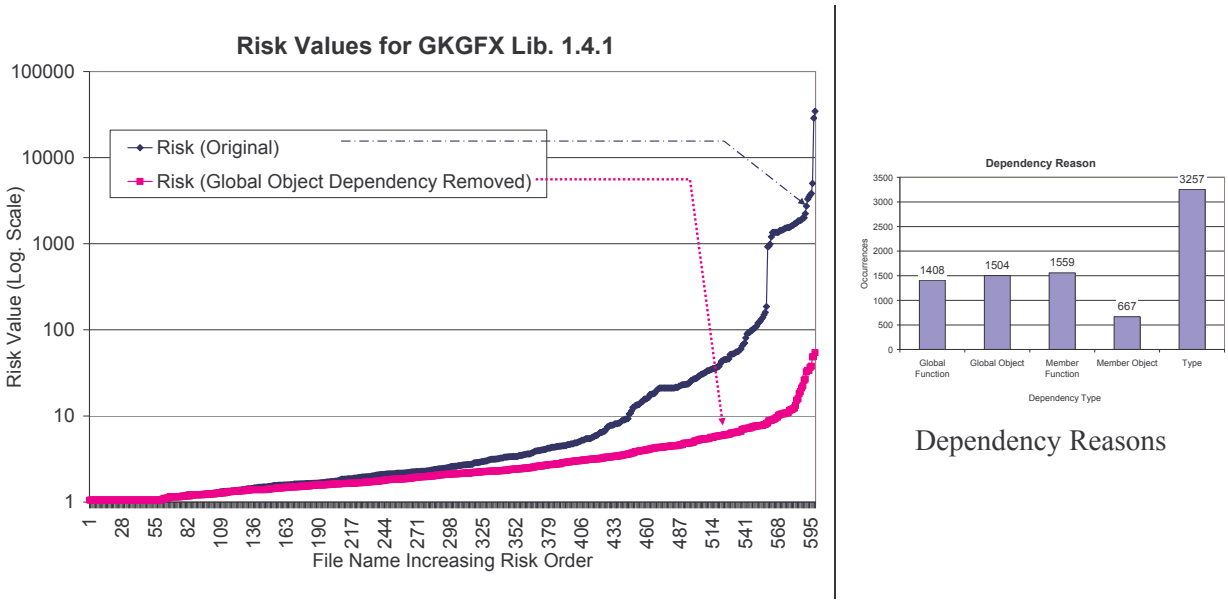


Figure 6.2 – Analysis of the component with size 45 in Figure 6.1

As it is seen in Figure 6.2, removing global function dependencies causes the strong component with 45 files to be sliced into smaller parts. This shows that dependencies on global functions are also causing large mutual dependencies.

Figure 6.3 shows logarithmic scale risk values of GKGFX Library before and after removing global object dependency, we see a large reduction in risk values.

[35]. This is "common wisdom" in the development community [63].



Dependency Reasons

Figure 6.3 – Product Risk for GKGFX Lib, simulation of global obj. dep. removal

As it is shown at Figure 6.1, after removing global variable dependencies, we see the strong component size has shrunk by about 55 percent. However, this made a big impact on maximum risk as in seen Table 6.1, where max risk reduced from 34619.89 to 54.35.

	With global objects	Without global objects
Maximum Risk	34619.89	54.35
Average Risk	10.87	3.65
Max. Strong Comp. Size	60	45

Table 6.1 – GKGFX risk values

All the files with high risk are members of strong components [58]. This also demonstrates that risk analysis is providing useful information.

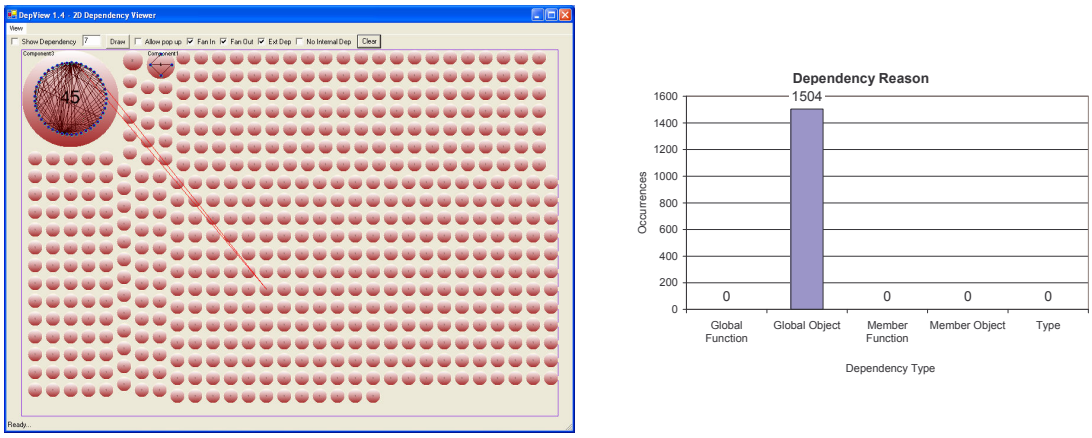


Figure 6.4 – Shown dependencies caused by only global objects for GKGFX, two-way.

In Figure 6.4, the largest strong component, consisting of 45 files has internal dependencies just due to global data. Type and global function dependencies are not included to observe the affect of global data. One fact to remember is that non-constant global variable usage causes two-way dependencies; one from a declaring file to invoking file and the other from the invoking file to declaring file. Two-way communication causes immediate circular dependency, since a change in a global object affects both. There are 1504 calls that occur due to global variables in this scenario. Between groups of two files, there can be several dependency links via a single global variable.

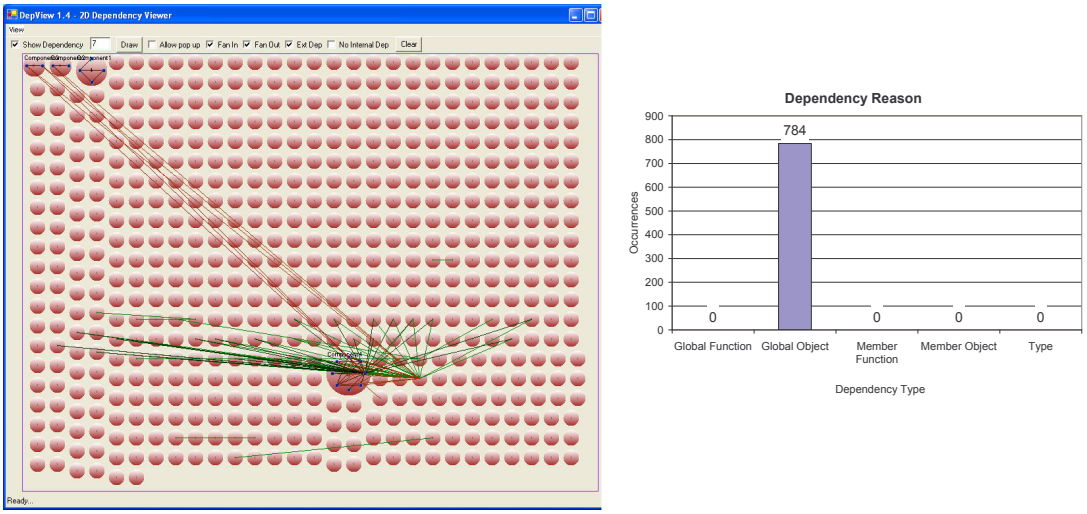


Figure 6.5 – Dependencies of GKGFX, caused by global objects only, one-way.

After treating all the used global object as constant, we obtain the Figure 6.5. Also, notice that number of dependency occurrences reduced from 1504 to 784. This indicates non-constant global data is causing almost twofold more dependency density than a constant one. In this experiment, constant global object causes only one-way dependency, which is from an invoking file to the declaring file, as in the case of type or global function invocation. We see that the component with 45 files disappeared; as a result, many external dependencies appeared. This demonstrates non-constant global variable can increase dependency complexity significantly.

6.1.2 Analysis of MFC

Using our tools, we analyzed Microsoft Foundation Class files, and find dependencies among MFC files with and without global objects. “Microsoft Foundation Classes, or MFC, is a Microsoft library that wraps portions of the Windows API in C++ classes, forming an application

framework for developing Windows programs. Classes are defined for many of the handle-managed Windows objects and also for predefined Windows and common controls".[36]

We explored the MFC library to discover whether similar behavior would be observed. Interestingly, MFC files do not use global objects as heavily as GKGFX library, as is seen in Figure 6.6. Removing global variable dependency for MFC does not change the structural quality radically, as is shown in the analysis below.

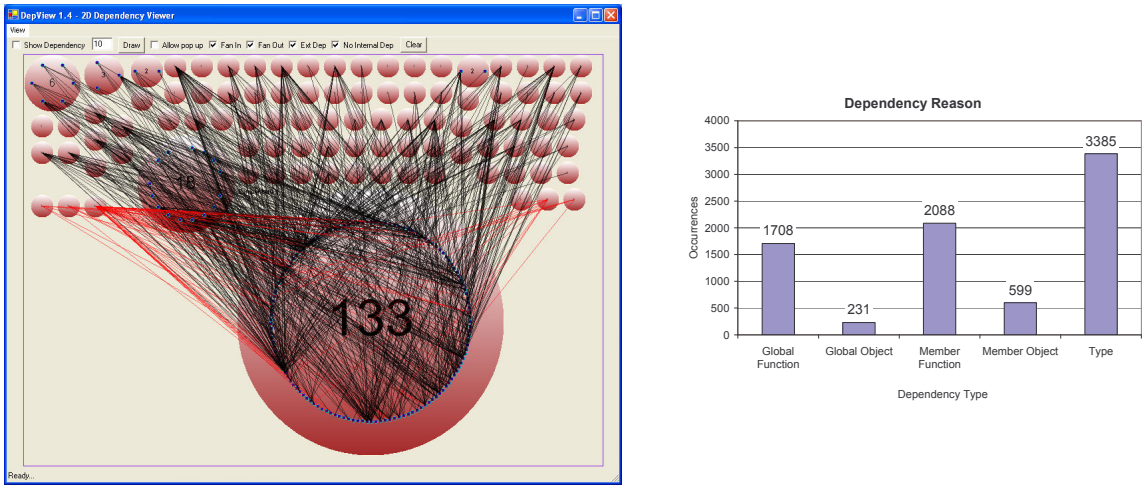


Figure 6.6 – MFC Dependency reason and external dependencies of Component #6

Figure 6.6 shows the results of a first run considering global variable dependencies; we see one big component consisting of 133 files. When we exclude dependencies caused by global variables, we obtained Figure 6.7. This shows that the largest component size is reduced from 133 to 128. Interestingly files involved in the large strong component are not due to global object dependencies. Even though it did not change the strong component size much, global object dependency caused a big reduction in overall risk value. This indicates when strong component size gets larger; each addition of a new file to a strong component can significantly increase risk.

[36] http://en.wikipedia.org/wiki/Microsoft_Foundation_Classes.

	With global objects	Without global objects
Max. Risk	30542.54	9131.51
Avrg. Risk	1497.85	447.61
Max Strong Comp. Size (files)	133	128

Table 6.2 – MFC risk values

The MFC library exhibits Test Risk singularity for alpha very close to 0.07, due to dense dependency in strong component. This means that its code would be extraordinarily sensitive to change. When we reduced alpha to 0.06, the singularity was avoided, leading to the analysis here. Since MFC is a viable commercial product, we believe that typical values for alpha in commercial systems may well be significantly less than 0.1, due to the expertise of its developers.

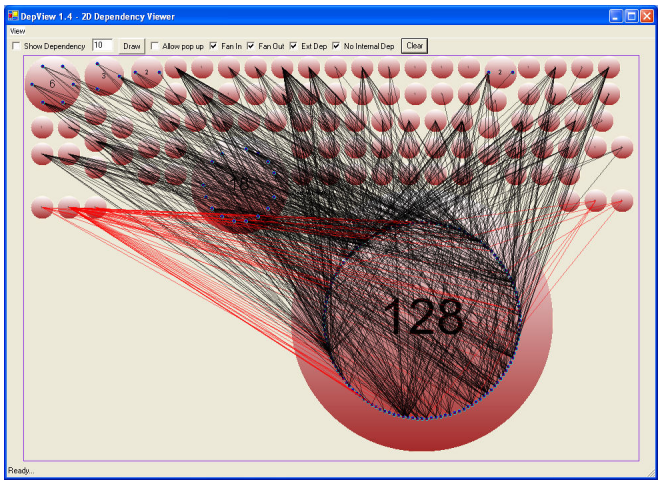


Figure 6.7 – MFC, Internal - External dependencies of Component #6

The highest risk value, shown in Table 6.2, reduced from 30542.54 to 93.83. Figure 6.8 shows risk values with, and without, global object dependencies. After removing global object dependency, the file order changed slightly. If this library's strong component were constructed by heavy use of global variables, we would see the order of files would change significantly, as well as risk magnitude.

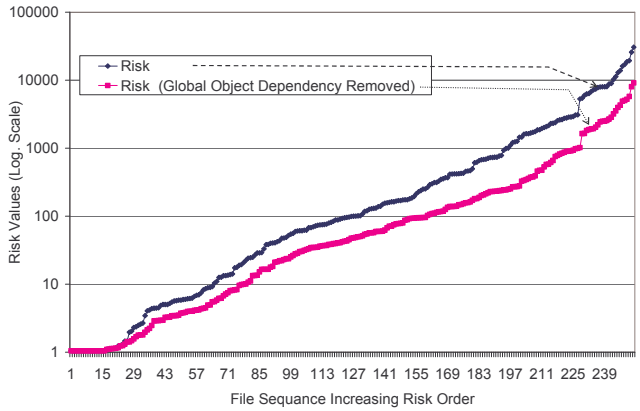


Figure 6.8 – Risk values for files in MFC Library, before and after global object dependency removal

The data for Figure 6.8 was calculated with the simplifying assumption that α is constant. To improve the accuracy of modeling, each project will want to calibrate their organization's software model using a process we have developed as part of this research in Chapter 5. Alpha measures the probability that a change in one component causes changes in another component that depends on it. One would expect lower values of alpha for components with solid, robust designs.

