SER3092

# ANALYZING STATIC STRUCTURE of LARGE SOFTWARE SYSTEMS

Based on Data from Open-Source Mozilla Project

James W. Fawcett, Murat K. Gungor, Arun V. Iyer

*Abstract* — ***Monitoring software quality in a development project is an important task required of project management, especially for large-scale projects. Our primary interest is evaluating quality of a system's structure and implications of the structure for project management, maintenance, and testing. Without the help of source code analysis tools it is difficult to understand a large project, evaluate its quality, and track progress effectively. In this paper we discuss application of tools developed for this purpose to the open-source Mozilla project.***

*Index Terms* — **Dependency analysis, metrics, open-source, software quality.**

## I. INTRODUCTION

Monitoring software quality [1] in a development project is an important task required of project management, especially for large-scale projects. This paper presents results of an empirical study of a large open-source project, Mozilla. Our primary interest is evaluating the quality of a system's structure and implications of the structure for project management, maintenance, and testing.

Without the help of source code analysis tools it is difficult to understand a large project, evaluate its quality, and track progress effectively [2]. In this paper we focus on analysis based on static type dependency between source code files. The tools, discussed here, are used to generate different chart-based and graphical views of a project's current state. The paper first presents the dependency model used and gives a brief description of our analysis tools. It then analyzes the Mozilla project.

Manuscript received April 21, 2005.

Dr. James W. Fawcett is with the Electrical Engineering and Computer Science Department, Syracuse University, Syracuse, NY 13244, USA (phone: 315- 443-3948; e-mail: jfawcett@twcny.rr.com).
Murat K. Gungor is with the Electrical Engineering and Computer Science Department, Syracuse University, Syracuse, NY 13244 USA (phone: 315-443-4003 ; e-mail: mkgungor@ecs.syr.edu).
Arun Iyer is with the Electrical Engineering and Computer Science Department, Syracuse University, Syracuse, NY 13244 USA (phone: 315-443-4003 ; e-mail: aviyer@ecs.syr.edu).

The results provide significant insight into the quality of the systems analyzed and provide some hints about how to improve their structure.

## II. DEPENDENCY MODEL

This paper is focused on dependencies between files, based on the types and global functions they contain. We do this because files are the units for analysis, management, and testing in most development organizations. 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.

Dependencies between software files are essential so that one component may provide services to another. However, dependencies complicate the process of making changes, perhaps to fix latent errors or performance problems, because of the effects a change may have on other files. When files each bind to many other files and mutual dependencies exist between them, maintenance and testing may become quite difficult to carry out effectively. It is not uncommon for a change in one file to precipitate a cascade of changes in other files, especially in the presence of mutual dependencies.

The dependency model used throughout this analysis 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
- 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].

There is one more important qualification we have to make about this model. In much of the code base we've analyzed for this paper there is a significant amount of code duplication across the directory structures analyzed. If we accept duplicate code our tool is not strong enough to sort out which instance is being referenced by another file, and so we will misclassify some dependencies. We've found these misclassifications lead to larger predicted mutual dependencies than are actually present in the system. Our solution currently is to eliminate all duplicate code. This results in some missed detections, but we want to err on the side of conservative estimate of coupling rather than too pessimistic estimate. Part of our future work will be directed to a more thoughtful handling of these ambiguities.

### *A. Architectural View of Dependency Analyzer (Depanal)*

DepAnal's goal is to find dependencies between C/C++ source code files based on static type analysis. Dependency relationships between the files are determined by the model described above, as in [1] [3] [4]. DepAnal makes two passes over each file in the project, as shown in Figure 1.

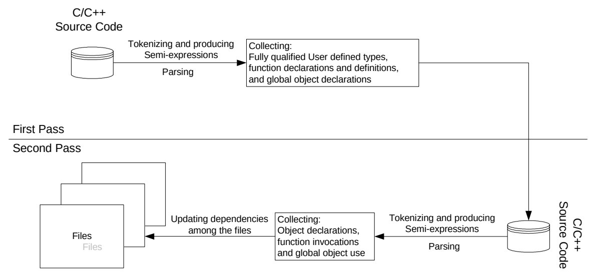
C/C++
Source Code
Tokenizing and producing Semi-expressions
Parsing
Collecting:
Fully qualified User defined types, function declarations and definitions, and global object declarations
First Pass
Second Pass
Files
Files
Updating dependencies among the files
Collecting:
Object declarations, function invocations and global object use
Tokenizing and producing Semi-expressions
Parsing
C/C++
Source Code

Figure 1 - Dependency Analyzer Architecture

First Pass: DepAnal processes each line of code in each file to capture user-defined types (class, struct, union, enum, typedef, etc…), global functions, and global object declarations. These are stored with fully qualified scope information for the second pass.

