SUBMISSION PAGE

(Not counted towards page limits)

Target conference	**IPSI-USA-2005 CAMBRIDGE, MASSACHUSETTS, USA**
Title of the paper	**Applied Software Development Risk Model**
Paper will be presented by	**Murat Gungor**
Keywords	**Dependency analysis, metrics, open-source, software quality, risk analysis.**
Affiliation	**Syracuse University, Syracuse, New York**
Postal address	**CST 1-122, Syracuse University, Syracuse, New York 13244 USA**
Author 1	**James Fawcett, Ph.D.** **jfawcett@twcny.rr.com** **(315) 443-3948**
Author 2	**Murat Gungor, MSCS** **mkgungor@ecs.syr.edu** **(315) 443-4003**

Applied Software Development Risk Model

Fawcett, W. James, Ph.D.; Gungor, K. Murat, MSCS

Abstract

The Process of developing large software systems creates many 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. The software development risk model, developed here, shows that the density of dependencies within such clusters plays a crucial role in this behavior. We develop a file-rank procedure which orders the entire system's file set by increasing risk.

This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to reduce development risk. We have applied this model to code from several projects with interesting results.

Index Terms: Dependency analysis, software quality, risk analysis, file ranking.

1. Introduction

Development of large software systems creates many, often thousands, of source code files with complex inter-dependencies. We develop, in this paper, a software development risk model. It shows 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 [2]. 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.

Our model is derived from a notion of Test Risk, based on the work of Jungmayr[1], 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 and test risk, both defined in the paper. This ranking process should prove to be useful while managing the development of large systems, indicating where attention should be focused to improve development 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, to the well known MFC library, used to develop windows applications, and to our own analysis software, all with interesting results.

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

2. Problem With Large Software Systems

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.

Therefore it is very important to provide timely feedback to software engineers and project management about the state of the software development project. Early detection of quality defects will avoid delays, difficulties and costs associated with development later in the project lifecycle. Higuera and Haimes 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*" [15].

Manuscript received May 1, 2005.
Dr. James W. Fawcett is with the Electrical Engineering and Computer Science Department, Syracuse University, New York, USA (e-mail: fawcett@twcny.rr.com).
Murat K. Gungor is with the Electrical Engineering and Computer Science Department, Syracuse University, New York, USA
(e-mail: mkgungor@ecs.syr.edu).

Source code itself carries valuable information relevant for monitoring development projects. Furthermore software source is always accessible to the project and carries up-to-date information, unlike project documentation, which may be out of date or may not exist. Source code provides quantitative information that can be turned into qualitative assessments of the state of development to provide timely feedback to software engineers and project management about the state of the software project [14].

Software systems can be extremely complex. Here is what Frederick Brooks; Kenan Professor of Computer Science, University of North Carolina, Chapel Hill [5] 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 [5], 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 due to race condition[1], 1985-87 [6], Denver Airport Baggage System failure[2], 1995 [7], Ariane 5 crash, arithmetic errors coupled with specification and design errors[3], 1996 [8], and Mars climate orbiters[4], 1999 [9], to cite a few.

Complexity causes not only malfunctions in operational systems, but problems with the development process resulting in cost and schedule overruns and project cancellations. The Standish Group published a widely cited report claiming these survey[5] results, 1995 [10]:

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.

The goal of this research is to provide tools – metrics and the programs needed to employ them, to detect when large projects are getting into trouble, based on examinations of their code bases. Our primary concern 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 don't require semantic analysis[6]. A secondary goal is to devise concrete procedures for making improvements to observed defects and quantizing the improvement with the same metrics used for analysis.

We have been examining several large systems: the open-source Mozilla and KHTML projects, and the libraries MFC[7], STL[8], and some of the Boost Libraries[9]. This gives us a mix of open-source, commercial, and Expert developed code. As you will see, the results are, so far, quite interesting.

The figures, Figure 1 and
Figure 2, below[10], represent dependency relationships within the GKGFX library (NGLayout Project [16]) from the Mozilla project version 1.4.1. The large disk in the figure represents a collection of mutually dependent files - a strong component in graph terminology - 119 files in all.

