An empirical study of fine-grained software modifications

Daniel M. German
Software Engineering Group
Department of Computer Science
University of Victoria
Victoria, Canada
dmgerman@uvic.ca

Abstract

Software is typically improved and modified in small increments. These changes are usually stored in a configuration management or version control system and can be retrieved. In this paper we retrieved each individual modification made to a mature software project and proceeded to analyze them. We studied the characteristics of these Modification Requests (MRs), the interrelationships of the files that compose them, and their authors. We propose several metrics to quantify MRs, and use these metrics to create visualization graphs that can be used to understand the interrelationships.

1. Introduction

Configuration management systems, and more specifically, version control systems keep track of the modification history of a software project. The logs from these systems will keep track of who modifies what, when and what the change was. CVS, the Concurrent Versions System is arguably the most widely used version control management system. In this paper we used softChange to retrieve and analyze the CVS logs of several mature projects [7] (softChange is a tool that retrieves the history of a project, analyses and enhances it by finding new relationships amongst it, and allows users to navigate and visualize this information [6]). The first stage was to extract the file revisions in a way similar to [2]. A file revision corresponds to every modification a its corresponding file. CVS does not keep track of "transactions", and therefore, it is not possible to know which files were committed at the same time by a given author. softChange rebuilds these transactions, which we call *Modification Requests* (MR).

MRs provide a fine-grained view of the evolution of a software product. The majority of the modifications in a MR are to, either, implement small improvements to the source code, or to fix defects in the project.

1.1. Research questions

We started research by posing some questions that became the foundation for this study:

- What do typical MRs look like? Are MRs dedicated to fixing bugs different from MRs intended to add functionality to the project?
- Are MRs different in different stages of development?
- What is the effect of modularization of the code base in the composition of MRs?
- Do files tend to be modified by only one developer?
- Can we infer some type of social network from the modification patterns of a software project?
- Can we create metrics that describe MRs?
- How can we visualize MRs?

1.2. Related Work

In [11] Mockus et al. analyzed Mozilla and Apache in an attempt to quantify aspects of developer participation and interaction and defect density and problem resolution intervals. Fisher and Gall have described a CVS fact extractor in [2], and discussed the main challenges of creating a database of CVS historical data and then use this database to visualize the interrelationships between files in a project [3]. In [1] they analyzed the modification requests and describe the different types of logical coupling among the files included in the MR. Xia is a plugin for Eclipse that provides some visualization for CVS repositories[12]. Liu and Stroulia have developed JReflex, a plug-in for Eclipse for instructors of software engineering courses [10]. It is designed to compare the differences in development styles in different teams, who does what, who works on what part of the project, etc. JReflex is intended to be a management oriented tool for browsing the CVS historical data. Both Xia

and JReflex work at the file revision only, and do not consider files that are modified at the same time as part of the same commit operation. Hassan and Holt studied the complexity of software systems based on their source code history in [8].

1.3. Organization

This paper is divided in 7 main sections. We continue by describing our methodology. We then proceed to analyze modification requests and propose some metrics, and visualizations of MRs and their authors. In section 4 we concentrate on how files in a MR change. We conclude with a discussion of our results followed by future work and conclusions.

2. Methodology

The first part of our research was the retrieval of the historical information from CVS and the rebuilding of its MRs. As described in the previous section, CVS does not keep track of which files are modified together and therefore, the first task was to rebuild this information. softChange uses a heuristic that is based on a sliding window algorithm to rebuild MRs based on its component file revisions. This algorithm takes 2 parameters as input: the maximum length of time that a MR can last δ_{max}, and the maximum distance in time between two file revisions τ_{max}. This algorithm is depicted in figure 1. Briefly, a file revision is included in a given MR if a) all the file revisions in the MR and the candidate file revision were created by the same author and have the same log (a comment added by the developer when the file revisions are committed); b) the candidate file revision is at most τ_{max}seconds apart from at least one file revision in the MR; and c) the addition of the candidate file revision to the MR keeps the MR at most δ_{max}seconds long.

Most MRs take few seconds to complete. But some tend to be rather long. There are several factors that affect the duration of a MR. First, the size and number of files that compose the MR; second, the bandwidth available between the developer's computer and the CVS server (a slow link will slow down the time required to do the commit); and third, the load of the CVS server. In our experiments we have found that $\tau_{max} = 45s$ and $\delta_{max} = 600s$ are good values for these parameters (these values were used to extract the MRs discussed in this paper). A more detailed discussion of the extraction stage is presented in [4].