Second Pass: DepAnal again processes each line of code to search for creation of types, global function invocations, and global object use to find dependencies between files. Our approach is similar to [5] for data collection and analysis, but provide different types of data transformations, presentation, and analysis.

The DepAnal tool collects data from the source code with the help of a C/C++ tokenizer and semi-expression composer, shown in Figure 2. Semi-expressions are sequences of tokens that end with a semicolon, an open brace, or a closed brace, or, in the case of preprocessor statements, that end with a new-line. We have found that collecting semi-expressions as part of the scanning process tends to simplify code analysis. Collected information is stored in an STL based data container in memory.

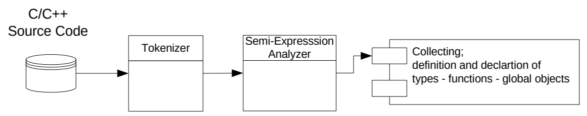
C/C++
Source Code
Tokenizer
Semi-Expresssion Analyzer
Collecting;
definition and declartion of types - functions - global objects

Figure 2 - Collecting data from source code

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 refactoring are needed.

We also developed three adjunct tools that provide additional views of the data:

1. Strong Component Analyzer: **SComp**[2] builds a dependency graph from the data provided by DepAnal and analyzes its strong components, that is, sets of files that are mutually dependent. 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. Size and Complexity Analyzer: **Anal**[3] 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.
3. DepView: Generates 2D graphical display of components and their dependency relationships.

What the DepAnal tool does not do: it ignores all macros and it makes no attempt to 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 II. 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

[1] We are planning to do this, along with several other extensions to our analysis, as later steps in our research.

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

[3] Perhaps we could have chosen a better name for this tool.

the results are accurate, within the limitations described in this paragraph.

## III. DEPENDENCY ANALYSIS RESULTS

The data presented in this section has been collected from the large open-source Mozilla project. All of our findings are based on the dependency model discussed in the previous section. We present several different views of the dependency data for both projects and draw some conclusions about what such data can disclose concerning a project's implementation.

The Mozilla project 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 [6]. 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 three 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 its related strong component density histogram.

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 [7] [8] [9] with results presented similar to those shown here. We focus on somewhat different aspects of program implementation than discussed in those papers.

### *A. Mozilla Data Collection*

We downloaded version 1.4.1 of the Mozilla Win32 configuration [10] [11]. 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 make files, setting environment variables, and settings in for the Visual Studio C++ compiler, used for all the builds for this paper.

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 some 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:

| | |
|---|---|
| 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 |

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 1 day on a PC with 1 Gigabyte of random access memory, running Windows XP Professional, with Pentium IV Processor.

### *B. Fan-In Data Extracted From Mozilla GKGFX Library*

Figure 3, 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.

A file with large fan-in is desirable from the perspective that it demonstrates high reuse of the types defined in that file[4]. 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. [12]

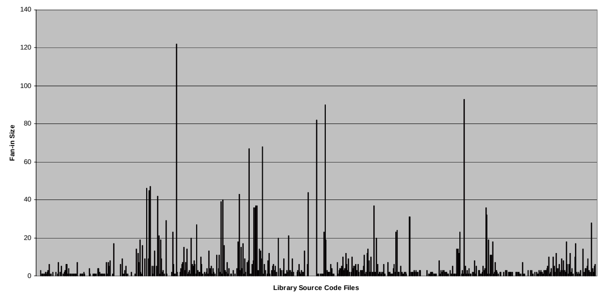

Figure 3 - Mozilla GKGFX Library Fan-in

There are scores of files, shown in Figure 3, 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

[4] We would expect to see high fan-in for some utility library routines, for example.

creates a high probability for consequential[5] change [13]. We have also looked at the wealth of change data provided by Mozilla's associated change data log to understand this process better.