6.2. Insertion of Interfaces and Factories

One technique that can be used to reduce the size of large strong components is to use interfaces and object factories to move dependencies from volatile implementations to immutable interfaces. Object factories are needed to avoid reintroducing dependencies on implementation, removed by binding to interfaces. This technique helps us decompose large strong components into a series of smaller components. We show, below, that that has a net, and often quite significant, improving effect on software quality. Deciding where to place interfaces is part of our current research, but initial results are quite encouraging. One approach that looks very promising is based on a partitioning process developed by another member of our research group.

In order to investigate the affect of using interfaces instead of some specific concrete classes to reduce dependency risk for the system, we simulated the use of interfaces by reducing

the predicted potential for change factor, alpha. Using GKGFX library, which has 598 source files total, we first applied risk analysis with alpha of 0.1 for all files as in Figure 6.9, and highest risk value for this analysis came out 34619.89.

To simulate the affect of inserting interfaces, we selected 24 files with high fan-in values (fan-in value 14 and greater) and for these files, we reduced the alpha value to 0.01, simulating the insertion of interfaces. As we know, alpha value indicates the potential for change of one file due to a change in a file on which it depends. If it is smaller, change in one file is less likely to cause change in other files. We picked high fan-in files, since these files are being used heavily by other files, and these high fan-in files play a key role in introducing interdependency on others.

After introducing these changes and applying our risk analysis to the same Mozilla library with these interface simulations, we found the highest risk value was reduced from 34619.89 to 3151.80, as shown in Figure 6.9.

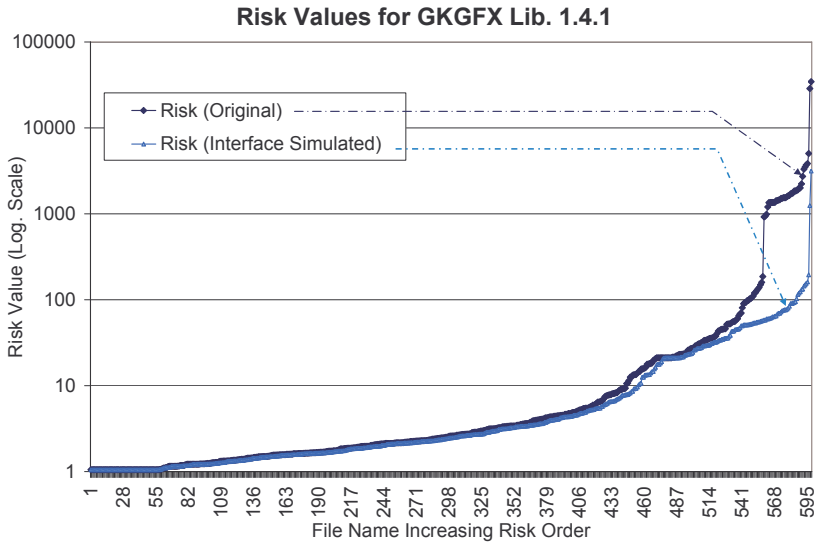


Figure 6.9 – Risk Analysis of GKGFX Library

In summary, modifying a relatively small number of files (24 files out of 598) to use interfaces, resulted in the highest risk value being reduced from 34619.89 to 3151.80. This

indicates the use of interfaces will increase testability of the system and reduce importance of files; consequently, risk will be reduced.

By introducing interfaces, we can eliminate some structural problems in the software system. Another, critically important outcome is that without resorting to semantic analysis of code, but simply simulating the addition of interfaces we see the results in terms of system risk. This task would be extraordinarily difficult without the file rank based risk analysis we have developed. Thus, without adding a large analysis load on developers, but just using simulation, we can locate files to focus on, which contribute the highest benefit to remedy structural problems.

6.3. Redesign and System Quality

Beside simulation, we redesigned our relatively small DepAnal tool. Below we compare analysis results of the new design DepAnal program with the old one. This will let us gauge whether the risk model will be useful for smaller systems as well as large systems like Mozilla. Figure 6.10 and Figure 6.11 show the risk values of files that construct the new and the old DepAnal respectively.

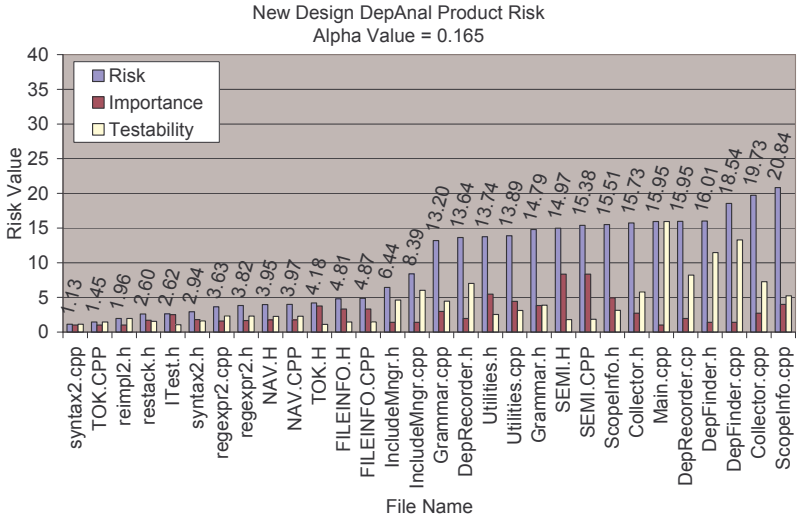


Figure 6.10 – Analysis of Risk of new design DepAnal, sorted by increasing risk order

Figure 6.10 and Figure 6.11 show not only changed files for the projects, but also the files that are reused, without change. During computation of risk values, an alpha value of 0.165 was used for both analysis. This alpha value is calculated by carefully recording change information of the new design DepAnal. Details are given in Chapter 5. In the new design, we see the largest risk value is 20.84, in old one it was 35.34 as shown Figure 6.11. Knowing all the implementation details, as the developers of the both projects, before seeing these Product Risk charts, our choice would definitely be the new design. Since it has better dependency structure, well defined job partitioning, and less complex implementation. Addition of new features and understanding the whole the system is much simpler than the old one. Having stated that, the risk values magnitudes express the same fact. This is matching what we are expecting as designer of the both projects.

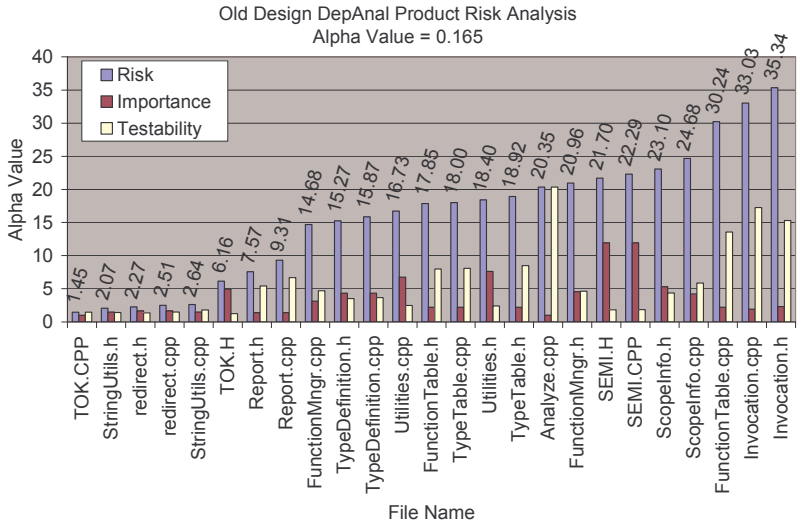


Figure 6.11 – Analysis of Risk of old design DepAnal, sorted by increasing risk order

6.3.1 Discussion of Old DepAnal Design

In the old design DepAnal, almost half of the files have risk value more than 20, which are relatively high values comparing with the new design. Even though both programs are intended to accomplish the same mission, by comparing risk values, we can come to some conclusions about their implementation quality.

In Figure 6.11, risk values (sorted in increasing order) are shown, along with importance and testability for each of the files of old design DepAnal. Invocation.h has the highest risk, due to large fan-out, size of 13 out of 25 files. It is almost impossible to keep in mind all 13 files, for what reason Invocation.h is accessing them, and it is very easy to forget the job of each of these files. Another issue is testing Invocation.h, we need to make sure all 13 files are tested thoroughly before testing Invocation.h. One another reason, why Invocation.h has high risk is that it is a member of a strong component of size of four with dense communication (Figure 6.13, below on page 137).

When we look at the lower left of chart 6.11, we see Tok.cpp to Report.cpp. All these files have low risk values compared to the other files involved in the project. This is so because all these files have relatively less complex implementation, and they depend on only a few number of files in order to accomplish their tasks. Tok.cpp even has importance of one, this points out no other file is using its services; this also indicates it is either a test stub or main executive. In this case, we know Analyze.cpp is the main executive so Tok.cpp is a test stub. Analyze.cpp is one of the high-risk files, since it depends on 17 other files. This is certainly not a desired property, given that it is directly communicating with too many files. That indicates it is taking too many responsibilities; therefore, abstraction is not partitioned well. As developers of DepAnal, we know the inner quality of the project, and this chart is confirming our knowledge.

6.3.2 Comparing Old vs. New DepAnal in Detail

In this section, we compare the dependency structure and product risk values of each DepAnal. In Figure 6.12 and Figure 6.13, we see strong components expanded into their individual files, each dot represents a dependency relationship. For any dot, the file vertically below it (x) depends upon the file horizontally to its left, on the ordinate (y). Any dot under the diagonal indicates the existence of a strong component. Each rectangle represents the strong component. The table, on the right of the figures shows the topologically sorted files.

This diagram discloses the quality of the project's dependency structure. Dots closer to diagonal are better, indicating local communication between the files. The number of dots in each square shows how densely these strong component members communicate with each other.

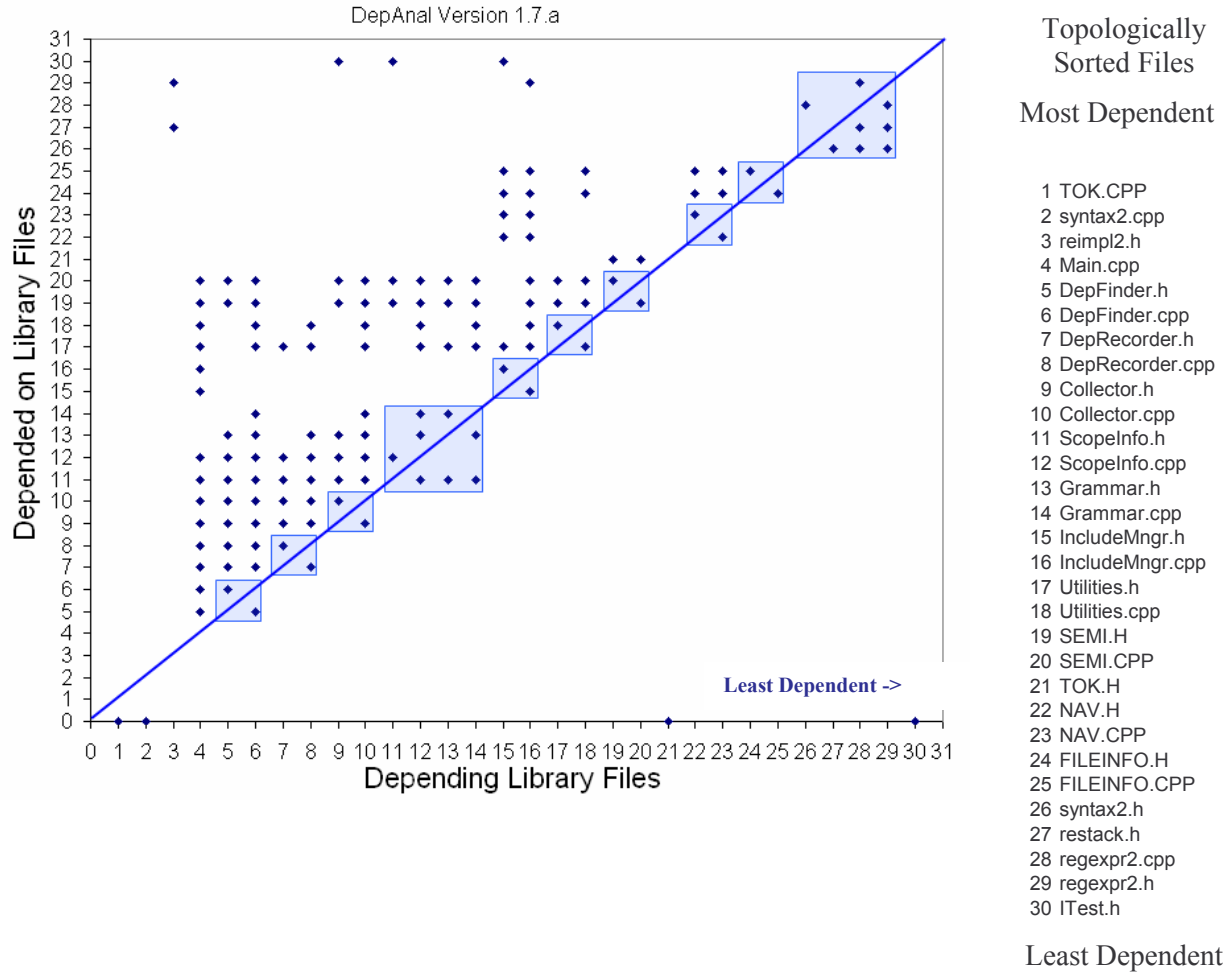


Figure 6.12 – Expansion of Strong Components – New Design DepAnal Ver. 1.7.a

Figure 6.12 show this project is nicely packaged with small strong components and most files communicates with nearby files. The number of dependency paths in this strong component is less than the strong component in Figure 6.13, e.g., the old design. One of the strong components in old design has four files but 10 dependency paths. There are more than twice as many (6-extra) dependencies between members of its files. This reduces the flexibility of the

files for change, since a change may initiate other changes to occur due to strong connections between them.