We decided to concentrate our attention in the evolution of the code base, hence, we looked into MRs that included source code files, which we call codeMRs. A codeMR is a MR that contains at least one source code file revision.

Based in our observations, codeMRs can be of different types:

```
// front(List) removes the front of the list
// top(List) and last(List)
//    query the corresponding elements of the list
// Initialize set of all MRs to empty
MRS = ∅
for each A in Authors do
    List = Revisions by A ordered by date
    do
        MR.list = {front(List)}
        MR.sTime = time(MR.list_1)
        while first(List).time − MR.sTime ≤ δ_max ∧
              first(List).time −
                  last(MR.list).time ≤ τ_max ∧
              first(List).log = last(MR.list).log ∧
              first(List).file ∉ MR.list do
              queue(MR.list, front(List))
        od
        MRS = MRS ⋃ {MR}
    until List ≠ ∅
od
```

Figure 1. Algorithm for the recovery of MRs

- Maintenance, e.g. defect fixing.
- Functionality improvement, e.g implementation of new features.
- Documentation. Since we are concentrating on source code, this includes changes in the comments of the files.
- Architectural evolution and restructuring, e.g. a major major change in APIs or the reorganization of the code base.
- Relocating code, e.g. a file is placed in a new directory, or a function is moved from one file to another.
- Branch-merging, e.g. code is merged from a branch or into a branch.

This list is not meant to be complete. Furthermore, these types are not mutually exclusive: a MR can include files with major changes in their documentation and those files might be moved to a different location at the same time. Documentation and architectural evolution MRs tend to include a large number of files. For example, we have discovered MRs including several hundred files in which a comment has changed (in one instance the license of the product changed; in another, the name of the company changed). Architectural changes might involve moving large amounts of code from one place to another, or changing APIs of important components that would require any file that uses that function to be changed.

We used the software project Evolution as a test case for our analysis (Evolution is a mail client for Unix sim-

ilar to Microsoft Outlook). softChange has been used to extract Evolution's historical logs, enhance them, and then query and visualize them. softChange helped us to understand how the project evolved, and how its developers collaborated [5]. We hoped that the understanding of the architecture and evolution of the project would helps in this research. In [5] we were interested in how the project has evolved at a course-grained level of detail while in this paper we are interested in the changes at a finer-grained level of detail.

The Evolution project was born in 1999. In Nov. 2003 we obtained a copy of its CVS repository (this copy did not include any internationalization files). It consisted of 5127 files, which had been modified 47814 times, by 148 contributors. We complemented our analysis with data from the CVS repositories of Mozilla–the multiplatform Web browser, PostgreSQL–a SQL database management system, and GNU gcc–the multi platform, multi language compiler.

3. Analysis of Modification Requests

We decided to identify two special subsets of codeMRs:

- **bugMRs**. bugMRs are codeMRs that correspond to a explicit defect fix as recorded by Bugzilla. Usually, if the MR fixes a bug, the developer will record the bug number in the log of the commit (see [4] for details).
- **commentMR**. commentMRs are codeMRs in which every one of its source code files was modified only in its comments. In order to determine if a file was modified only in its comments we used the following algorithm:
  - The previous and current file revisions of the file were recreated.
  - For each of the two file revisions generate their corresponding *clean* revision:
    - Re-indent the code (using the `indent` Unix command). With this step we expect to avoid false positives in which the code has been reformatted.
    - Remove comments and empty lines
  - If the previous and current revisions are identical, then we consider the revision a documentation change (the current algorithm will consider changes in white space only as documentation changes; a change in white space might reflect an attempt by the developer to make the code more readable, for example).

  If all the revisions in the MR are a documentation change, then the MR is considered a commentMR.

It is possible to state that the set of all documentation MRs contains the set of the commentMRs; and that the set of all maintenance MRs contains all the bugMRs

3.1. Number of Files in a MR

We started by testing the following hypotheses with respect to the number of files in a given MR:

- That bugMRs tend to contain few files; and
- that commentMRs tend to contain a large number of files.

Most MRs contain very few files. Figure 2 shows the distribution of the number of files in a MR for the projects Evolution, GNU gcc, PostgreSQL and Mozilla.