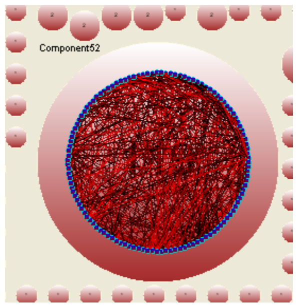


Figure 1 – Internal dependencies of component #52 consist of 119 files.

Every one of these files depends, either directly or indirectly on every other. The dependency relationships are shown by the dense lines within the disk. Each dot around the circle is one of the 119 files. The plot, below,
Figure 2, adds dependencies of files outside the same strong component on files inside. If any file inside the strong component is changed, it may break the operation or design of any other file in

[1] This system is complex
[2] Problems with both mechanical and software complexity.
[3] "Very tiny details can have terrible consequences," says Jacques Durand, head of the project, in Paris. "That's not surprising, especially in a complex software system such as this is." [7]
[4] Data in English units instead of metric in software application code.
[5] Sample size of 365 respondents, representing 8,380 applications [9].
[6] Most of our analysis to-date is based on static type-based dependencies.
[7] Microsoft Foundation Class library, part of the Visual Studio Software Developers Kit.
[8] Standard Template Library, part of the C++ standard library.
[9] C++ libraries supplied by participants in the last C++ standardization process.
[10] These figures were generated by our visualization tool, DepView.

the component and any of the external files using services of this component, as shown by all these dependency relationships.

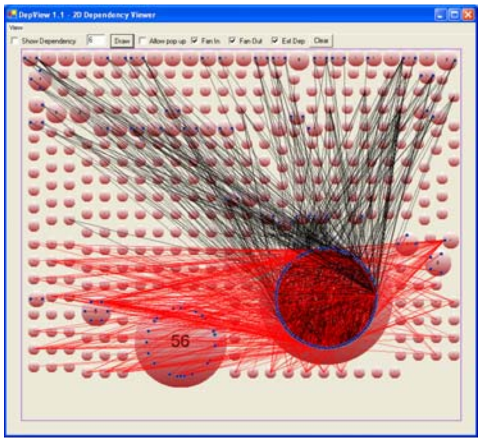


Figure 2 – Internal & external Fan-In dependencies of component#52 consist of 119 files.

The density and complexity of these relationships demonstrates that this component has extremely poor testability characteristics. Should a developer find a defect in one of these files and fixes it, a huge number of files – more than 119, need to be retested to demonstrate that the change caused no other breakage.

The number of source files is too large to pay attention to the semantics of each file that may negatively affect quality of the software project. We need a way to rank files based on their impact on system quality. We have several questions, answers for which will help us to identify those files or groups of files that present the greatest development risk. Which files are contributing the most to large strong component size? Can we order the risk of files by using each file's interrelation with other files in the system? How does internal quality of a file and the files on which it depends affect overall system quality?

These questions led us to consider ranking files by their risk level. This way, important files will be ranked high, according to their risk contribution to the software system. These files become the target on which developers focus first in order to alleviate structural problems. Below, we develop a software risk model by considering dependency relations among files and files' internal metric information.

Our goal is to enable a Project Manager to visualize his large (thousands of files) code base and determine where corrective action is needed and to continuously monitor development progress of the system. The static dependencies we have been discussing are visible on a micro scale. Each developer knows what other files her code depends on. However, the dependencies on a macro scale are invisible to humans, due to the overwhelming complexity of real large projects[11], in the large.

The results to-date have shown that analysis of the dependency structure of a large project is attainable, useful, and potentially provides a basis for corrective action. Exploring means for corrective action is the major area for the remainder of this research. Inspired by a discussion by Jungmayr [11] on testability, we are working on 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. Although the work of Jungmayr inspired this approach, our algorithm differs in its measures of dependency, in its quality assessment, and in the use of importance. Furthermore, unlike Jungmayr, we classify and treat differently dependency types, e.g., mutual, global, call-back 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 – an interesting partitioning problem that is being worked on by another student. 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.

After identifying potential dependency problems, we also expect to develop some concrete ways to improve the dependency structure of a large system without a detailed understanding of its semantics. We present some initial findings about that here.