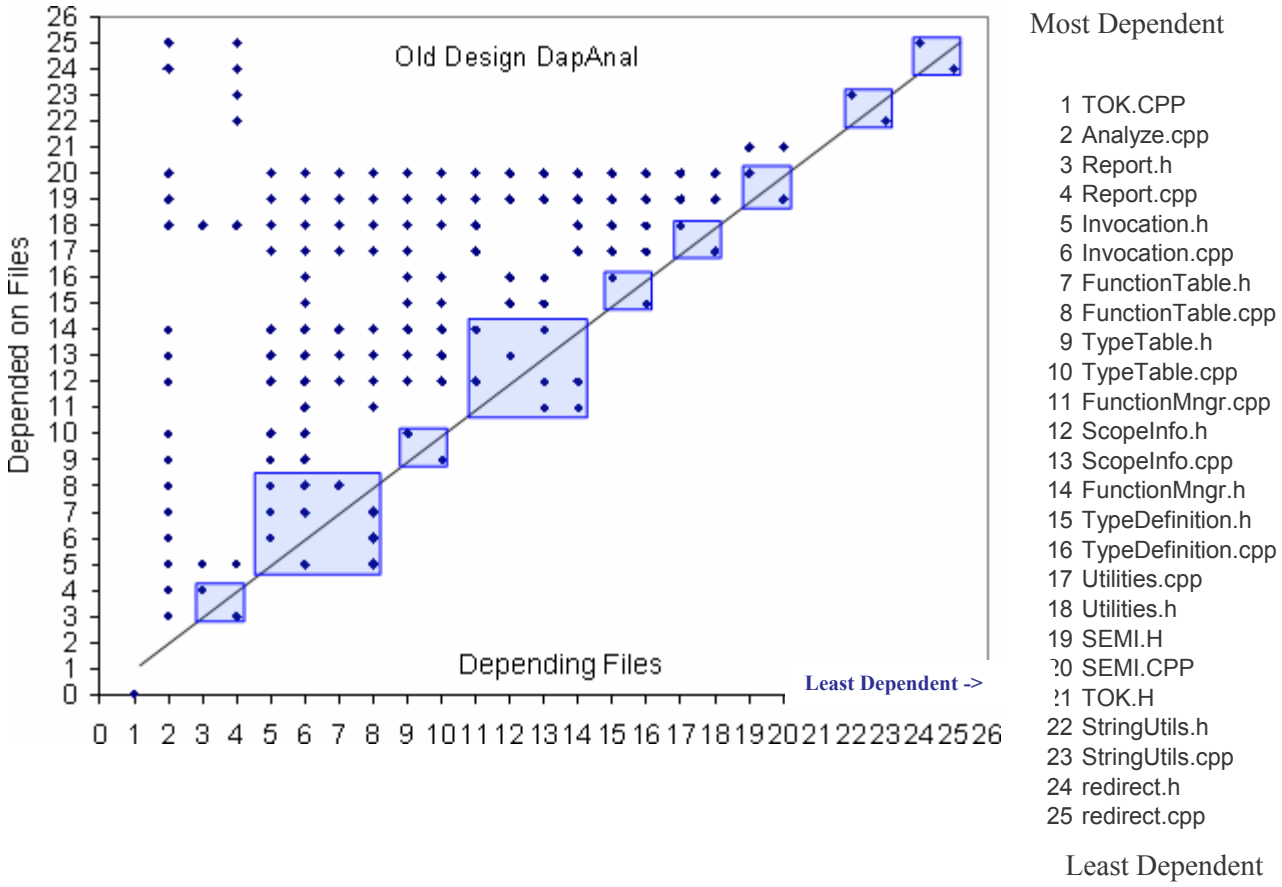


Figure 6.13 – Expansion of strong components, old design DepAnal

Using Figure 6.12 and Figure 6.13, we can obtain the data listed in Table 6.3, showing new design has a significantly better static structure.

Category Name	Old Design	New Design
Number of dots above diagonal	124	119
Sum of distance to diagonal (Dots above)	844	719
Number of dots below diagonal	15	18
Sum of distance to diagonal (Dots below)	22	25

Table 6.3 – Comparing structural quality of old and new design DepAnal

The sum of the distance of all the dependency dots in old design, from the diagonal, is 866 (844+22), and in new design this number is 744 (719+25). We noticed that the number of dependency dots is almost equal; stressing that, from a risk point of view, local communication is more desirable than more non-local, as is shown in Figure 6.14.

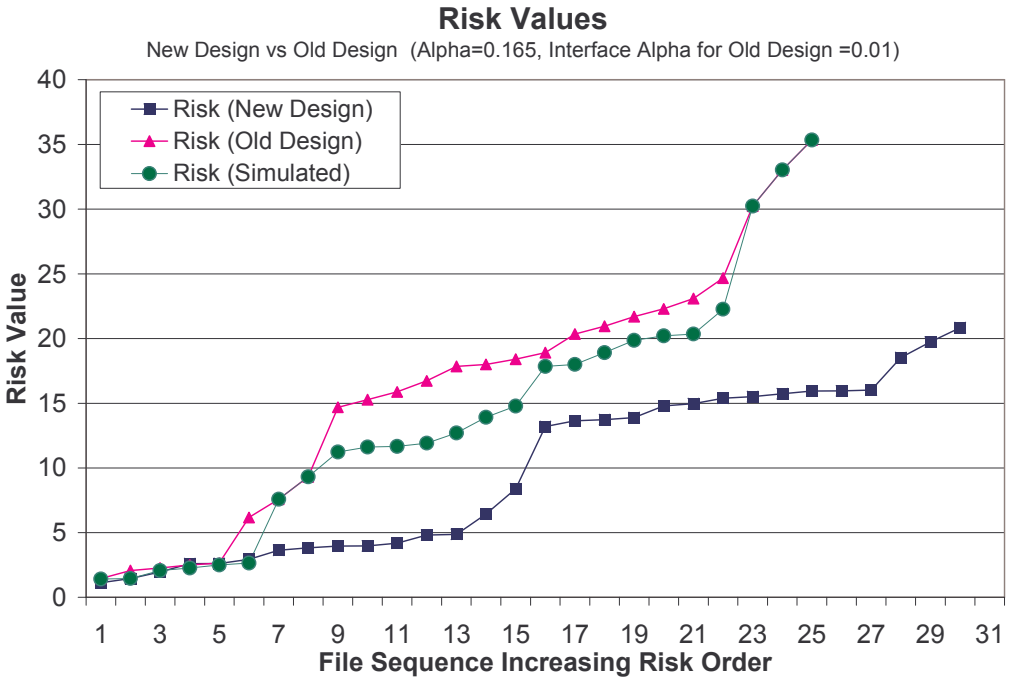


Figure 6.14 – Product Risk Values, Old Design vs. New Design DepAnal

Figure 6.14, above, shows the simulation of interfaces for old design DepAnal to compare with new design. We assumed that files with fan-in size 10 or greater are good candidates for use of interfaces. This resulted in five interfaces to the system with 25 files. For the simulated interfaces, alpha values are taken as 0.01. As we see, new design has better risk values than simulated one.

Before the analysis, we were expecting the chart below, which is matching the result above. Renovating a project on top of an existing architecture does not give as much flexibility as

redesign. Moreover, after gaining experience from the previous design, the designer can avoid making the same mistakes.

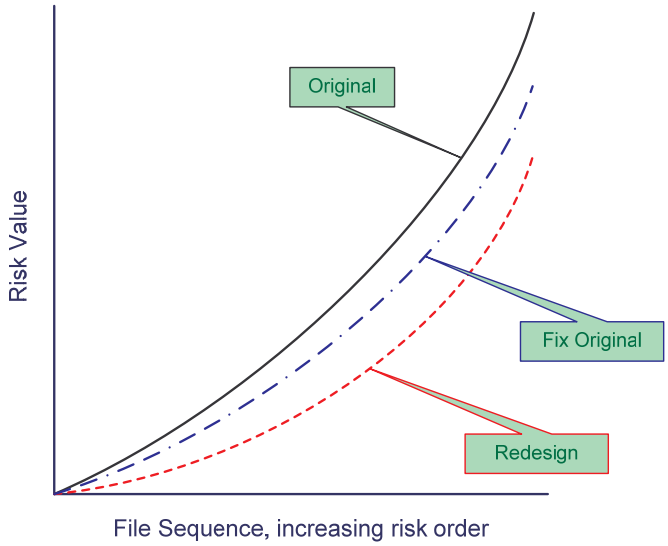
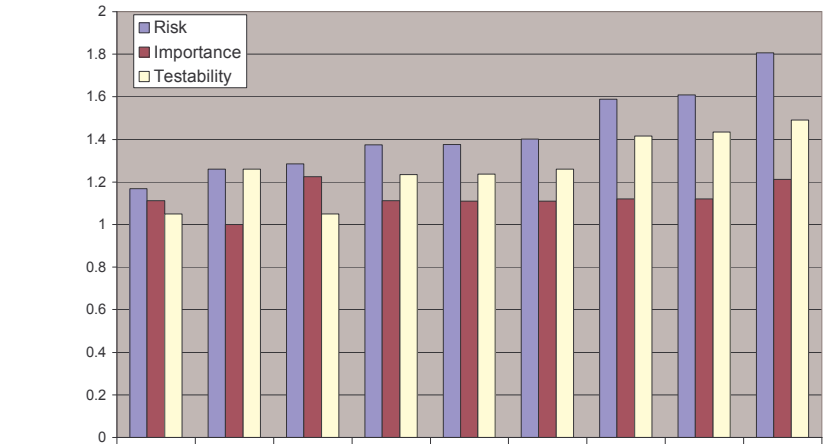


Figure 6.15 – Expected risk values before and after constructive changes.

In Figure 6.15, we see three curves above; the one at the top is the original risk before constructive steps are taken. In the middle, after applying suggested fixes, we see what would be the overall risk values relative to the original risk. Finally, the bottom one is the expected risk values after completely redesigning the project.

6.4. Strong Component and Product Risk

Strong components are the groups of files, which cannot function with the absence of any other member file, due to mutual dependencies between them. Traditional testing order is to test first files, which depend on no other files, and then test those files, which depend only on already tested files. However, in the case of mutual dependencies there is no such order. Therefore, we have to treat strong components as a unit during testing; this also reduces testability and increases importance of each file and consequently the risk of files gets higher. Let us examine these statements with the basic example below.



	D.h	main.cpp	E.h	F.h	G.h	H.h	B.h	C.h	A.h
Risk	1.1677	1.2609	1.2854	1.3741	1.3756	1.401	1.5877	1.6085	1.8061
Importance	1.1121	1	1.2242	1.1121	1.1112	1.1111	1.1211	1.1211	1.2111
Testability	1.05	1.2609	1.05	1.2356	1.2379	1.2609	1.4162	1.4347	1.4912

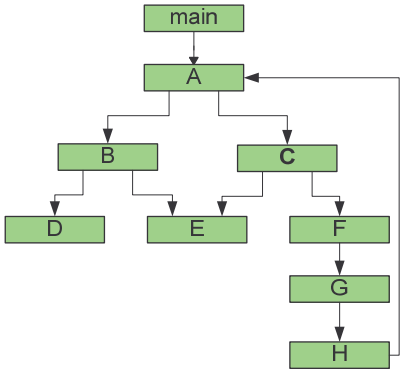


Figure 6.16 – Dependency graph and its corresponding risk chart.

In this example, files do not have any complex implementation; they have empty classes and function bodies. Therefore, testability value of files is only affected by the dependency relationship between files. Here, we take the α value of 0.1. For importance values, without analysis, we may think file H should be the most important file among all the files, since it is at the lowest level and many files depend on it. Additionally it is a member of strong component. Nevertheless, file E has the highest importance value, since strong component with 5 members and additionally file B depends on file E. Implicitly file H is also dependent on file E. All these factors causes file E to have high importance.

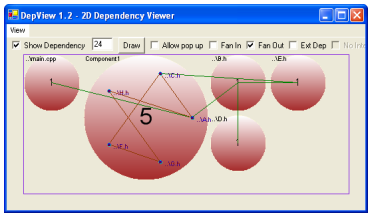


Figure 6.17 – DepView for basic project above.

For the testability, by looking at the dependency graph we might jump to the conclusion that the main file should be the hardest file to test, because it depends on all the other files

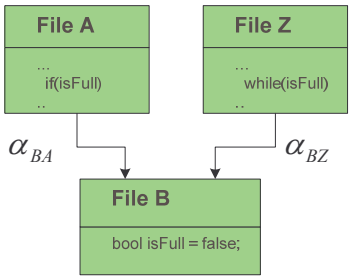
directly or indirectly. However, we see that file A has the highest testability score. There are several reasons for this to be so. Both the main file and file A depend on the same files, since file A depends on itself because of the feedback call from file H. File A has two direct dependencies (B and C), but the main file has only one direct dependency, therefore change propagation caused from lower level files encounter file A before reaching the main file.

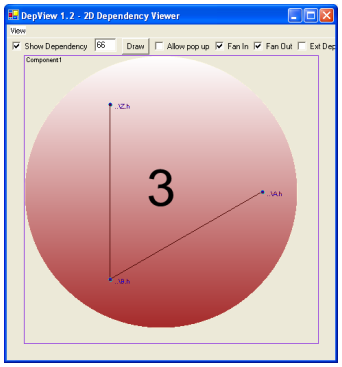
One interesting fact is that file B also has high testability value, which is higher than the members of the strong component. As stated above, we take the α value 0.1; this means changes are not likely to affect files with distant indirect dependencies. In AppendixA.4, at page 170 we took α value 0.9 to better observe the relationship between α and strong components. In that the appendix we see that members of strong components, and files directly depending on strong components, always have high testability values and consequently high-risk values. In the case above, due to low α value and high direct dependency, B has higher risk value than F, G and H, which are members of the strong component.