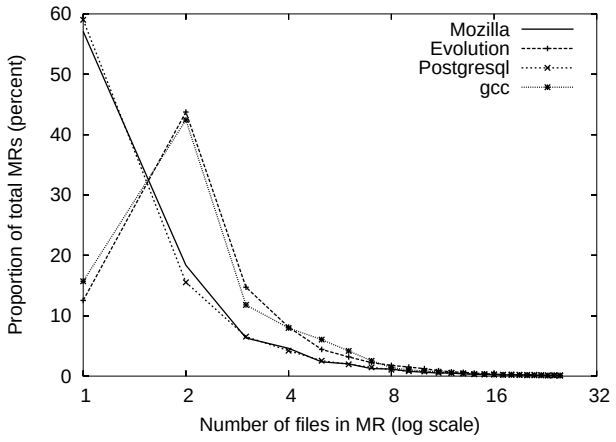

Figure 2. Number of files in a MRs in various projects.

The plot only shows MRs with 25 or less files. There are very few larger MRs (for example, in Evolution we detected a MR which included 650 files, and in Mozilla one that included 5838 files). Note that the four projects have two, almost identical curves. This effect was interesting enough to further explore. We discovered that the use of ChangeLog files (files that document the changes made to the software) accounted for this sharp difference. Evolution and GNU gcc use ChangeLogs, and almost every MR that includes two or more files includes a change to a ChangeLog file. Mozilla and PostgreSQL do not use them. When ChangeLogs are not taken into account all the curves look remarkably similar.

We looked more in detail into the MRs of Evolution. Figure 3 shows the distribution of the size of the 3 main subtypes of MRs. Note that most of the standard MRs (all MRs

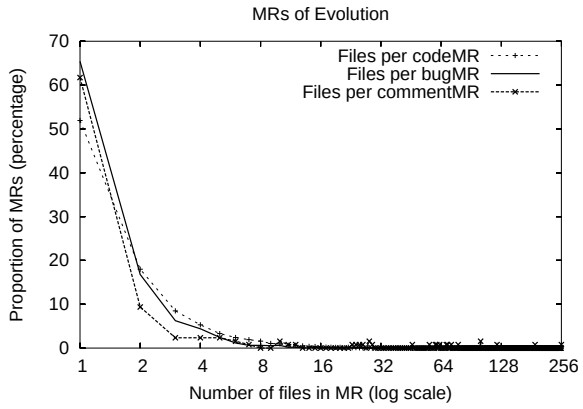

Figure 3. Number of files in different types of MRs

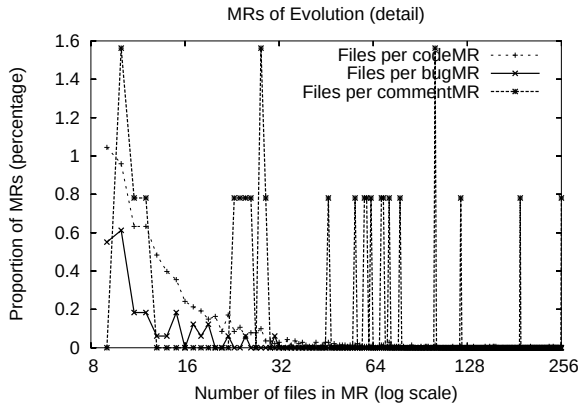

Figure 4. Number of files in different types of MRs (detail)

in the project) contain at least two files and most of them are small.

At first sight, all types of MRs follow a similar distribution, but there are remarkable differences. First, the largest bugMRs measures only 31 files only, while there are some commentMRs that contain a significant number of files (in one case 256). Figure 4 shows a detail of the tail of the curves. Note that large MRs tend to be commentMRs.

The average size of a MR is 4.5 files (13.64 standard deviation). The average size of a codeMRs is 3.42 files (10.23 standard deviation), while the average size of a bugMR is 2.95 files (2.44 standard deviation) and 13.48 files (35 standard deviation) for commentMRs.

We can conclude that the MRs of Evolution support our 2 hypotheses: MRs fixing defects typically contain few files, and documentation MRs tend to contain more files than other types of MRs.

3.2. Modification Coupling of Files in codeMRs

In [1] Fisher and Gall argued that historical modification logs can be used to detect a coupling relationship between two files: if two files are modified at the same time, then these two files are related.