In Figure 4 we show fan-in density for the same library. This is simply a histogram for the data in Figure 3. This plot shows that a large fraction 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.

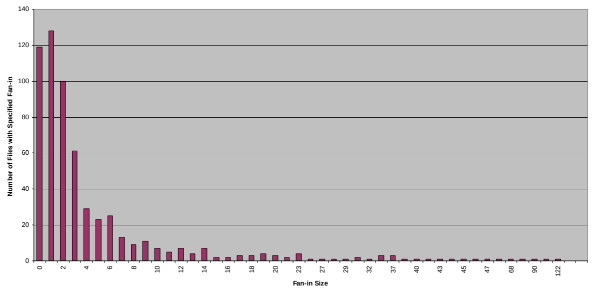

Figure 4 - Fan-in Histogram for Mozilla GKGFX Library

### C. Fan-Out Data Extracted from the Mozilla GKGFX Library

Fan-out for the GKGFX library is shown in Figure 5, below. 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 – file is taking responsibilities for many, perhaps only loosely related, tasks and needs the services of many other files to manage that.

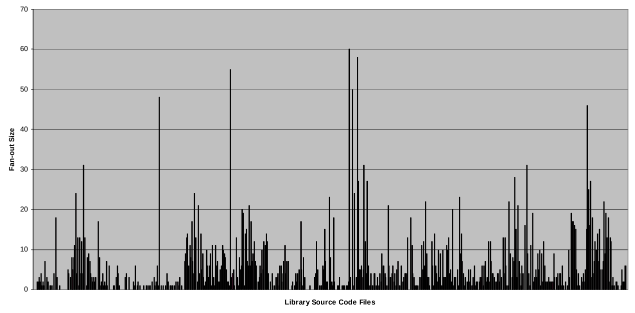

Figure 5 – Mozilla GKGFX Library Fan-out

Figure 6 shows a Fan-out histogram for the data in Figure 5. 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.

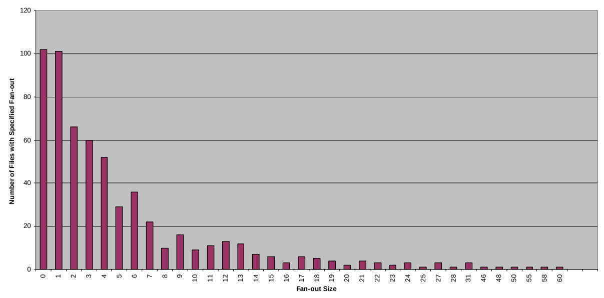

Figure 6 – Fan-out Histogram for Mozilla GKGFX Library

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

### D. Strong Components in the Mozilla GKGFX Library

A strong component is a set of source code files, elements in the dependency graph that are mutually dependent. Any given file from a strong component depends, either directly or indirectly6 , on every other file in the component. There can be no complete dependency ordering within a strong component, so there is no way to prepare a classic testing schedule based on testing only 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 7 shows a strong component histogram for the GKGFX library. There are many strong components of modest size, and one huge component, consisting of 119 files. 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.

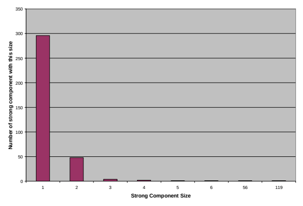

Figure 7 – Mozilla GKGFX Library Strong Components

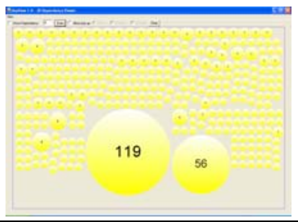

Figure 8 –GKGFX Strong Components

The dependency coupling that forms strong components may be due to the use of non-constant global data [14], 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 [13].

Another issue that this plot illustrates is the lack of well defined modules. The dependency model we use for this

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

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

analysis recognizes mutual dependencies between declaration and implementation of a type 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, Figure 8, we see that most of the files in this library do not fall into the classic module structure.

In Figure 9, each circle represents a strong component; number on the circle shows how many files are in that strong component. In the figure, largest strong component consist of 119 files, lines from center of the circle show fan-outs and lines coming the left corner of the circle show fan-ins to this component.