6.5. Global Variable and α

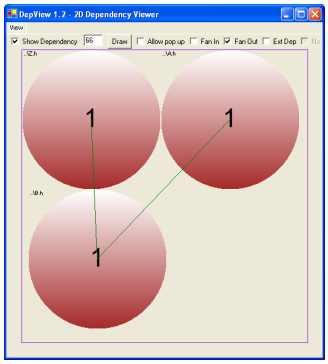
Global variables provide an easy way to communicate with other files, but brings risk along with convenience. If a global variable is non constant, it not only makes the using file depend on declaring one, but also makes declaring one depend on the using one. This two-way dependency always causes mutually depended file groups. All the analysis in this research is based on static dependency relationships, therefore whether a global variable actually remains constant or non-constant does not change the dependency structure, provided that it can change. However, we would like to observe how this dynamic property may affect dependency structure. Figure 6.18 shows file B declaring a global variable, named “isFull” and both file A and Z are using that global variable in their internal calculation. In the Figure 6.18, file A and file Z do not depend on each other, since they are not using services of each other. On the other hand, the global variable declared in file B is non-constant, this causes file A to depend on file Z via file B

and vice versa. Since, if any of the file changes the value of that global variable, it will affect the flow of control in other invoking files.





Non-constant global variable



Constant global variable

Figure 6.18 – Global variable dependency and alpha value (α)

Any change to the value of a global variable affects all the depending files, as seen in Figure 6.18, global variable in file B causes a strong component with size 3 in the middle. For testing, a strong component must be treated as a unit. The larger a strong component becomes, the more difficult it is to adequately test. Mutual dependency increases likelihood of cascading changes - it causes α_{BA} and α_{BZ} values to be large (close to 1). For example, it is close to 1, due to the fact that if file B renames or deletes the global variable, depended files no longer can function. Altering the value of global variable in B may change decisions made in using files, which is another reason that global variables cause strong dependency relationships between files, and consequently high α value.

6.6. Summary

We explored the effect of different dependency types over the static structure of a large system, Mozilla 1.4.1, without a detailed understanding of its internal semantics. We simulated possible changes to observe the affect on its static structure via our research algorithms and tools. Simulation is not perfectly accurate – it provides estimates and in some cases upper bounds on the improvements that can be made. However, it is critically important that a software manager or

architect can estimate the affects that proposed changes may make, before embarking on an expensive and time-consuming redesign.

Chapter 7

Conclusions and Future Work

In this dissertation, we have studied systems that are so large that no one person can understand their entire semantics. After examining existing open-source projects and code developed for this research, we concluded that static dependency structure is an important characteristic of a system. It provides an abstraction over all the details within a complex code base, needed to understand the current state of a large project. Due to the size of systems on which we focus, we needed to develop methods that do not require semantic analysis. We developed structural quality assessment metrics to uncover potential or existing structural problems in software, and we developed tools that can analyze large software systems and provide quantitative and qualitative results. For example, to compute the dependency structure of the entire Mozilla project, version 1.4.1, consisting of 6193 files, required about 4 hours of running time, on a moderately powerful desktop machine. In addition, we simulated constructive changes to observe improvements in the structure of the systems we analyzed.

In this chapter, we summarize our results and discuss methods that we use, and give directions for further studies.

7.1. Study Results and Contributions

After analysis of the source code for Mozilla, Microsoft Foundation Classes, and our own DepAnal, we are able to uncover potential or existing structural problems in software from source code. We developed a novel source file ranking algorithm using notions of product risk, importance, and testability of a file. Risk rank of a file is determined by both its position in the static dependency structure and also on its internal implementation quality. This ranking process is useful while managing development and maintenance of large systems, indicating where attention should be focused to improve testability, and reduce product risk.

The granularity of analysis output is important, in order to provide information that is simple enough to understand, but detailed enough to provide a critical analysis of the system. For that reason, we conducted our analysis at the file level, but based on internal types, functions, and their dependencies. Additionally, this dependency information can be obtained automatically, which is a key attribute for this analysis.

One of our goals is to provide immediate feedback to software developers about the state of a software development project. We are not depicting type-to-type or function-to-function dependencies, although our tool examines them, for the reason that we are dealing with large numbers of source files. Every file can define several types, and this will increase the volume of information to the extent that it would be difficult to draw conclusions about it. For instance, one of the libraries of Mozilla project, GKGFX has declaration of 6423 types (e.g. class, struct), within its 598 files as shown in Figure 7.1. Visualizing or trying to draw conclusion with the relationship of 6423 items would be arduous, and almost impossible to comprehend.

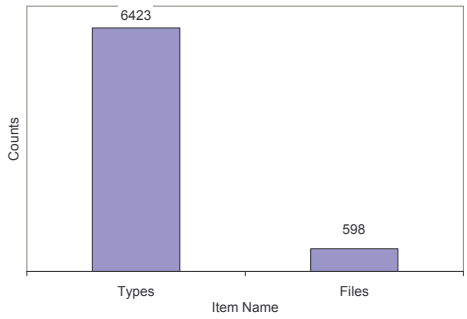


Figure 7.1 – GKGFX Library item counts

Another reason for exploring dependencies at the file level is that files are widely accepted as units for software testing, configuration management, maintenance, reuse etc. The granularity of dependency among files lets us observe qualitative characteristics of large software systems with adequate detail.

We have also presented a novel metric that indexes software components according to their potential for reuse. This reusability index provides help to developers by ranking source code in existing systems, based on its place in the structure of the system and its internal metrics. This enables developers to evaluate a file for reuse before looking at its code. This reusability value could be useful information for classical software configuration management systems (SCM).

Without the help of analysis tools, it is difficult to understand a large project, evaluate its quality, and track progress effectively. Therefore, we developed several tools capable of analyzing large-scale software[37]. These tools enable a software manager to monitor a software project rapidly without waiting for documentation files to be produced, obtaining structural quality information directly from source code. Two major applications developed for this research are DepAnal and DepView. Automated dependency analyzer, DepAnal scans each file, collecting user defined types, functions and variables along with their invocations, to find

[37] The first version of our DepAnal tool required more than 24 hour to analyze all the 6193 files in Mozilla, 1.4.1. The redesigned tool now accomplishes that in about 4 hours.

dependencies among the source files. Dependency viewer, DepView displays 2D graphical interactive file relationships together with strong components. Beside these tools, there are other helper tools, which are developed to assist software people to understand their large code base and determine where corrective action is needed, and to continually monitor the progress of the system.

After identifying potential dependency problems, we also explore effects of different dependency types over the dependency structure of a large system, without a detailed understanding of its internal semantics. We simulated possible changes to observe the affects on static structure. This way we can estimate how much improvement we can obtain if we apply any of the techniques before engaging time and effort making changes. Subsequently, with the generated algorithms and tools, we monitored the improvements to observed structural defects.

Controlled implementation of change is achievable by being able to estimate impact of changes. One of our research goals was to understand the impact of a change in a software source file to other source files. Since change impact value will depend, in part, on how well both files are implemented, e.g. on the skill of the project's architect and developers, we created an experiment to estimate change impact factors; describe its application; and show measured results of the change impact factors. All these processes were carefully recorded and documented so that others, following our example, can calibrate their change impact factor values for a project based on its change history. Before the experiment, we were expecting that the risk values obtained using calculated alpha values would be in the neighborhood of risk values obtained using constant alpha values. We observed that this was not the case - the real risk analysis should be accompanied with a semi-automated change chain recording mechanism, to obtain real change impact values of each file.

Table 7.1 shows summary of main contributions and results of our study that we discussed above and in detail in the dissertation.

1	We developed a source file ranking algorithm using notions of product risk, importance, and testability of file. - This identifies components that need individual attention and suggests possible ways to avoid impending problems before they become chronic. - The product risk model predicts that as the density of dependency relations increases in strong components of the dependency graph, risk factor grows and becomes unbounded at critical densities. - We applied the model to a library from a real open-source project where the model predicted that most of the development risk is in about 10% of the library files (a very useful result, probably unknown to the projects' developers).
2	We introduced a model that indexes software components according to their potential for reuse.
3	We designed and conducted an experiment to investigate the impact of change in one file on other files, in terms of consequential changes they require. The results of this are a record of the probability of change over time, during the software development.
4	We designed and developed tools implementing these algorithms and methods. The importance of this is that they are capable of analyzing very large sets of files in reasonable time, e.g., all of Mozilla (6193 files) in about four hours.

Table 7.1 – Results and contributions

In Table 7.2, we listed the results obtained as consequence of our study. These are byproduct of the main results listed above in Table 7.1.

1	The study enables a software manager to monitor a software project rapidly without waiting for documentation files to be produced, directly obtaining structural quality information from source code. Such as: ▪ Visualize the static structure, as a whole ▪ Visualize its web of dependencies ▪ Determine mutually dependent files and size of the mutually dependent file set,
2	Our empirical study demonstrated that useful information about significant problems can be identified, in both large and small systems, without a detailed knowledge of the entire code base.
3	We applied our tools on industrial projects to observe and report on the applicability and quality of estimation and to evaluate the overall effectiveness of our approaches.
4	We can determine how well the project is packaged into modules, and provide guidance about how a project can improve that.

Table 7.2 – Consequential results of the study

Some of our research in progress has yielded other tentative results. In Table 7.3 – we show those early results of the work that will continue.

1	We explored the effect of different dependency types on the structure of large systems without requiring a detailed understanding of its internal semantics. These analyses expose possible corrective procedures and our tools support simulation of improvements in observed structural defects, when these corrections are made.
2	Statistical analysis of file properties versus change potential from Mozilla change database.

Table 7.3 – Initial results of work that will continue later

7.2. Future Work

One interesting new thread of research would develop semi-automated methods to calibrate change impact factors for specific projects using smart configuration management tools. This is important because the alpha factors are determined in part by the nature of the code being developed, and also by the skill of the developers and management team.

It would be useful to explore the optimization of system structure by repackaging "location tolerant" dependencies. Employing "penalty weights" to decide which types and global functions to move and where they should be moved. One reason for doing this is that only dependencies based on simple type usage can be manipulated without breaking code, simply by rearranging code packages – an interesting partitioning problem. Exploring insertion of interfaces and class factories to loosen coupling between concrete classes has the potential to decrease the probability of long chains of consequential change. Code in files communicate via couplings to other classes, either through interfaces, or by binding directly to implementation details. But change inside of any file is less likely to affect others that bind to an interface instead of its concrete implementation.

It is expensive to parse all the source code to observe an effect of elimination of an object, type or function to overall dependency structure. For this reason, it would be timesaving to store all the declarations and invocations of types, global functions and objects for later to use. This way we can observe changes in dependency structure, and strong component size, when we remove one or a group of types, or global variables, or global functions. This will enable developers to see the contribution of any type, function or global variable over dependency structure. Moreover, it will quickly demonstrate the effects of any dependency elimination and what causes particular dependencies without lengthy analysis over the entire project.

It would be useful to have a run time dependency analyzer, which is always active during development. If a developer introduces a strong component it would pop up in the developer's computer environment, indicating that you have just created a strong component with the size of some number. It can display how many people are using the current file a developer is working on. When somebody just refers to his file, at that time he will be notified about it. This add in feature in development environment would provide immediate feedback to developers, so that they may enable quick decisions about a design strategy.

Appendix

In this section, we present our initial studies that served to shape the research focus of this thesis, and we present supportive inquiry showing the need of this study and additional basic samples demonstrating efficiency and preciseness of the analysis results.

A.1.Relationship between Code Metrics and Change History

In this section, we study relationships between code metrics and change count histories for a large project. The analysis is file based. That is, we compute a variety of metrics for each source code file in several large libraries from the open-source Mozilla project, and relate them to the number of cumulative changes for each of those files in several builds. We use files because changes are recorded for files in the data we examined; and because files are the units of configuration management in large software projects.

Others, German [29] and [30], Huntley [31], have examined open-source project data but we've found no modeling of the reliability of metrics to measure potential for change, as reported here. Graves, et. al. [32] analyzed relationships between change metrics and predicted faults, using data from a telephone switching system. Our focus is on modeling change history using code metrics.

Appendix

We chose Mozilla because it is large[38], accessible, and has provided a wealth of change data in its source code repository (CVS) database. Most of the data presented here is drawn from the Windows build of Mozilla [12], for several releases, spaced approximately one year apart.

We downloaded the CVS archive for these releases and, using make tools provided by the Mozilla project, built Windows executables for one specific release, 1.4.1. During the build, we captured all the files being compiled and used that file set for our analysis of variation of metrics with time[39].

In the next section, we show how changes and the total number of files have grown over the life of the entire[40] Mozilla project. We then show how changes, number of files, estimated defect counts, and metric values, have varied over the four releases and one CVS check-out we analyzed.

In the third section of Metric Analysis, we analyze four libraries and the entire Windows build for the 1.4.1 release, 10 October 2003. The analysis uses Multiple Linear Regression (MLR) [33][34] to model production of changes as functions of the metrics set, described below. The results are evaluated in terms of resulting t-test and adjusted R-square statistics. In all analyses, we find statistically significant relationships between some of the metrics used and change history, for each of the four libraries, and for the entire Windows build. The results show, however, that not all of the change is related to these metrics and is dominated by two of them, fan-out and total lines of code. As we look further, we see fan out is the best predictor even better than files size, other have all secondary effect.

---

[38] There are 6193 source code files in the Windows build for version 1.4.1

[39] Mozilla code management is based on libraries that contain files for all supported platforms. We used the output of building 1.4.1 for Windows to identify the source code for the Windows build, and used those files present in each of the other builds to analyze changes in average metric values with time.

[40] Entire means all files and all changes for all of the platforms supported by Mozilla.