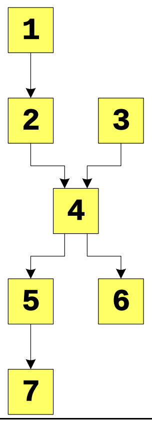
3. *Risk-Based File Ranking Algorithm*

In order to narrow down those files which need close attention, we rank files according to internal implementation metrics and external interaction with other files in the project, we call this ranking the Software Development Risk Model. Files with high ranks are targets for software engineers to use care while re-using, enhancing functionality or fixing latent errors, since any change to that file may force a chain of new (consequential) changes.

In the Figure 3, an arrow shows the dependency relationship between two source files, where each square represents a source file. If the arrow points from file A to file B, then file B provides services to A, therefore A depends on B. In this example, files 6 and 7 are the most independent files since they do not depend on any other files' services. It is straightforward to test them, at least in terms of these structural relationships.

[11] A project developing 5 million lines of code in two and a half years needs about 350 developers.

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 project file 1 has high test risk, due to its dependence on all the other files except file 3, either directly or indirectly. But 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 3: Simple dependency between files

To discover the state of a software system, we develop a file-rank [3] procedure which orders the entire system's file set by increasing risk, the product of importance and test risk.

This is leads us to define two things:

- Importance of a file
- Test risk of a file

In the following section importance and test risks are elaborated.

A. File Importance, I

Here we define importance from the perspective of change impact. Importance, I, can be greater than or equal to 1. File 1 has importance 1 ($I_1 = 1$), since no other files depend on file 1, it can be changed without worrying about anything other than its internal implementation. If we pick a file which is being used by other files, it will have higher importance, since any change applied to that file may affect the files above it.

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

Here we use coefficient alpha (α_{ij}), which shows the risk of impact on files j caused by change in file i. Thus, if there is no risk that a change in file i will affect file j, 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.

α_{ij} is 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, α_{21} = 1, and the importance of file 2 is 1 + α_{21} = 2, e.g. the number of files changed when file 2 changes.

$I_1 = 1$
$I_2 = 1 + \alpha_{21} I_1$ $I_2 = 1 + \alpha_{21}$
$I_6 = 1 + \alpha_{64} I_4$ $I_6 = 1 + \alpha_{64} + \alpha_{64}\alpha_{42} + \alpha_{64}\alpha_{42}\alpha_{21} + \alpha_{64}\alpha_{43}$
$T_1 = \beta_1 + \alpha_{21} T_2$ $T_1 = \beta_1 + \alpha_{21}\beta_2 + \alpha_{21}\alpha_{42}\beta_4 + \alpha_{21}\alpha_{42}\alpha_{54}\beta_5 + \alpha_{21}\alpha_{42}\alpha_{64}\beta_6 + \alpha_{21}\alpha_{42}\alpha_{54}\alpha_{75}\beta_7$
$T_2 = \beta_2 + \alpha_{42} T_4$ $T_2 = \beta_2 + \alpha_{42}\beta_4 + \alpha_{42}\alpha_{54}\beta_5 + \alpha_{42}\alpha_{64}\beta_6 + \alpha_{42}\alpha_{54}\alpha_{75}\beta_7$
$T_6 = \beta_6$

Table 1 Example of importance and Test Risk of some files in Figure 3

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.

B. Test Risk of File, T

Test Risk of a software file is an important issue in assuring that required functionality is implemented without errors. "*A lack of testability contributes to a higher test and maintenance effort*" [1]. 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 1 Test Risk of file 6 (and 7) is 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 C

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

Magnitude of Test Risk metric varies according to the depended upon files' internal structure 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 be retested and perhaps change.

C. 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 paper 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:

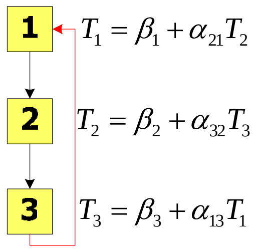
$$\beta_i = 1 + \frac{1}{N}\sqrt{(\frac{m_{1i}}{M_1})^2 + (\frac{m_{2i}}{M_2})^2 + ... + (\frac{m_{Ni}}{M_N})^2}$$

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

Lowercase m is the measured metric, uppercase M is boundary value metric.

D. Case of Circular Dependency

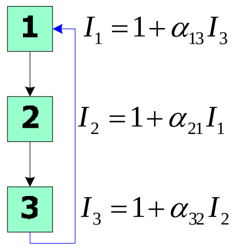
In the case of circular dependency, each member has the same importance and Test Risk size, since there is a mutual dependency between each file, and any change can affect any other file, as shown in Figure 5.