We decided to analyze when and how the same two files were modified together. For this purpose we define three metrics:

- *Modification coupling*. The modification coupling between 2 files A and B is the likelihood that if file A is modified, then file B will also be modified. Formally, we define modification coupling between two files A and B as the proportion of the changes to A in which B is also modified (when A and B are part of the same MR). We denote the modification coupling of a and b as $M(a, b)$. M is, therefore, non reflexive.
- *Frequency of Modification*. The frequency of modification of a file A $Frequency(A)$ is the number of MRs in which the file A is modified. For a pair of files A, B, $Frequency(A, B)$ corresponds to the number of MRs that contain both A and B.
- *Modification neighbors*. The set of modification neighbors of a file A $Neighbors(A)$ is the set of all the files that have been modified in a MR in which A is part of.

For coupling analysis, we wanted to concentrate on two types of codeMRs: maintenance and functionality improvement. We did not want to consider modification coupling in documentation or restructuring changes because they tend contain a large number of files and their semantic relationship is weak. We proceeded to select codeMRs based in the following rules:

1. Eliminate codeMRs that contain files that were deleted in that MR. Unfortunately CVS does not support moving files. The user has to delete and create a new file in a new location. In order to avoid code restructing MRs in which files are moved around we proceeded to eliminate codeMRs that contained a deletion of a file.
2. Eliminate codeMRs that include the first version of a file. It is frequent that the first revision of the file is the result of importing a group of files into the project, or the result of moving a file (see previous item). We decided to avoid any MR that included a first version of a file.
3. Do not use codeMRs in branches. Branches in CVS pose difficult problems for researchers [2]. One of the reasons is that it is not explicit when code from the trunk is committed to the branch, or when code from the branch is committed back into the trunk. If

branches are not taken into account properly, a change to a file might be taken into account twice: once when it goes into the branch, and once when it goes back into the trunk. Furthermore, branch-merging into the trunk will most likely include many files (they are all related, but we are interested in the finer-grained relationships of each MR that compose a branch, rather than the entire one). Because branch-merging detection is expensive and error prone we have decided not to take into consideration any revision committed to a branch.

4. Eliminate codeMRs that contain a relatively large number of file revisions. This step will attempt to remove from our analysis MRs that are more likely to be documentation, architectural, or branch-merging codeMRs (we kept codeMRs with at most 20 source code files).
5. Eliminate commentMRs
6. Add all bugMRs.

To facilitate our analysis and to avoid trends early in the development of the product, we selected only MRs from 2002. In 2002 there were a total 3094 MRs, of which 2261 were codeMRs with at least 2 files and 155 were bugMRs with at least 2 files. Finally, there were 94 commentMRs in this period.

The total set of MRs in our analysis included a total of 1557 codeMRs. We will call this the *working set*. Only 645 of these MRs contained more than one file.

Modification coupling depends heavily on the programming language of choice. Given that Evolution is written in C, we were interested to verify the following hypothesis:

- For most .h files and their corresponding .c file (`file.h` and `file.c`), $M(file.h, file.c)$ will tend to be close to 1 (i.e. usually if `file.h` is modified, `file.c` is also modified).

We proceeded to compute the modification coupling for all .h files and their corresponding .c file. Figure 5 shows the distribution of the frequency of modification couplings for every pairs of candiate files. The plot clearly shows that most .h files are modified with their corresponding .c file. It is important to mention that .h files are also modified alone, and in our observations most of these modifications are to constants (mainly strings).

Other important questions are whether files are changed usually in groups, and whether changes to files tend to be localized to the same module (we define a module as each of the main directories of the source code). We wanted to verify the following hypotheses:

- A file tends to be modified with the same files.

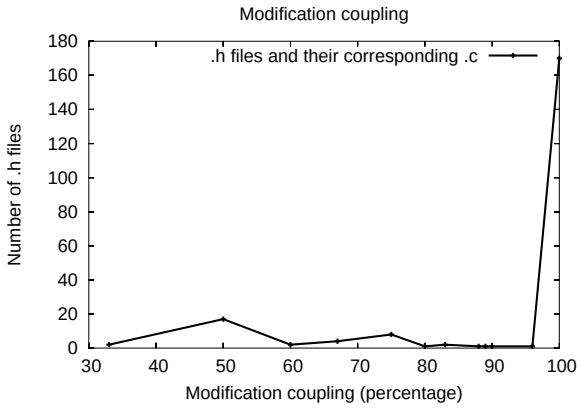

Figure 5. Modification Coupling of .h files and their corresponding .c file

- Most MRs are composed of files that belong to the same module.