A.1.1.*Project Wide Measure of Size and Change*

All change and defect data were extracted from the CVS change logs of the Mozilla project. Figure 1.1 shows the number of files and cumulative changes over the lifetime of the entire Mozilla project, as of 10 September 2004. The latest data consisted of 36,800 files, of which, 14,210 are C/C++ source code[41]. Mozilla CVS captures changes, with and without bug numbers. In the metric analysis, we count only changes for source code files with an associated bug number in its change log. The numbers for the entire Mozilla project are shown in Table 1.1.

Changes	All Files	Source Code Files
All Changes	502,753	305,844
Changes with Bug numbers	255,904	156,903

Table 1.1 – Cumulative Change Counts, 10 September 2004

To quantify defects, we counted the number of unique bug numbers for all the changes against a specified file. The results for defects were far less statistically significant than for change counts. We observed aggregate file check-ins with shared logs, which may inflate estimated defect counts. So, we believe our construction of defect counts, based on this data, is not very accurate, but find no other data in the CVS change logs or Bugzilla database used by the Mozilla team, that relates to defects. For that reason, we will not consider defects further in this study, other than to show variation of our definition over several releases in Figure 1.5.

For large projects, like Mozilla, the volume of files and their rate of change make it virtually impossible for one person to understand the structure and semantics of the entire project. It is crucial that the tools we develop to analyze systems of this size do not require detailed understanding of all the lines of code in the project, or even the lines of code in a single build for a single platform.

---

[41]. Files that are not source code include files with extensions ini, mk, idl, html, css etc.

Appendix

Our goal is to develop tools that program managers and architects can use to understand when a large program is developing problems in its code base. One measure of these problems is the volatility of its changes. Making changes are expensive in schedule time and staffing costs. Managing change is an essential part of managing budget and schedule.

We show, in Figure 1.1, the history of cumulative change, number of source files in the code base, and, in Figure 1.2, the number of changes per file, as a function of time over the entire history of the Mozilla project [35], starting on 28 March, 1998, as measured from the first CVS check-in, through our last data extraction on 10 September, 2004.

The level of effort required to manage hundreds of thousands of changes, as experienced in the Mozilla code base since the project began six years ago, would be difficult to sustain for any project, but especially for projects that do not follow the open-source model with large numbers of volunteer developers. Project managers need mechanisms to predict and control change. We show, in the following, that changes experienced by the Mozilla code base are significantly related to only a few of the metrics analyzed.

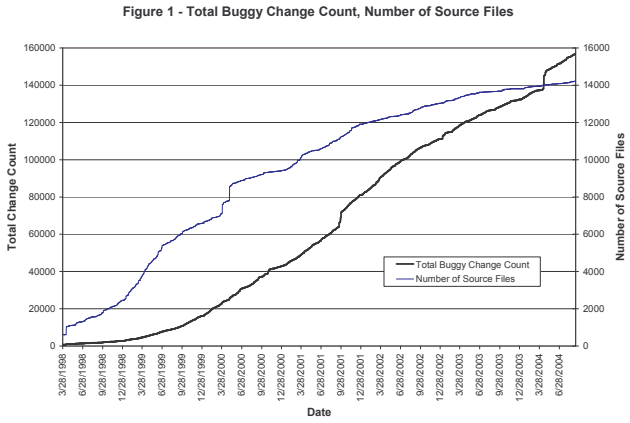


Figure 1.1 – Total buggy change count number of source files

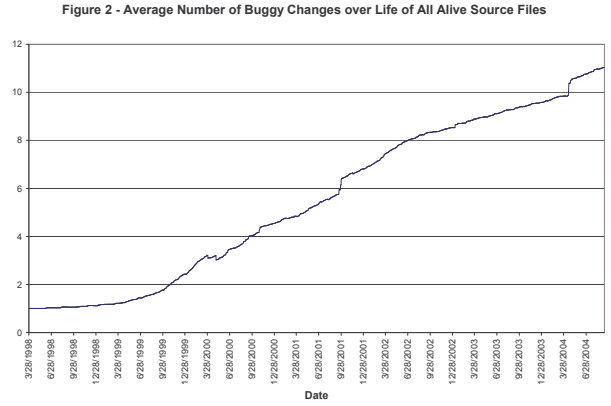


Figure 1.2 – Average number of buggy change of all alive source files.

A.1.2.*Metric Analysis*

For analysis of the relationship between various metrics and change counts we use only source files for the Windows platform build. There are 6,193 source code files in release 1.4.1 for Windows, for example, and we count the changes with bug numbers for those files.

Several of the metrics we examine, e.g., Fan-In, Fan-Out, and size of strong components (groups of mutually dependent files), are based on the dependency graph between files in a project [10]. This dependency graph captures static type and function calling dependencies. It is built using a dependency analyzer tool we developed based on a modest subset of the C/C++ language grammar. We also examine size and complexity metrics evaluated with a tool used by one of us for grading graduate software design project assignments[42]. We have also looked at maximum function cyclomatic complexity, maximum function size per file, average function size per file, and complexity per line of code, but settled on the metrics in Table 1.2 as being the best measures of those we examined.

The modeling tool used here is Multiple Linear Regression Analysis (MLR), which attempts to predict historical change counts as a linear combination of the metrics in Table 1.2

Fan-In: Number of files that depend on a given file.	Global declaration count per file (GblObjDec): The number of global data declarations in a specified file.

[42] We think of metrics global object declaration count, average cyclomatic complexity, and average function size per file, as measures of code quality, but have not demonstrated that they are associated with defect counts, so we avoid use of that term in the paper.

Fan-Out: Number of files a given file depends on.	Total lines of code (TLOC): The total lines in source file, including white space, declarations, executable code, and comments, e.g., every line in each function body, summed over all the functions in each file.
Instability[43]: I = Fan-Out / (Fan-Out + Fan-In)	Lifetime The number of days that the file has been under CVS control.
Size of strong component (SCSize): Number of files that have mutual dependencies with a given file. Every file in a strong component has a direct or indirect dependency on every other file in the component.	Average cyclomatic complexity[44] per file (AvgCC): The number of regions defined by the control flow graph of a function, e.g., one plus the number of loops and branches[45] per function, averaged over all the functions in each file.

Table 1.2 – Metrics used in this Analysis

Clearly, change counts are not synonymous with quality. A file with excellent quality may change because its requirements change or because the interface presented by some file on which it depends has changed. Also, a file with low quality may not change often because it is so big and complex that developers are reluctant to make any but the most urgent changes to its code.

[43] Similar to the class-based model of Martin [37].
[44] Our complexity measure is similar, but not identical, to the McCabe Cyclomatic Complexity metric.
[45] This includes continue, break, and goto statements.

However, change effort is directly related to a program's ability to meet its budget and schedule obligations [36]. It would be interesting to examine change effort directly, but the data available in Mozilla CVS does not support deriving effort, only change count, so we have used that information throughout this study.

A.1.3.*Analysis of Windows Build Releases*

In this section, we analyze five Mozilla builds for the Windows platform, separated by approximately one year, each.

	Release	Date
1	0.6	06 December 2000
2	0.9.7	20 December 2001
3	1.0.2	07 January 2003
4	1.4.1	10 October 2003
5	CVS Check Out	10 September 2004

Table 1.3 – Analyzed Mozilla Releases

First, we show the number of files, cumulative changes, and defects, for each build, in Figure 1.3 Figure 1.4 and Figure 1.5.

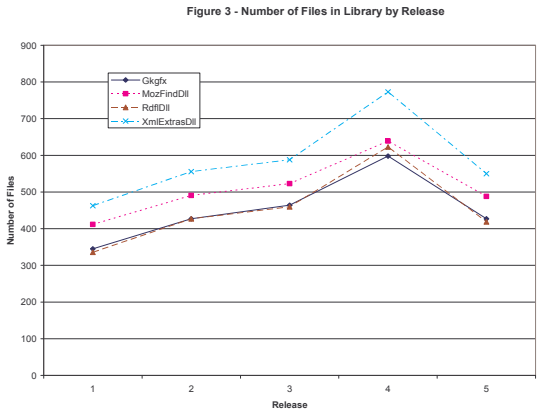


Figure 1.3 – Number of files in libraries by release

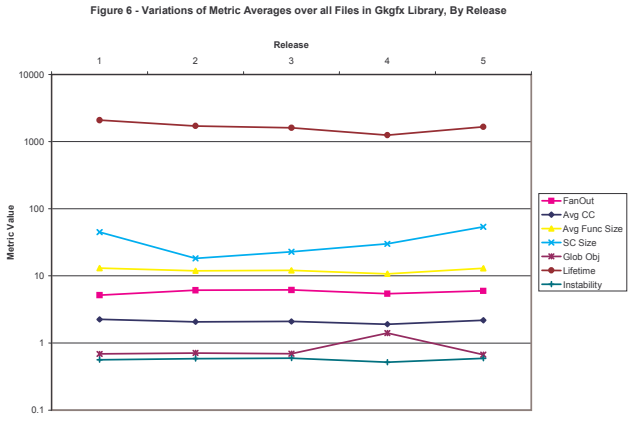


Figure 1.4 – Variations of Metric Averages over all Files in GKGFX Library, By Release

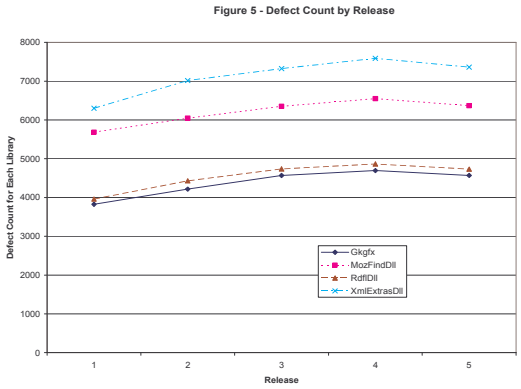


Figure 1.5 – Defect count by release

Figure 1.4 shows that metric values are fairly stable over the four years of code base evolution captured by these releases. It would be interesting to observe a project where these measures were used to direct corrective effort to see if corrections had a significant affect on their average values.

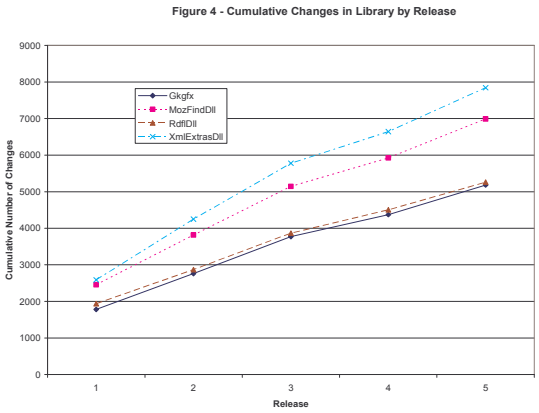


Figure 1.6 – Cumulative changes in library by release

A.1.4.*Some Techniques Used As Part of This Analysis*

Multiple Linear Regression [33] [34] is a widely used statistical data analysis technique to find relationships between several independent variables and one dependent variable. We used MLR analysis to model production of changes as functions of several possible metrics sets based on change data from Mozilla project. In the analysis, we try to predict relationships between software metrics as independent (known - because they are computed by our tools) variables and the number of changes experienced by the code as the dependent (unknown) variable. By using

change history and metric information, we try to model these relationships, and assess the model's prediction accuracy for estimation of future change based on past history.

The results are evaluated in terms of resulting t-test and adjusted R-square statistics. In all analyses [11], we try to find statistically significant relationships between some of the metrics used and change history. Figure 1.7, at page 152, shows predicted and actual changes for Mozilla's MozFindDll library, showing good correlation between predicted and actual values. The results are, however, incomplete, because the data was incomplete. To elaborate on this, for example, a file with high cyclomatic complexity is expected to require a lot of effort to change, since it is challenging to get it right the first time due to large numbers of control paths. However, in some cases the data unexpectedly shows a low number of changes. Mozilla only provides change count, but does not record how much effort is spent to perform those changes. We believe that effort information would be much more suitable than change count in this case. Unfortunately, that is not available from the Mozilla change logs.

Regression analysis has been applied in [40] [46] to find the answer to the parallel question "how internal software metrics relate to external software attributes". In these studies, fault-proneness was taken as dependent (unknown) variable.

A.1.5.*Multiple Linear Regression*

Our goal is to determine if the metrics, shown in Figure 1.4, are related to changes shown in Figure 1.6. In Table 1.4, we show a sample set of results from a Multiple Linear Regression Analysis (MLR) for the MozFindDll library. This models cumulative change for all files in this library, using MLR, as a function of the eight metrics shown. The Adjusted R Square statistic indicates that this model accounts for about 73 percent of the actual changes observed. The t statistic magnitudes greater than 2 indicate that the metrics Fan-out, Average Cyclomatic Complexity, and TLOC are statistically significant. Taking into account typical values for each

of these metrics and the coefficients from the model, we find that Fan-out and TLOC dominate predicted change.

In Figure 1.7, you will see plotted the actual changes, and the changes predicted by the linear regression model. We have sorted the file sequence by actual change value to make the plot easier to interpret. When the model predicts small change the actual changes tend to be small and when predicted changes are large the actual change tends to be large. The results are similar for each of the four libraries examined, as indicated by the plots in Figure 1.8 through Figure 1.10. When we analyze the entire Windows build, the Adjusted R-Square statistic and actual versus modeled change improves.

Table 1.4 – Results of Multiple Linear Regression, MozFindDll, Release 1.4.1

SUMMARY OUTPUT

Predicted and Actual Changes for Mozilla's MozFindDll Library
Release 1.4.1, 10 October 2003

Regression Statistics	
Multiple R	0.858457045
R Square	0.736948497
Adjusted R Square	0.731926034
Standard Error	9.752451239
Observations	428

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	8	111644.6583	13955.58229	146.7304964	1.9991E-116
Residual	419	39851.21786	95.11030517		
Total	427	151495.8762			

	Coefficients	*Standard Error*	*t Stat*	*P-value*
Intercept	2.630161	3.649069	0.720776	0.471449
FanIn	-0.001011	0.046063	-0.021958	0.982492
FanOut	0.949522	0.078886	12.036699	0.000000
AvgCC	-0.545876	0.165891	-3.290572	0.001084
SCSize	0.001222	0.003771	0.324012	0.746090
GObjDeclCount	0.023998	0.101890	0.235531	0.813912
TotalLOC	0.016478	0.001033	15.944396	0.000000
LifeOn_2003_10_10	0.000095	0.001898	0.050215	0.959975
Instability	-2.514524	1.726392	-1.456520	0.145998

In the correlation matrix, given in Table 1.5, we see that Fan-out is most strongly correlated with predicted change, and also to a lesser extent, correlated with TLOC and Average Cyclomatic Complexity. Predicted change most strongly correlates with TLOC, followed closely by Fan-out.