1
$T_1 = \beta_1 + \alpha_{21} T_2$
2
$T_2 = \beta_2 + \alpha_{32} T_3$
3
$T_3 = \beta_3 + \alpha_{13} T_1$

$$\begin{aligned}
T_1 &= \beta_1 + \alpha_{21} T_2 \\
T_1 &= \beta_1 + \alpha_{21}(\beta_2 + \alpha_{32} T_3) \\
T_1 &= \beta_1 + \alpha_{21}(\beta_2 + \alpha_{32}(\beta_3 + \alpha_{13} T_1)) \\
T_1 &= \frac{\beta_1 + \alpha_{21}\beta_2 + \alpha_{21}\alpha_{32}\beta_3}{1 - \alpha_{21}\alpha_{32}\alpha_{13}}
\end{aligned}$$

Figure 4: Circular dependency, Testability.

In Figure 4 and Figure 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.


1
$I_1 = 1 + \alpha_{13} I_3$
2
$I_2 = 1 + \alpha_{21} I_1$
3
$I_3 = 1 + \alpha_{32} I_2$

$$\begin{aligned}
I_1 &= 1 + \alpha_{13} I_3 \\
I_1 &= 1 + \alpha_{13}(1 + \alpha_{32} I_2) \\
I_1 &= 1 + \alpha_{13}(1 + \alpha_{32}(1 + \alpha_{21} I_1)) \\
I_1 &= \frac{1 + \alpha_{13} + \alpha_{13}\alpha_{32}}{1 - \alpha_{13}\alpha_{32}\alpha_{21}}
\end{aligned}$$

Figure 5: Circular dependency, Importance.

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[12].

Figure 6 and Figure 7 show the matrix representation of importance and Test Risk for Figure 5.

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

Figure 6 – Matrix representation of importance

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

Figure 7 – Matrix representation of Test Risk

In Figure 8, if for all i, j, α_{ij} are greater than 0.7071, behavior becomes undefined, as the change sequence becomes unbounded.

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

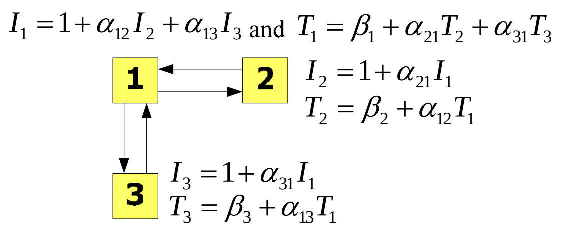

$I_1 = 1 + \alpha_{12} I_2 + \alpha_{13} I_3$ and $T_1 = \beta_1 + \alpha_{21} T_2 + \alpha_{31} T_3$
1
2
$I_2 = 1 + \alpha_{21} I_1$
$T_2 = \beta_2 + \alpha_{12} T_1$
3
$I_3 = 1 + \alpha_{31} I_1$
$T_3 = \beta_3 + \alpha_{13} T_1$

Figure 8 – Three mutually depended files.

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

and

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

It can be clearly seen in Figure 8 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 density of dependency paths increase the critical value for α_{ij} decreases.

E. Risk Contribution of a Single File

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

$$R_i = I_i x T_i$$

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

We develop 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 development risk.

Risk factor provides feedback about individual files and also provides insight about the global state of a software project. For instance, if a developer needs to test a file, its risk factor will provide an idea of 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. INTERPRETING RESULTS

In this section, we describe interpretation of obtained results and how to asses the quality of software systems from analysis of their source code. And the following section describes candidate improvement techniques.

A. Visualizers Providing a Different View

All the results, obtained from software projects are represented as text, and interpreting these texts is almost as hard as reading source code. We felt the need of providing other kinds of representations, which would disclose the qualitative nature of this information in easily understandable fashions. We developed a 2D dependency viewer to give us comprehensible views of large software systems.