To test these hypothesis we proceeded to create a coupling graph. A coupling graph depicts modification coupling in a given period. Every file modified in the period corresponds to a node, and its edges point to all of its modification neighbors. The edges of this graph are labeled with the modification coupling of its nodes, and with the frequency of their modification. A node has as attributes its frequency of modification, and the number of its neighbors. Nodes are clustered based on the module to which they belong.

The graph for the entire working set was too large and busy to include here. Instead we decided to concentrate in smaller periods. There were two interesting periods around 2002/11/07, when version 1.2.0 was released (a major stable release). The month of October 2002 was spent making sure that there were no errors in the version, while most of November was spent adding features for the next new unstable release. We will refer to October 2002 as the *Maintenance* period, and to November 2002 as the *Improvement* period.

Figure 6 shows the coupling graph for the improvement period and figure 7 shows the coupling graph for the maintenance period. In the visual representation of the coupling graphs nodes are colored according to the module they belong, and the width of the edge between two nodes A,B corresponds to its $Frequency(A, B)$.

We can make the following observations:

- The maintenance period has fewer MRs than an improvement period.
- The graphs tends to be composed of small disconnected subgraphs, or clusters of nodes interconnected

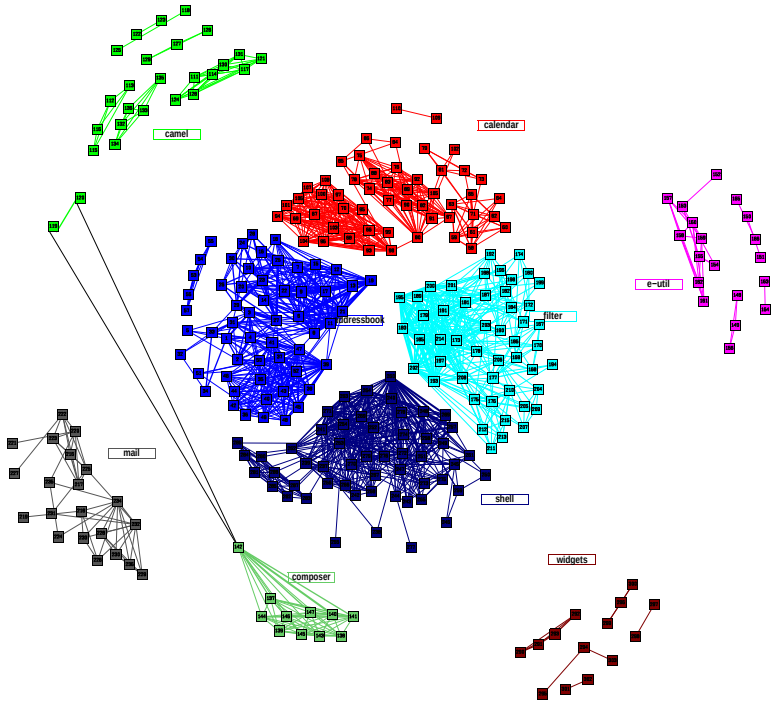

Figure 6. Modification coupling graph for *Improvement* period.

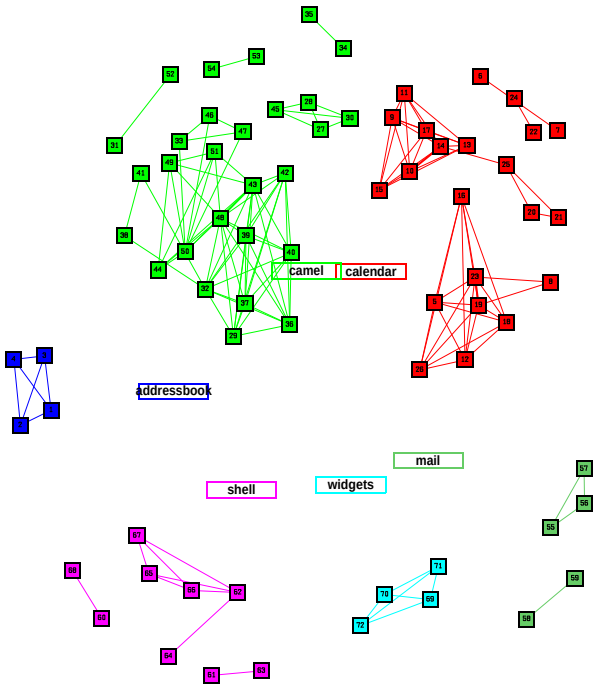

Figure 7. Modification coupling graph for *Maintenance* period.

by few edges. This suggests that a file is modified along a small set of other files.