Table 1.5 – Correlation Matrix for MLR Model MozFindDll, Release 1.4.1

Predicted and Actual Changes for Mozilla's MozFindDll Library
Release 1.4.1, 10 October 2003

	FanIn	*FanOut*	*AvgCC*	*SCSize*	*GObjDeclC*	*TotalLOC*	*LifeOn_20(*	*Instability*	*Cumulative*
FanIn	1								
FanOut	0.071353	1							
AvgCC	-0.000963	0.39313	1						
SCSize	0.099422	0.423766	0.306617	1					
GObjDeclCount	0.059384	0.21411	0.025957	0.139319	1				
TotalLOC	0.176208	0.588954	0.593417	0.341359	0.140996	1			
LifeOn_2003_10_10	0.183703	-0.090313	0.067559	0.030067	0.008245	0.147321	1		
Instability	-0.372109	0.413485	0.197215	0.166101	0.025796	0.143086	-0.242959	1	
Cumulative Change	0.153469	0.731609	0.412359	0.357888	0.180698	0.784982	0.053095	0.201202	1

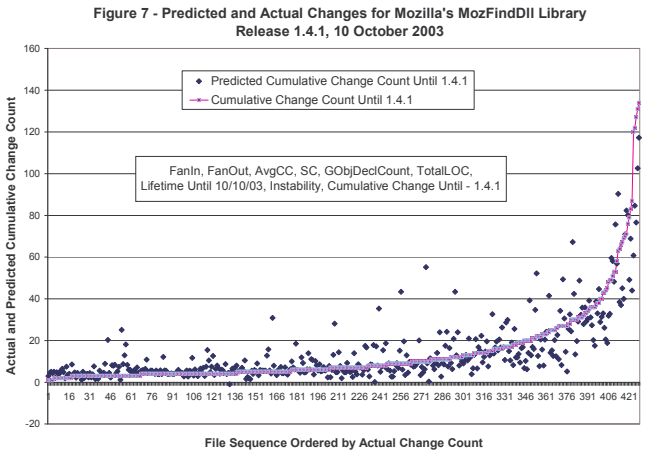


Figure 1.7 – Predicted and actual changes for Mozilla's MozFindDll library

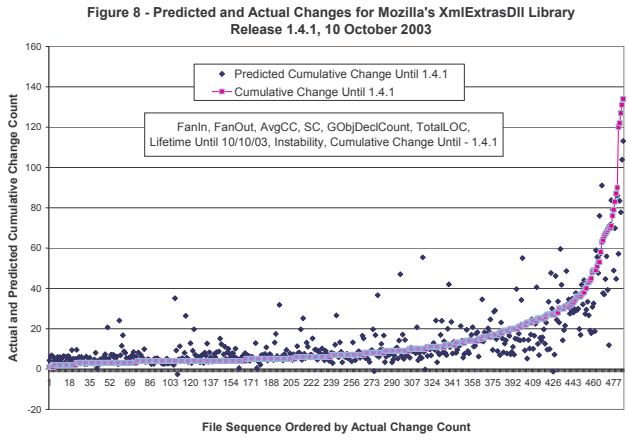


Figure 1.8 – Predicted and actual changes for Mozilla's XmlExtrasDll library

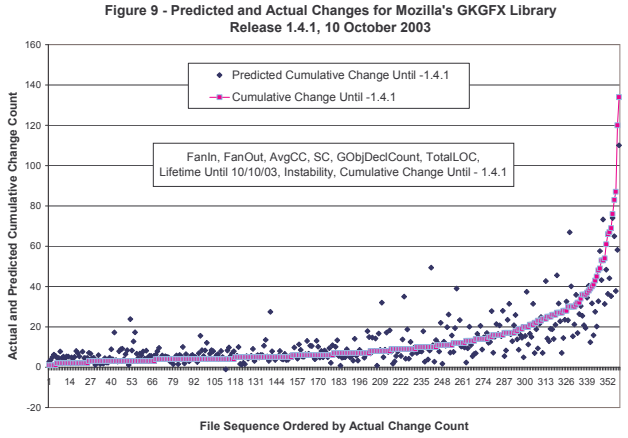


Figure 1.9 – Predicted and actual changes for Mozilla's GKGFX library

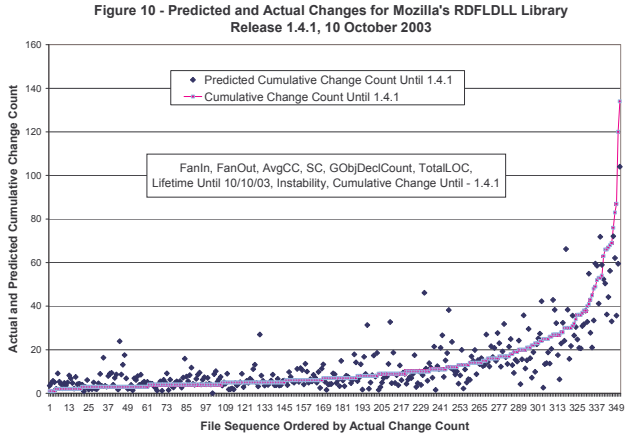


Figure 1.10 – Predicted and actual changes for Mozilla's RdflDll library

In Table 1.6, we show results of an MLR on the entire Windows build for Mozilla, release 1.4.1. The model accounts for about 80 percent of the variation in cumulative change count and Fan-out, Average Cyclomatic Complexity, number of Global Object Declarations, Total Lines Of Code, and Instability are all statistically significant.

Table 1.6 – Results of Multiple Linear Regression. Windows Build of Mozilla, Release 1.4.1

SUMMARY OUTPUT

Predicted and Actual Changes for Windows Build of Mozilla Release 1.4.1, 10 October 2003

Regression Statistics	
Multiple R	0.901019564
R Square	0.811836255
Adjusted R Square	0.811388379
Standard Error	17.03564781
Observations	3370

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	8	4208412.589	526051.5737	1812.637741	0
Residual	3361	975406.8887	290.2132962		
Total	3369	5183819.478			

	Coefficients	*Standard Error*	*t Stat*	*P-value*
Intercept	-0.890092	2.180555	-0.408195	0.683156
FanIn	0.000173	0.007200	0.024007	0.980849
FanOut	1.371579	0.026634	51.496815	0.000000
AvgCC	-0.873727	0.090125	-9.694663	0.000000
SCSize	-0.002014	0.000218	-9.240209	0.000000
GObjDeclCount	-0.264539	0.034726	-7.617985	0.000000
TotalLOC	0.018727	0.000542	34.543551	0.000000
LifeOn_2003_10_10	0.003227	0.001258	2.565292	0.010352
Instability	-5.578067	0.946077	-5.895994	0.000000

Table 1.7 – Correlation Matrix for MLR Model. Windows Build of Mozilla, Release 1.4.1

Predicted and Actual Changes for Windows Build of Mozilla Release 1.4.1, 10 October 2003

	FanIn	*FanOut*	*AvgCC*	*SCSize*	*GObjDeclC*	*TotalLOC*	*LifeOn_20(*	*Instability*	*Cumulative*
FanIn	1								
FanOut	0.036412	1							
AvgCC	-0.00219	0.260892	1						
SCSize	0.056421	0.34662	0.212668	1					
GObjDeclCount	0.016996	0.12169	0.088853	0.077633	1				
TotalLOC	0.08118	0.721062	0.406104	0.249548	0.188281	1			
LifeOn_2003_10_10	0.125683	0.035249	0.134067	0.11667	0.046017	0.152023	1		
Instability	-0.198229	0.402716	0.22057	0.184166	0.020203	0.184365	-0.157599	1	
Cumulative Change	0.064236	0.850391	0.218924	0.216204	0.082584	0.795019	0.090537	0.240712	1

Note that Cumulative change correlates most strongly with Fan-out, then GblObjDec and TLOC.

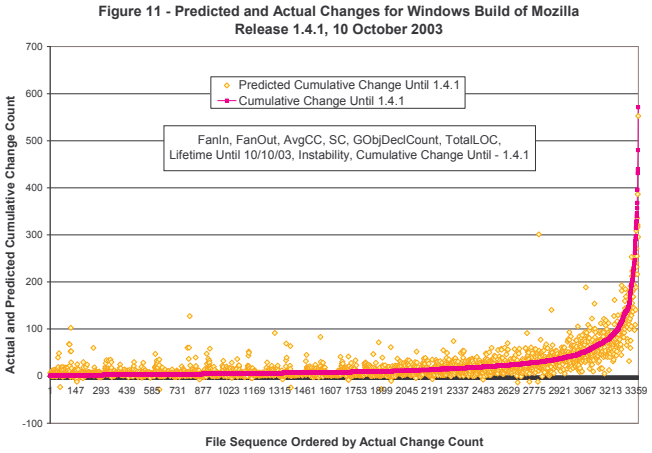


Figure 1.11 – Predicted and actual changes for Windows Build of Mozilla Release 1.4.1. 10 October 2003

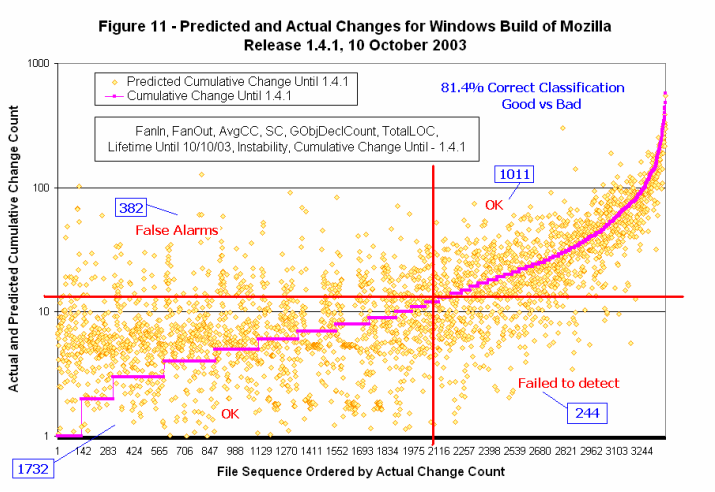


Figure 1.12 – Predicted and actual changes for Windows Build of Mozilla Release 1.4.1. 10 October 2003 (Log)

Comparison of Multiple Linear Regression Analysis Results

Library	Adj R-Sq	Fan-In	Fan-Out	Avg CC	SC Size	GblObjDec	TLOC	Lifetime	Instability
Gkgfx	0.65353		significant	significant			significant		significant
MozFindDll	0.7325		significant	significant			significant		
RdflDll	0.69269		significant				significant		significant
XmlExtrasI	0.70157		significant	significant		significant	significant		significant
All Mozilla	0.80665		significant	significant		significant	significant		significant

Note: Blank entries indicate that metric had no significant affect on predicted change

Table 1.8 – Summary of MLR Statistics

In Table 1.8 , we show the significant metrics for each library and the entire Windows build, along with their Adjusted R-Square statistic for the fit to each library. Note that the significant metrics were not the same for each library. Only Fan-Out and TLOC are significant for all analyses.

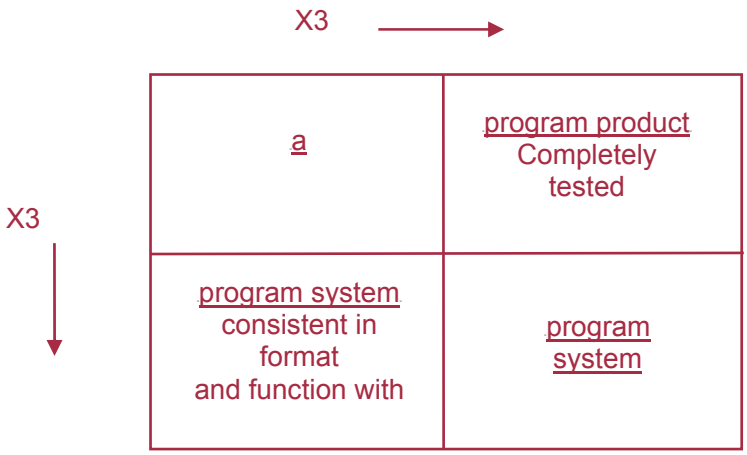
A.1.6.*Summary of Metric Analysis*

Only Fan-out and Total Lines Of Code (TLOC) are strong predictors of cumulative change for the Mozilla Windows 1.4.1 build code base. Surprisingly, Average Cyclomatic Complexity (AvgCC), the number of Global Object Declarations (GblObjDec), and size of Strong Components (SCSize) have virtually no modeling power for cumulative change in that code base.

The Mozilla data provides no measure of effort expended to make changes. It would be very interesting to examine a code base for which such data was available. It is possible that complexity, use of global data, and large mutual couplings may be more highly correlated with effort than we found for change.

A.2.Software Development Effort

Fred Brooks, Program Manager for IBM System/360 and Operating System/360, and later Chair of Computer Science Department, UNC at Chapel Hill, uses this example in his famous "Mythical Man Month".



Most of us conceive of a program as implementing a specific function, running on the platform on which it was developed. Customers want a program product and often a program system product. Note that there is, by Brooks' estimate, a nine to one factor in effort required to build the later over the former.

I develop reasonably well tested code at the rate of about 300 lines per day. So for system products I should expect to generate only about 33 lines of code per day - perhaps 2/3 of that in a team environment!