In Figure 9, we see a visualization of the Mozilla GKGFX library. Smallest circles represent individual source files – strong components of size 1, others have larger mutual dependency sets, e.g., strong components. The number at the center of each circle indicates the size of a strong component. The dimension of each circle is proportional to size of the strong component. And a line between circles shows dependency among circles

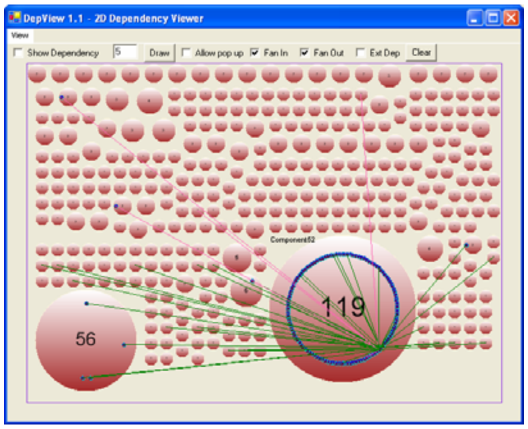


Figure 9 – Mozilla GKGFX library DepView screen shot.

Component #52 is the largest component in the GKGFX library with 119 source files.

Figure 9 shows dependencies (Fan-in and Fan-out) of one of the files in that component. As we see, this file depends not only on the files inside the associated strong component but also 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 that introduced change does not have an effect on the functionality of that file. In Figure 1, above, we see the internal dependency relations within the component #52; the density of dependencies in component indicates significant development risk.

B. Seek Fundamental Qualitative and Quantitative Measure of System Quality

Two possible ways of proceeding exist. One is focusing on strong components; the other is focusing on individual files. In order to evaluate the quality of system, we need quantitative inputs about software system under study. We see in Figure 10, data gathering and processing flow during analysis of software.

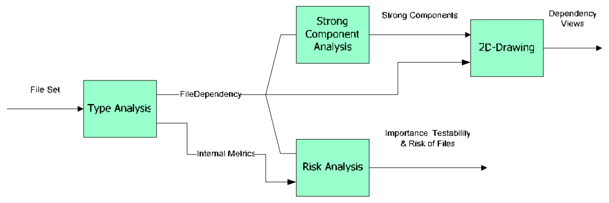


Figure 10 – Analyzing and visualizing software system's quality

Quantitative inputs that could be used are:

- Importance metric of a file
- Testability metric of a file
- Average Cyclomatic complexity per function
- Maximum Function size
- Number of function declarations
- Strong component size
- Global object declaration count
- Total lines of code
- Change duration
- Number of change occurred during change period

We studied not only large software systems, but also we tested on our own project, which has 25 source files. Since we know our project well: which files carry high risk, which files need to be re-factored; however we were not sure about the size of the strong components and their interaction with other files:, we obtained results that are encouraging towards effectiveness of our approach to this research problem even for relatively small size project as in Figure 11.

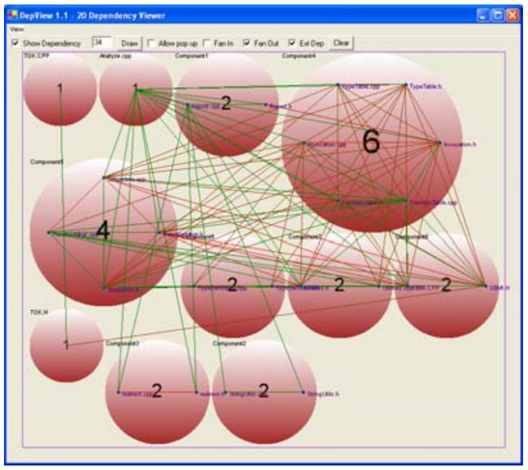


Figure 11 – DepAnal's internal dependency structure. Consist of 25 files

In Figure 11, we see DepAnal's file level dependency structure, before seeing this picture, we were not aware that DepAnal has a strong component with 6 files. For this small size project, the largest strong component contains more than 20% of the source files in the entire project. As developers of this project, we felt that this view demonstrated potential problems in seconds, without the user needing detailed knowledge of this project.

5. Empirical Study On Mozilla Library, GKGFX And MFC

We downloaded version 1.4.1 of the Mozilla Win32 configuration [12] [13]. 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 obtained without diving into implementation details, which is very important for the software project's testers, developers, and managers.