- The modularization of the project has a profound impact in the disjointness of the different subgraphs. Notice that in the maintenance period the is no MR that spans two modules, and in the improvement period only 2 modules are interconnected by a total of 3 files. It is believed that the success of an open source project depends on the ability of its maintainers to divide it into small parts in which contributors can work with minimal communication between each other and with minimal impact on the work of others [9]. This division of work seems to be based on the modularization of the project.

3.3. Authorship

We were also interested in understanding how authors are related to modification coupling. We were interested in answering the following questions:

- How many people tend to modify a single file?
- How many authors contribute to a given module?
- Can we infer some type of social network from the modification patterns of a software project?

We proceed to define an authorship graph. In an authorship graph there were two types of nodes: files, and authors.

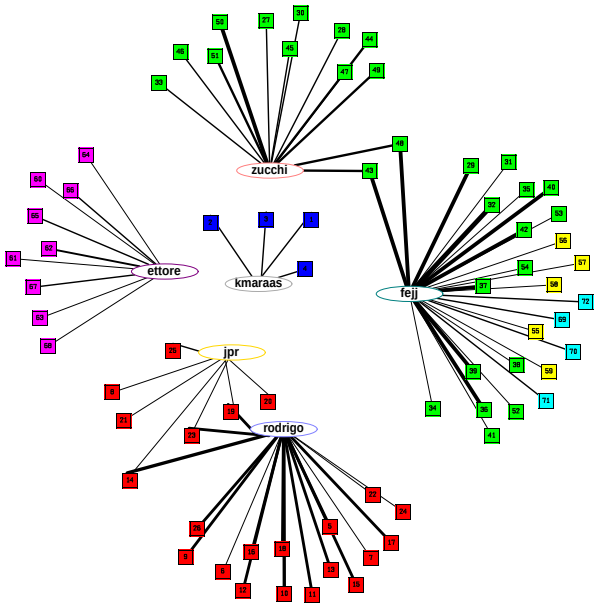

Figure 8. Authorship graph during *Maintenance* period.

Mod	Id	Prop	Acc
shell	ettore	0.65	0.65
	danw	0.11	0.76
	toshok	0.05	0.81
calendar	jpr	0.40	0.40
	rodrigo	0.32	0.72
	ettore	0.07	0.79
camel	fejj	0.66	0.66
	zucchi	0.25	0.91
	danw	0.03	0.94
addressbook	toshok	0.57	0.57
	clahey	0.13	0.70
	ettore	0.09	0.79

Table 1. Top 3 programmers of some the most active modules during 2002. The first column shows the name of the module, the second shows the userid of the programmer and the last two the proportion of their MRs with respect to the total during the year.

An edge was created between a file F and an author A if A had modified the file F. The edge was then labeled with the number of times that A had modified F. File nodes were clustered according to the module they were part of. Figure 8 shows the authorship graph for the maintenance period, while figure 9 corresponds to the improvement period. We have previusly analyzed the MRs for the entire year (2002) in an attempt to determine who the core maintainers of each of the modules were [5]. The results are depicted in table 1.

We can make the following observations about these graphs:

- Most files are modified ("owned") by one individual.
- There is a correlation between the perceived core maintainer of the module and the author who is most connected to that module in the graph (the core maintainer of the *calendar* module had very few MRs during the two periods of observation. Further analysis of the changes (both in 2002 and 2003) suggests that he was relatively inactive during this period, but stayed as core maintainer of the module).
- The files in some modules are split into 2 clusters where each cluster is primarily modified by one author.
- Some pairs of authors tend to modify a large number of common files, but these files tend to be localized in one module.

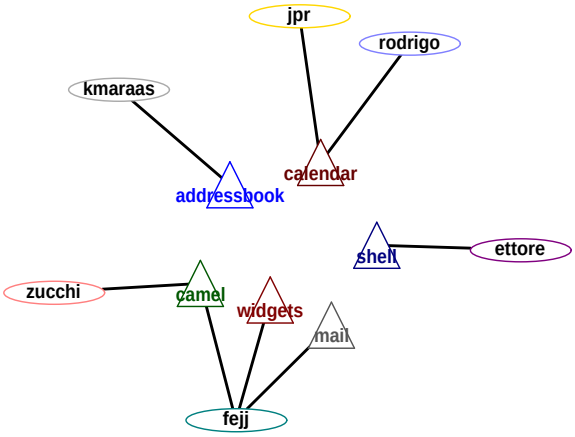

Figure 10. Friendship in Oct. 2002.