Here in Syracuse at least two very large software systems have been successfully implemented:

- Over The Horizon Radar (OTHR) contained about 3,000,000 lines of code, written mostly in FORTRAN with some C and some assembly language as well.
- BSY-2 submarine battle management system software is several times larger than OTHR, written mostly in Ada.

A 5 million-line system would require something like:

$$\frac{5{,}000{,}000\ lines}{22\ lines/day}\ x\ \frac{1}{240\ days/year} = 947\ person\ years\ of\ effort$$

Therefore, to complete the project in 2.5 years would require the services of at least 380 well trained software developers.

Two points are almost self-evident:

- a system this large must be partitioned into many relatively small, nearly independent components in order to get it to work
- it would be much better not to create such a large system at all, but rather, to build most of it from reused software components, reserving new code for new requirements.

A.3.Correspondence with Professional Interested in Tools like DepAnal

This email received inquiring tool featured exactly DepAnal, although DepAnal is not publicized yet.

----- Original Message -----

From: <lucaferra@tiscali.it

To: "Murat Gungor" <mkgungor@ecs.syr.edu

Sent: Wednesday, May 05, 2004 11:27 AM

Subject: Re: C++ static dependency analyzer

Hi Murat!

Very kind of you answering to my e-mail!

No, I'm not a student (anymore :)), nor a researcher. I'm working for a company (I'm a software engineer) and I was looking for any commercial (or free) tool to get out that job. My problem is that I have to refactor a project composed by about 250 files with awfully grown dependency relationships that absolutely need to be simplified. I know it's not a trivial matter, because it implies finding out, for each.cpp and .hpp file, every symbol it references, understand if it's necessary to include the file where it's defined or it's enough to declare a foreward reference, then catch the file where the symbol si defined, and so on. If the file is a .cpp or a .hpp with embedded code, the things get worst, because for every called method or accessed attribute of foreign objects called, the respective .hpp where the object is defined has to be identified.

Then I may want to decompose compiling units to break down too complex dependencies. Another problem arises with templates, as long as the template definition class and the actual

parameter definition file of the template object have to be included. Then, of course, I have to deal with defines, macros, global and external symbols, nested classes and so on.

I found some C++ source management tools (for example Source Insight or Understand for C++) that carry out some work, such as gathering all the symbols referenced by a file or a class, or a method, or depict a dependency graph, but none of the ones I tried was able to address every goal; for example, if I inspect a method with Source Insight, it's able to move through a pointer chain finding out the classes involved (ex.obj1_ptr-obj2_ptr-obj3_ptr-method()) but doesn't, at the same time, find the classes involved in parameter passing or automatic objects. Thus, I need to explore first the list of symbols of a file, then each method of my classes, and so on, moving from tool to tool!! I think I will get mad before finishing!!!! :(

So, if you didn't get asleep....

maybe you had the same problem, or you know some workaround or...
who knows... a tool that, fed up with my project files, calculates, file by file,
the right dependencies....?

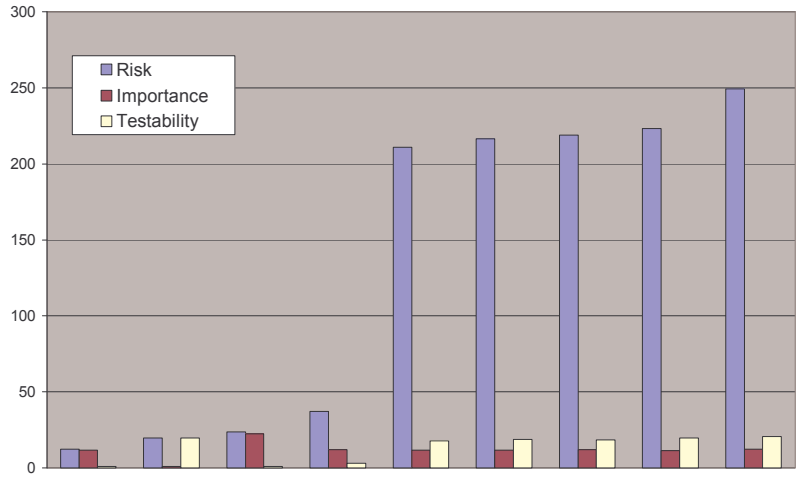
Thanks a lot,

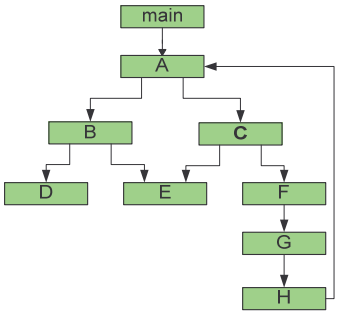
Regards,

Luca

A.4.Demonstrating the Effect of Alpha

In the example below, the project has one strong component with size of five. And we will observe the risk, importance and testability with respect to alpha and strong component.





	D.h	main.cpp	E.h	B.h	F.h	G.h	C.h	H.h	A.h
Risk	12.3692	19.5053	23.6884	37.0857	211.0034	216.5728	218.8581	223.1788	249.2883
Importance	11.7802	1	22.5604	11.978	11.7802	11.6022	11.978	11.4419	12.1977
Testability	1.05	19.5053	1.05	3.0962	17.9117	18.6666	18.2717	19.5053	20.4372

Figure 4.1 – Dependency graph and its corresponding risk chart, alpha = 0.9

In the Figure 4.1, α value is taken 0.9 to make more perceivable the effect α values and β value is taken 1 to mask the effect of internal implementation quality and just focus on coupling between files. If we look at the files with high-risk values, all of them are member of the strong component. In addition, there is a big gap between the risk values of strong component and other individual files. From the perspective of importance, E has the highest value, since not only file B and C depend on file E, but also file A, main, F, G and H depend on file E indirectly. And file A's importance follows file E. However, from the dependency graph, it looks as if only one file depend on file A, which is main file. Because, file A is member of strong component all

the files in strong component directly or indirectly depend on file A. Consequently, this increases importance of file A.

It is straightforward to test file D and E, since they do not depend on any other file. Main file depend on all the files, nevertheless it does not have highest testability, but file A. Again, we see the effect of strong component, file A depends on all the other files, including itself. One interesting fact is that main file and file H has the same testability value, since both of them depend on identical files.

		1 nsreadableutils.h	2 nsicomponentregistrar.h	3 nsidirectoryservice.h	4 nsifile.h	5 nsilocalfile.h	6 nsembedstring.cpp	7 nsembedstring.h	8 bufferroutines.h	9 nsstr.cpp	10 nsstr.h	11 nsstring.cpp	12 nsstring.h	13 nsstring2.cpp	14 nsstring2.h	15 nsstrprivate.h	16 nsaflatstring.h	17 nsasinglefragmentstring.h	18 nsastring.h	19 nsbufferhandle.h	20 nsdependentsubstring.h	21 nsasinglefragmentstring.cpp	22 nsastring.cpp	23 nsdependentsubstring.cpp	24 nsreadableutils.cpp	25 nsxpcom.h	26 nsgredirserviceprovider.cpp	27 nsgredirserviceprovider.h	28 nsmemory.cpp	29 nsxpcomglue.cpp
nsreadableutils.h	1	**X**										X	X					X	X			X	X		X					
nsicomponentregistrar.h	2		**X**		X																										
nsidirectoryservice.h	3			**X**	X																										
nsifile.h	4				**X**														X				X								
nsilocalfile.h	5				X	**X**													X				X								
nsembedstring.cpp	6						**X**	X			X																				
nsembedstring.h	7						X	**X**											X				X								
bufferroutines.h	8								**X**		X																				
nsstr.cpp	9								X	**X**	X																			X	
nsstr.h	10									X	**X**																			X	
nsstring.cpp	11									X	X	**X**		X	X	X			X	X	X		X								
nsstring.h	12									X	X	X	**X**	X	X		X		X				X								
nsstring2.cpp	13									X	X	X	X	**X**	X		X		X	X	X		X								
nsstring2.h	14									X	X	X	X	X	**X**		X		X		X		X								
nsstrprivate.h	15									X	X					**X**															
nsaflatstring.h	16																**X**	X				X									
nsasinglefragmentstring.h	17																	**X**	X			X	X								
nsastring.h	18																		**X**				X								
nsbufferhandle.h	19																			**X**					X						
nsdependentsubstring.h	20																	X	X		**X**	X		X							
nsasinglefragmentstring.cpp	21																	X				**X**									
nsastring.cpp	22																		X		X		**X**								
nsdependentsubstring.cpp	23	X					X														X			**X**							
nsreadableutils.cpp	24	X										X	X					X	X	X	X	X	X		**X**					X	
nsxpcom.h	25		X	X	X	X													X							**X**					
nsgredirserviceprovider.cpp	26				X	X	X	X																		X	**X**	X		X	
nsgredirserviceprovider.h	27			X		X																					X	**X**			
nsmemory.cpp	28																									X			**X**	X	
nsxpcomglue.cpp	29		X	X	X	X													X				X				X	X	X	**X**	

Table 4.1 – Dependency table of a strong component with 29 files from Mozilla.exe component from Mozilla Project Ver. 1.4.1 processed by DepAnal and then proved manually.

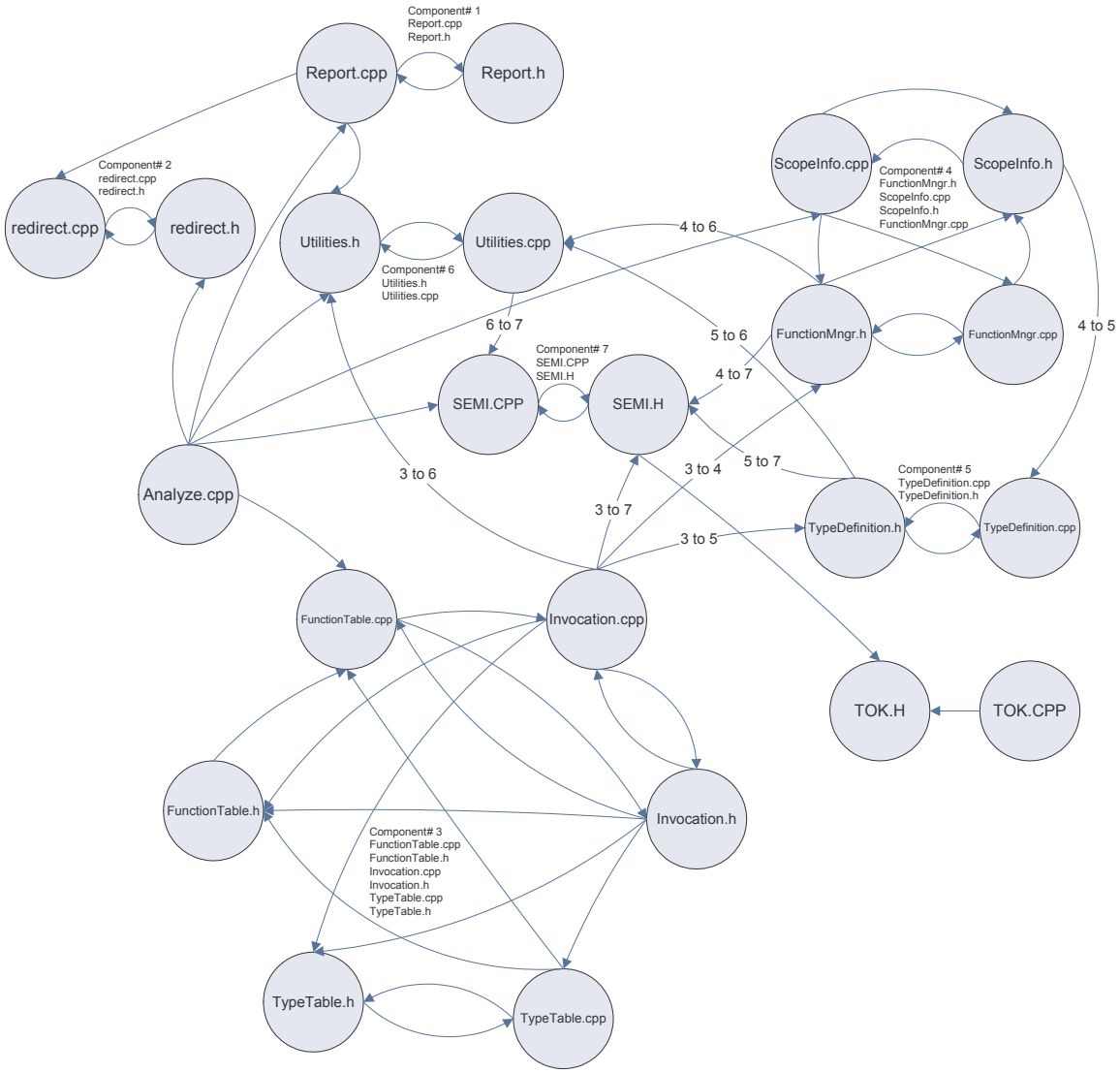


Table 4.2 – Dependency Graph of a strong component from Table 4.2 does not show all the dependency lines for readability.

List of Acronyms

DepAnal	Dependency Analyzer
DepFinder	Dependency Analyzer, new design of DepAnal
DepView	Dependency Viewer
CIF	Change Impact Factor, a relative frequency of required consequential changes in files in the project
StrongComp.	Builds a dependency graph from the data provided by DepAnal and analyzes its strong components, that is, sets of files that are mutually dependent.
Fan-in	Number of files that depend on a given file.
FI	Fan-in
Fan-out	Number of files a given file depends on.
FO	Fan-out
AvgCC	Average Cyclomatic Complexity
CC	Cyclomatic complexity
TLOC	Total Lines Of Code
LOC	Line of Code
SCSize	Size of Strong Components

GblObjDec	Global Object Declarations
MLR	Multiple Linear Regression Analysis
RI	Reusability Index,
CVS	Concurrent Versioning System. CVS is an open source version control and collaboration system.