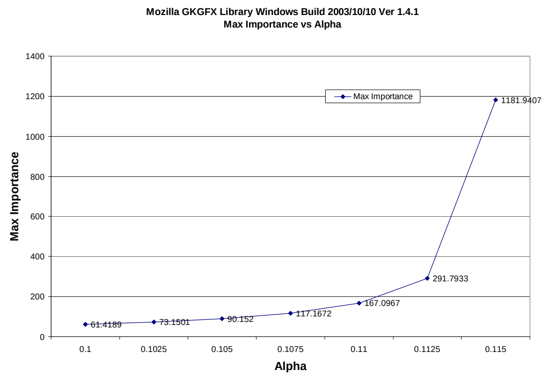


Figure 12 – Max Importance vs. Alpha (α) value for Mozilla GKGFX Library Version 1.4.1.

First, we explored the variation of maximum importance with $\alpha_{ij} = \alpha$, making the simplifying assumption that it is constant for all files. 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 12, that Importance grows without bound above α =0.115. 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.

Next, we calculated Risk factor values using average cyclomatic complexity[13] (AvgCC) and Fan-out[14] values for each file in GKGFX when calculating β; we took desirable limits to be 10 for AvgCC and 5 for Fan-out. We used 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.

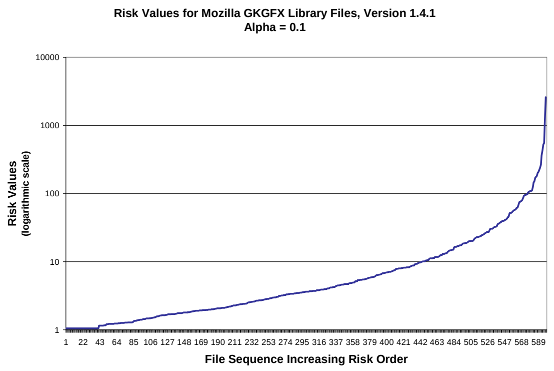


Figure 13 – Risk values for files in GKGFX Library

Figure 13 shows the risk rate of all the files in Mozilla library, GKGFX, in increasing order. Note that about %10 of the files have most of the Risk in the library code. Interestingly, the file with the highest risk is part of the second largest component (with 56 files in Figure 9). This shows that the high risk files are not guaranteed to be part of largest strong component.

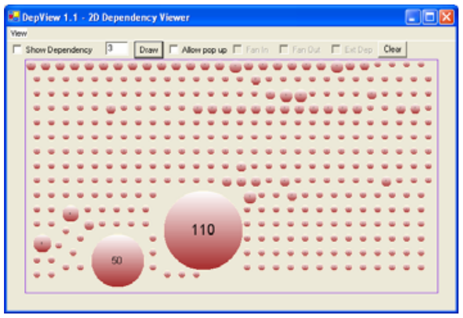


Figure 14 – Components of in GKGFX Library after removing global object dependency.

Not surprisingly, all the files with high risk are members of strong components [4]. This also proves that risk analysis is providing dependable information.

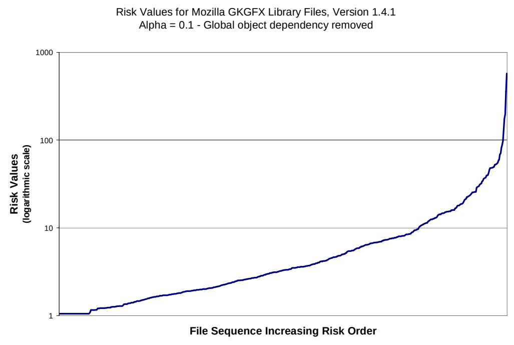


Figure 15 – Risk values for files in GKGFX Library, after global object dependency removed

Figure 15 shows logarithmic scale risk values of GKGFX Library after global dependency removed, we see there is dramatic reduction in risk values. As we see in Figure 14, after removing dependencies on global data, strong component size is significantly reduced. Moreover, this made a big impact on maximum risk as in seen Figure 15 and Table 2, max risk reduced from 2593.4 to 572.93.

	With global object	Without global object
Maximum Risk	2593.4	572.93
Average Risk	22.86	10.02

Table 2 – GKGFX risk values

6. *Empirical Study Of MFC*

In this section, we explored the affect of two specific changes on MFC project's structural quality. Effect of elimination of global data is examined, and using interfaces and class factories to avoid binding to concrete classes, is simulated with very interesting results.