In order to understand the effect of modularization in these graphs, the files of each module were collapsed into a single node. The result can be seen in figures 10 and 11. Clearly, authors tend to modify code in few modules (some in one only).

The information of these graphs can be useful for multiple purposes. For example, it can help developers by making them aware of who are the people who tend to work in the same code as they do.

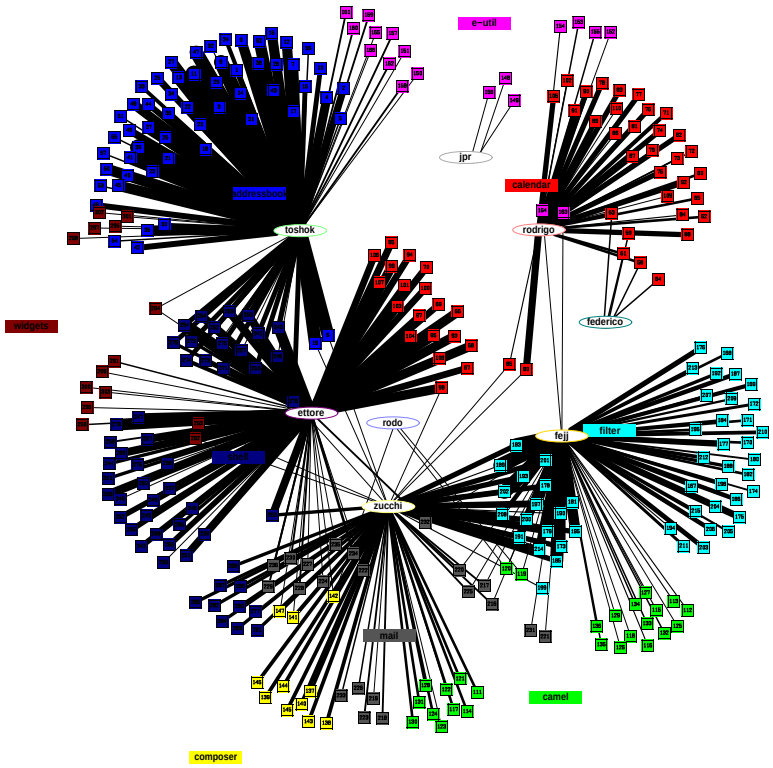

Figure 9. Authorship graph during *Improvement* period.

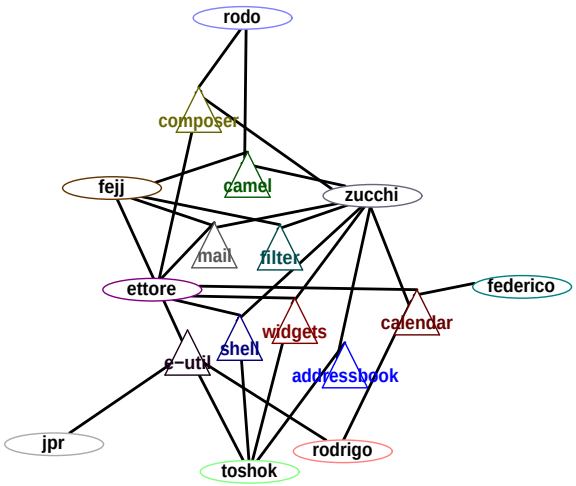

Figure 11. Friendship in Oct. 2002.

4. Analysis of the evolution of files

We were also interested in understanding how a given file evolves, and in particular, how its functions or methods (to which we will refer as functions from now on) change over time. We wanted to know when functions are added to a file, and when functions are removed from a file. This information could be useful to determine the type of MR that a file revision belongs to. We were interested in testing the hypothesis that:

- bugMRs tend to have fewer added or removed functions.

Doing a complete semantic analysis of each file revision is a very expensive operation and also depends on having a proper syntactic and semantic analysis tool for the programming language of the corresponding file. We decided to take a light-weight approach, in which a file was lexically analyzed in order to find the functions defined in the file. For this purpose we used Ctags (*exuberant tags* http://ctags.sourceforge.net/). Ctags "generates an index (or tag) file of language objects found in source files that allows these items to be quickly and easily located by a text editor or other utility. A tag signifies a language object for which an index entry is available (or, alternatively, the index entry created for that object)." Ctags supports many different languages, including C, C++, Java, Perl, Cobol, Pascal, Python.