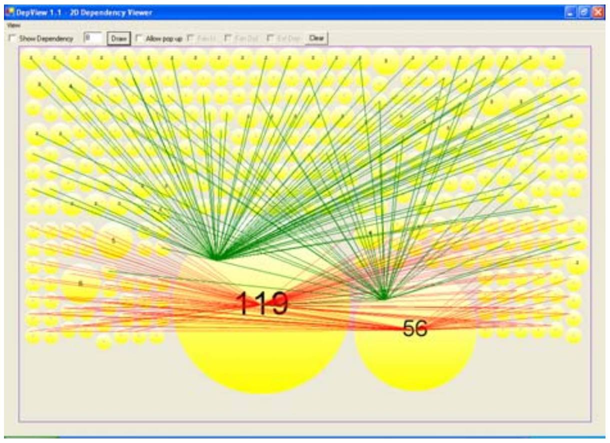

Figure 9 –This figure shows dependencies of only two of the largest strong components with other components.

In Figure 9, we show only external dependencies among components, besides this, there are large numbers of dependencies between the members of a component.

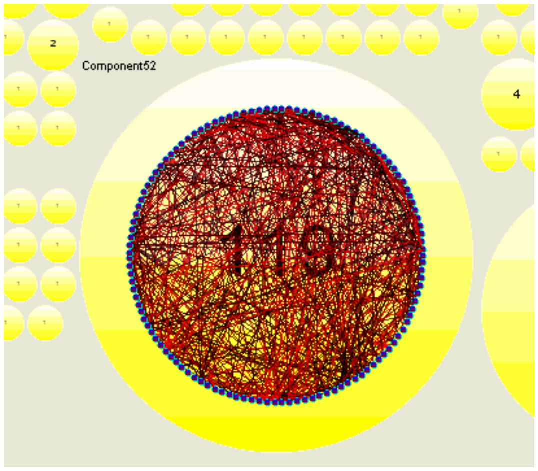

Figure 10 – Internal dependencies of Component 52, the largest GKGFX strong component.

If strong component size gets larger, it reduces the ability to adapt to new change, since change may result in further, consequential, unexpected changes. This reduces the gain from change, and management may no longer accept new changes after some point. This illustrates how an un-maintainable system may be created.

Figure 10 shows how files are densely connected to each other in the largest of the GKGFX libraries biggest strong component. This component will have a very large risk associated with its testing. Because of its internal dependency density, any change is likely to cause a cascade of consequential changes requiring further testing.

Including external fan-in and fan-out dependencies of the largest strong component in GKGFX, shown in Figure 11, reveals that if any other depended-upon component changes; Component 52 also needs to be tested to make sure that it still performs according to its requirements.

Approximately half of the dependencies shown in Figure 11, below are Fan-In dependencies on the largest strong component. Consequently all of those external files will suffer test risk inherited from the component due to their dependency.

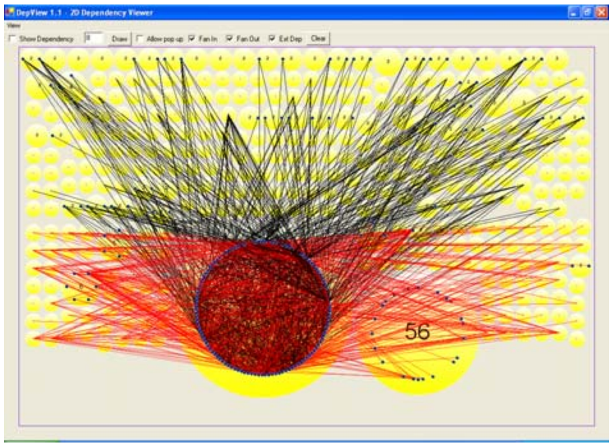

Figure 11 – Internal & External Fan-In and Fan-Out dependencies for largest GKGFX strong component

## IV. CONCLUSIONS

In summary, type-based dependency analysis appears to be a useful tool with which to direct implementation and testing of 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.
- 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.

## REFERENCES

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

[2] 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.

[3] 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

[4] 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).

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

[6] 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

[7] 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

[8] 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

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

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

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

[12] 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

[13] 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

[14] 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

**James W. Fawcett** (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.

**Murat K. Gungor** received his BS degree in industrial engineering from Sakarya University in Turkey, and received his MS degree in computer science from Syracuse University. 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.

**Arun V. Iyer** received his MS degree in Computer Engineering. His research interests include software quality metrics and re-factoring.