Bibliography

[1] Frederick Brooks Jr., Mythical Man-Month, 20th Anniversary Edition, Addison-Wesley, 1995

[2] Windows Is So Slow, But Why?; Sheer Size Is Causing Delays for Microsoft http://select.nytimes.com/gst/abstract.html?res=F30C14FD3F540C748EDDAA0894DE404482

[3] http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html A paper by Nancy Leveson and Clark Turner dissecting problems with the Therac-25 XRay machine, which caused the deaths of several patients over a period of 18 months.

[4] "Software's Chronic Crisis", Scientific American, September 1994, www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1944.html

[5] James Gleick, "A Bug and a Crash", www.around.com/ariane.html An informed layman's analysis of the Ariane5 crash by the author of Chaos

[6] CTU ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf NASAs post mortem report on the Mars Climate Orbiter crash.

[7] "The Standish Group Report, Chaos, 1995", www.projectsmart.co.uk/docs/chaos_report.pdf This widely cited document reports results of a survey of companies implementing large IT projects.

[8] Stefan Jungmayr, “Identifying Test Critical Dependencies”, IEEE International Conference on Software Maintenance (ICSM'02), 2002

Jungmayr presents an interesting definition of testability that is based on dependency structure.

[9] Andrea Capiluppi and Juan Ramil, “Change Rate and Complexity in Software Evolution”, Ninth IEEE Workshop on Empirical Studies of Software Maintenance, 2004 Capiluppi and Ramil show that the portion of files from the open-source Arla project that have large change rates tend to have large portion of the highly complex functions

[10] Vassilios Tzerpos, “Automatic Source-File Dependency Structure Extraction for C Programs”, Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research - Builds dependency structures from include relationships, which is less accurate than our type-based approach

[11] James Fawcett, Murat Gungor, Arun Iyer, Kanat Bolazar “Relationship between Code Metrics and Change History”, ISCA 20th International Conference on Computers and their Applications, March 2005.

[12] Mozilla on Microsoft Windows 32-bit Platforms, http://developer.mozilla.org/en/docs/Windows_Build_Prerequisites

[13] Ramanath Subramanyam, M.S. Krishnan, “Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects”, IEEE Transaction on Software Engineering, Volume 29, No.4, April 2003

[14] Aaron Binkley, Stephen Schach, “Inheritance-Based Metrics for Predicting Maintenance Effort: An Empirical Study”, Technical Report 97-05, Computer Science Department, Vanderbilt University, Nashville, TN, 1997

[15] Aaron Binkley, Stephen Schach, Metrics for Predicting Run-Time Failures”, Technical Report 97-03, Computer Science Department, Vanderbilt University, Nashville, TN, 1997.

[16] Mozilla the Configurator, http://webtools.mozilla.org/build/config.cgi

[17] Norman E. Fenton, Niclas Ohlsson, “Quantitative Analysis of Faults and Failures in a Complex Software System”, IEEE Transactions on Software Engineering, Volume 26, Issue 8, August 2000

[18] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.s. Marron, Audris Mockus, “Does Code Decay? Assessing the Evidence from Change Management Data”, IEEE Transaction on Software Engineering, Volume 27, No.1, January 2001

[19] Stephen Schach, Bo Jin, David Wright, Gillian Heller, Jeff Offutt, “Quality Impacts of Clandestine Common Coupling”, http://www.vuse.vanderbilt.edu/~srs/preprints/clandestine.preprint.pdf

[20] Daniel Hoffman, Paul Strooper, “Tool Support for Testing Concurrent Java Components”, IEEE Transaction on Software Engineering, Volume 29, No. 6, June 2003

[21] Barbara G. Ryder, Frank Tip, “Change impact analysis for object-oriented programs”, Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM, 2001

[22] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, Young-Kee Song “Developing an object-oriented software testing and maintenance environment”, Communications of the ACM, Volume 38 , Issue 10, October 1995

[23] MFC Library - MFC Reference http://msdn.microsoft.com

[24] The KDE Project, http://developer.kde.org/source/

[25] Y. Yu, H. Dayani-Fard, J. Mylopoulos, “Removing false code dependencies to speedup software build processes”, Proceedings of the 2003 conference of the Centre for Advanced Studies conference on Collaborative research Pages: 343 – 352, 2003 Toronto, Ontario, Canada.

[26] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner “Bunch: A clustering tool for the recovery and maintenance of software system structures”, In Proceedings of International Conference of Software Maintenance, Aug. 1999.

[27] S. Robitaille, R. Schauer & R. K. Keller, “Bridging Program Comprehension Tools by Design Navigation", IEEE International Conference on Software Maintenance, San Jose, CA (Oct., 2000).

[28] Zhifeng Yu, Václav Rajlich, “Hidden Dependencies in Program Comprehension and Change Propagation”, Ninth International Workshop on Program Comprehension (IWPC'01) May 2001, Toronto, Canada

[29] Daniel German, et.al. “Visualizing the evolution of software using softChange,” Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE), pp. 336-341, 2004.

[30] Daniel German, et. al., “Automating the Measurement of Open source Projects,” ICSE 2003, 3PrdP Workshop on Open Source Software Engineering.

[31] Christopher L. Huntley, “Organizational Learning in Open-Source Software Projects: An Analysis of Debugging Data,” IEEE Trans. Engineering Management, vol. 50, no. 4, pp. 485-493, 2003.

[32] Todd L. Graves, et. al., “Predicting Fault Incidence Using Software Change History,” IEEE Trans on SE, vol. 26, no. 7, pp 653-661, July 2000.

[33] Larry Stephens, Advanced Statistics Demystified, McGraw Hill Inc., May 2004.

[34] Schuyler W. Huck, Reading Statistics and Research, Addison Wesley Longman, 2000.

[35] Michael W. Godfrey, Eric H. S. Lee, “Secrets from the Monster: Extracting Mozilla’s Software Architecture,” Proc. of the Second Intl. Symposium on Constructing Software Engineering Tools (CoSET-00), Limerick, Ireland, June 2000.

[36] Andrea De Lucia, Massimiliano Di Penta, Silvio Stefanucci, Gabriele Venturi, “Early Effort Estimation of Massive Maintenance Processes,” IEEE Proc. of the Int. Conf. on Software Maintenance, pp. 234-237, 2002.

[37] Robert Martin, Agile Software Development, Prentice Hall, 2003.

[38] Geoffrey K. Gill and Chris F. Kemerer, “Cyclomatic Complexity Density and Software Maintenance Productivity,” IEEE Trans on SE, vol. 17, no. 12, pp. 1284-1288, Dec 1991. (Not referred)

[39] Rudolf Ferenc, István Siket and Tibor Gyimóthy, Extracting Facts from Open Source Software. In Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004), Chicago Illinois, USA, to appear, September 11-17, 2004.

[40] Ping Yu, Tarja Systa, Hausi Muller, “Predicting Fault-Proneness using OO Metrics - An Industrial Case Study”, Conference on Software Maintenance and Reengineering 2002 (CSMR’02) IEEE.

[41] Sergey Brin, Lawrence Page, “The Anatomy of a Large-Scale Hypertextual Web Search Engine” Computer Networks and ISDN Systems, 1998.

[42] Google. www.google.com

[43] Larry Page, Sergey Brin, R. Motwani, T. Winograd, “The PageRank Citation Ranking: Bringing Order to the Web”, Technical Report of Stanford Digital Library Technologies Project, Stanford University 1998.

[44] G. Pinski and F. Narin. "Citation Influence for Journal Aggregates of Scientific Publications: Theory, with Application to the Literature of Physics". Information Processing and Management, 12(5):297--312, 1976.

[45] Katsuro Inoue, Reishi Yokomori, Hikaru Fajiware, Tetsuo Yamamoto, Makoto Matsushita, Shinji Kusumoto , “Component Rank: Relative Significance Rank for Software Component Search”. IEEE, 2003.

[46] Victor R. Basili, Lionel C. Briand, Walcelio L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transaction on Software Eng. Vol.22 No.10, 1996

[47] Robert Martin Year 1994, OO Design Quality Metrics, an Analysis of Dependencies, Workshop on Pragmatic and Theoretical Directions in Object-Oriented Software Metrics, OOPSLA'94, October 1994.

[48] Sarita Bassil, Rudolf K. Keller, A Qualitative and Quantitative Evaluation of Software Visualization Tools, In Proceedings of the Workshop on Software Visualization, pages 33-37, Toronto, ON, May 2001.

[49] S. Bassil, R.K. Keller, "Software Visualization Tools: Survey and Analysis", Proc. of the 9th International Workshop on Program Comprehension (IWPC'01), IEEE, May 2001.

[50] Martin Fowler, Reducing Coupling, IEEE Software July/August 2001

[51] Andrea Capiluppi - Juan F. Ramil, Studying the Evolution of Open Source Systems at Different Levels of Granularity: Two Case Studies, Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2004), September 2004

[52] R. Kazman, S.J. Carrière, View Extraction and View Fusion in Architectural Understanding, IEEE, Fifth International Conference on Software Reuse, 1998.

[53] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss and Mikko Tarkiainen, Columbus – Tool for Reverse Engineering Large Object Oriented Software Systems, In Proceedings of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001), Szeged, Hungary, pages 16-27, June 15-16, 2001

[54] R.P. Higuera and Y.Y. Haimes, "Software Risk Management," Technical Report CMU/SEI-96-TR-012, Software Engineering Institute, 1996

[55] M.M. Lehman and L.A. Belady, Program Evolution: Processes of Software Change. Academic Press, 1985.

[56] Mozilla Layout Engine used for web page reorganization, http://www.mozilla.org/newlayout/

[57] T. Roetschke and R. Krikhaar. Architecture analysis tools to support evaluation of large industrial systems. In Proc. IEEE International Conference on Software Maintenance (ICSM), pages 182-191, Montréal, Canada, October 2002.

[58] J. Lakos. Large-scale C++ software design. Addison-Wesley, 1996.

[59] James W. Fawcett, Murat K. Gungor, Arun V. Iyer, “Analyzing Static Structure Of Large Software Systems” Based on Data from Open-Source Mozilla Project, SERP'05, The 2005 International Conference on Software Engineering Research and Practice, Nevada USA on June 2005.

[60] James W. Fawcett, Murat K. Gungor "Software Development Risk Model", Applied to Data from Open-Source Mozilla Project, SERP'05, The 2005 International Conference on Software Engineering Research and Practice, Nevada USA on June 2005.

[61] James W. Fawcett, Murat K. Gungor, “Applied Software Development Risk Model”, IPSI-USA-2005 Cambridge, Massachusetts, USA

[62] Steven Morphet, James Fawcett, Kanat Bolazar and Murat Gungor, “Neural Net Analysis of the Propensity for Change in Large Software Systems”, 2006 International Joint Conference on Neural Networks, IJCNN 2006, Vancouver, BC, Canada

[63] Ricky E. Sward, A. T. Chamillard, "Re-engineering global variables in Ada",Annual International Conference on Ada archive, Proceedings of the 2004 annual ACM SIGAda international conference on Ada: The engineering of correct and reliable software for real-time & distributed systems using Ada and related technologies, pages 29-34, Atlanta, Georgia, USA 2004.

[64] Nassar, D. M., Rabie, W. A., Shereshevsky, M., Gradetsky, N., Ammar, H.H, BoYu, Bogazzi, S., and Mili, A. Estimating Error Propagation Probabilities in Software Architectures. Technical Report, College of Computer Science, New Jersey Institute of Technology 2002. http://www.ccs.njit.edu/swarch/ep.pdf

[65] Paul Festa, “Apple snub stings Mozilla”, http://news.com.com/2100-1023-980492.html

[66] History of Mozilla Application Suite, http://en.wikipedia.org/wiki/History_of_Mozilla_Application_Suite

[67] Muthu Ramachandran, "Software reuse guidelines", ACM SIGSOFT Software Engineering Notes, May 2005, Volume 30, Issue 3, pp. 1-8

[68] Shari Lawrence Pfleeger and Shawn A. Bohner, "A Framework for Software Maintenance Metrics," IEEE Transactions on Software Engineering, May 1990, pp. 320-327.

[69] S. Bohner and R. Arnold. "Software Change Impact Analysis". IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[70] S. Barros, Th. Bodhuin, A. Escudie, J.P. Queille, and J.F. Voidrot, "Supporting Impact Analysis: A Semi-Automated Technique and Associated Tool," Proceedings of the Conference on Software Maintenance, 1995. IEEE, Piscataway, NJ, USA, 95CB35845 pp. 42-51.

[71] Arnold, R.S. and Bohner, S.A., "Impact Analysis — Towards a Framework for Comparison", International Conference on Software Maintenance 1993, IEEE Computer Society, LosAlamitos, CA, USA, 1993, pp. 292–301

[72] M. Lee. Change Impact Analysis of Object-Oriented Software. Ph.D. Dissertation, George Mason University, Feb.1999.

[73] M. Lee, A. J. Offutt, and R. T. Alexander. "Algorithmic Analysis of the Impacts of Changes to Object-oriented Software". In TOOLS-34 '00, 2000.

[74] James Law and Gregg Rothermel, "Incremental dynamic impact analysis for evolving software systems", In Proceedings of the International Symposium on Software Reliability Engineering, Nov. 2003

Vita

NAME OF AUTHOR	Murat Kahraman Güngör
PLACE OF BIRTH	Kayseri, Turkey
DATE OF BIRTH	November 6, 1976
EDUCATION	
JULY 2006	Ph.D. in Computer and Information Science, Collage of Engineering and Computer Science, Syracuse University, Syracuse, NY USA
MAY 2001	M.S. in Computer and Information Science, Department of Electrical Engineering and Engineering and Computer Science, Syracuse University, Syracuse, NY USA
May 1997	B.S in Industrial Engineering, Sakarya University, Sakarya, TURKEY
EXPERIENCE	
Teaching Assistant	CSE 686/891 – Internet Programming (SU02-SU06) CSE 784 – Software Studio (FL02-FL05) CSE 687 – Object Oriented Design (SP03) CSE 784 – Software Studio (FL02-FL05) CSE 681 – Software Modeling and Analysis (SU05-SU06)