A. Elimination of Global Data

One of the promising ways for improvement is exclusion of global variable. In this part of the paper, our goal is to find out the effect of global variables on the size of strong components. Using our tools we analyzed MFC files, and find dependencies among MFC files with and without dependencies caused by global objects.

Figure 16 shows the result of a first run considering global variable dependencies; we see one big component consisting of 135 files.

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

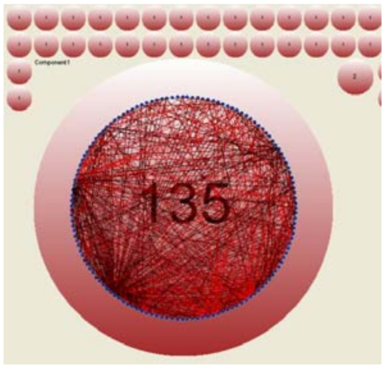


Figure 16 – MFC - Internal dependencies of Component #1 with 135 files.

When we exclude dependencies caused by global variables we obtained
Figure 17. This shows that the largest component size is reduced from 135 to 25. Interestingly 110 files involved in the big strong component are due to global object dependencies. This indicates that use of global variables should be avoided as much as possible[15].

	With global object	Without global object
Max. Risk	204.62	37.22
Avrg. Risk	13.18	2.84

Table 3 – MFC risk values

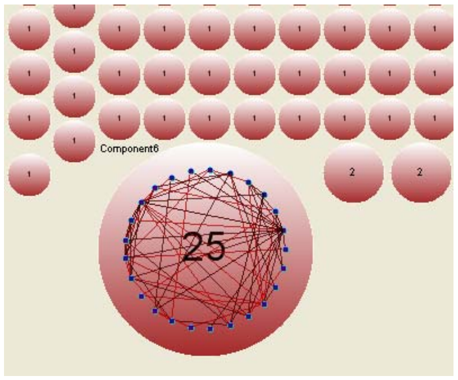


Figure 17 – MFC - Internal dependencies of Component #6 with 25 files, after removing global object dependency.

MFC has 251 source files, which is less than half the size of Mozilla's GKGFX Library. When we remove global object dependencies, the size of the largest strong component is significantly reduced.

Similarly risk values are reduced: the highest risk value reduced from 204.62 to 37.22. This indicates the effect of strong component size on risk rate, Table 3 - Figure 18 – Risk values for files in MFC Library, global object dependency removedwith global object dependencies, and Figure 18 without global object dependencies. The files with highest risk rate are different, in the first case appinit.cpp, and in the second wincore.cpp (after removing global object dependency).

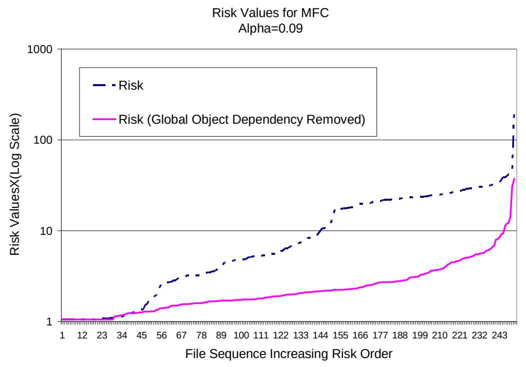


Figure 18 – Risk values for files in MFC Library, global object dependency removed

The MFC library exhibits Test Risk singularity for alpha very close to 0.1. This means that its code would be extraordinarily sensitive to change. When we reduced alpha to 0.09, the singularity was avoided, leading to the analysis above. Since MFC is a viable commercial product, we believe that typical values for alpha in commercial systems may well be less than 0.1. Estimating alpha is an area of research we are now pursuing[16].

B. *Insertion of Interfaces and Object 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 interface by reducing the predicted potential for change factor, alpha. Using GKGFX library which has

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

[16] 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. Finding means to measure and predict alpha values is an issue we are currently investigating.

598 source files total, we first applied regular risk analysis with alpha of 0.1 for all files as in Figure 19, and highest risk value for this analysis came out 2543.

To simulate the affect of interface, we selected 23 files with high fan-in values (fan-in value 31 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 2543 to 853, as shown in Figure 19.

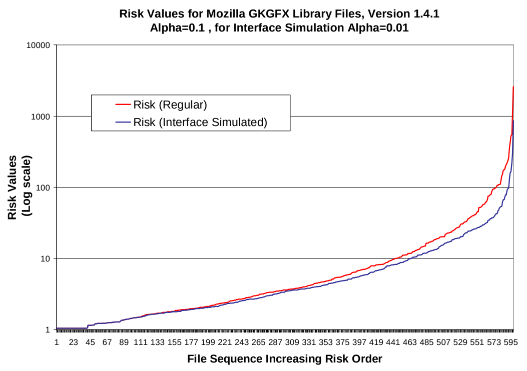


Figure 19 – Risk Analysis of GKGFX Library

In summary, modifying a relatively small number of files (23 files out of 598) to use interfaces, resulted in the highest risk value being reduced from 2543 to 853. 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 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 reduced system risk. This task would be extraordinarily difficult without the file rank based risk analysis we have developed. Thus, without adding a great analysis load on developers, but just using simulation, we can locate files to focus on, which contribute the highest benefit to remedy structural problems.

Now, we apply the same analysis to the simpler DepAnal program. This will let us gauge whether the risk model will be useful for smaller system as well as large system like Mozilla. Figure 20, below we show the outcomes of risk analysis on DepAnal as it currently exists, and also when we simulate the introduction of interfaces.

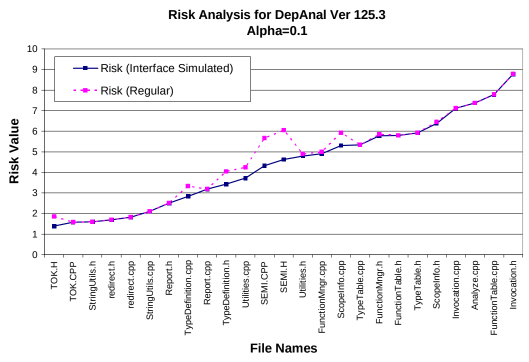


Figure 20 – Analysis of Risk of DepAnal

We introduce 4 interfaces into a system that has 25 files. For the simulated interfaces alpha values are reduced to 0.01. The chosen files all have the high fan-in values, and as it is seen from the figure, overall risk is reduced.

7. CONCLUSIONS

In this paper we present a new software development Risk Model and have shown that the model can be used to predict problem areas, as concentrations of high risk files.

Inoue et. al. [3] proposed a usage-based file rank proced-ure. Their goals are to retrieve reusable 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 paper and the former are have some similarities but the algorithms and final results are quite different. Our paper uses a two-level structure with Test Risk and importance as the bases for ranking.

Jungmayer [1] proposes a testability analysis process that inspired our work. We have added propagation likelihood weights and the notion of importance to build a stronger model.

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've 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.

REFERENCES

[1] Stefan Jungmayr, "Identifying Test-Critical Dependencies", Proceedings of the International Conference on Software Maintenance (ICSM'02), IEEE, 2001.

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

[3] Katsuro Inoue, Reishi Yokomori, Hikaru Fajiware, Tetsuo Yamamoto, Makoto Matsushita, Shinji Kusumoto, “Component Rank: Relative Significance Rank for Software Component Search”, 25th International Conference on Software Engineering, 2003.

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

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

[6] 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.

[7] “Software’s Chronic Crisis”, Scientific American, September 1994, www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1944.html

[8] 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

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

[10] “The Standish Group Report, Chaos, 1995”, www.projectsmart.co.uk/docs/chaos_report.pdf.

[11] Stefan Jungmayr, “Identifying Test Critical Dependencies”, IEEE International Conference on Software Maintenance, 2002

[12] Mozilla on Microsoft Windows 32-bit Platforms, www.mozilla.org/build/win32.html

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

[14] 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.

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

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

Fawcett W. James (M’61–LM’04) received his PhD degree in Electrical Engineering from Syracuse University. His research interests include software complexity and developing infrastructure to re-engineer software reuse processes and make accessible, for reuse, not only code, but also documentation and test products.

Gungor K. Murat received the BS degree in industrial engineering from Sakarya University in Turkey in 1997, and received his MS degree in computer science from Syracuse University in 2001. Currently (2005) he is continuing his PhD study at Syracuse University. His research interests include static software analysis, software change and using software metrics to understand and improve static structure of large software systems.