The algorithm used to recover the functions added and removed in a file revision was:

- Extract the clean previous and current revision of the file (as described in 3, a clean revision is created by re-

indenting the original revision and then removing its comments and empty lines).

- Extract the set of the functions defined in each revision
- To create the set of removed functions subtract the functions defined in the current release from the ones defined in the previous release.
- To create the set of added functions subtract the functions defined in the previous release from the ones defined in the current release.

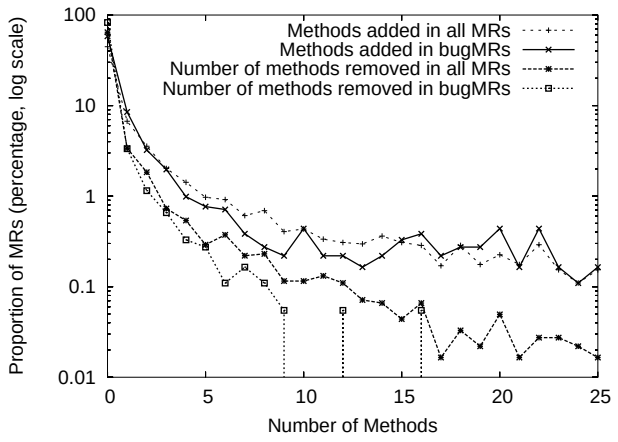

Figure 12. Number of functions added or removed in a bugMR.

The distribution of the number of functions added and removed for both is depicted in figure 12. The curves depict the number of methods added and removed from bugMRs and from any MR. The curves are very close to each other, but in both cases, the bugMR curve is (in most points) below the curves for all MRs, which provides support to our hypothesis that bugMRs tend to add and remove less functions than MRs in general.

We expect to conduct further studies in how functions are added, removed and modified as a software product evolves.

5. Discussion

The analysis described in this paper allows us to support some hypotheses related to the way that software changes at the modification request level. For example, we found that a defect fix involves usually few files or that periods of defect fixing have less MR activity than periods of feature improvement, and therefore less files are modified in the former period than in the latter.

We have been able to use the modification coupling of two files as a way to discover the interrelations between files: there are many instances in which two files tend to be modified together. Knowing which files tend to be modified together can be useful during the maintenance and evolution of the project. In many cases the related files are obvious (for example, a corresponding .c file is usually modified when the .h file is modified). But there are cases, for example, a file A that calls a function defined in B, or even more subtle, in which both files A and B are modified because the user interface of the product had some improvements that affected both files. In the latter case semantic analysis might not discover this relationship (there might be no call from one a function in A to a function in B).

The modification coupling and the authorship graphs provide interesting visualizations of the files being modified and the authors of the modifications. In particular, the authorship graph can be used to discover "developer" interactions based on the assumption that if two developers work in the same file, they need to communicate (formally or informally). It also depicts the importance of modularization. A well modularized product might reduce the required communication between developers.

6. Future Work

This work is an attempt to understand the characteristics of modification requests. The historical logs of software projects a wealth of data that can be used to further our understanding on how software changes.

In particular, similar studies like the one described here are needed, looking at different software products, in order to validate some of the hypotheses posed here. And the creation of new hypothesis is also needed, for example, does software written in C have modification patterns similar or different from Java.

The visualization of these data is also an interesting area of research. One area we are particular interested in is the animation of the authorship and modification coupling graphs, with the intent of showing how the file's and author's interactions evolve through time.

Finally, more work is needed in the creation of metrics for the analysis of modification requests, and how these metrics can be used to compare different projects, or to assist management and programmers in their daily activities.

7. Conclusions

The empirical study described herein attempts to provide a view into how software changes in a fine-grained approach. By looking at modification requests we have tried to understand how authors interact and how files are interrelated.

We first provided a rough categorization of MRs into: code, maintenance, functionality improvement, documentation, architectural and re-organizational, and branching. We have also provided different metrics to quantify files in MRs: modification coupling, frequency of modification, and modification neighbors. Based on these metrics, we created two graphs: the modification coupling graph (which shows the files that are neighbors) and the authorship graph (which shows who are the people who modify a given file).

We used the historical data of several open source projects for our analysis (primarily Evolution) to demonstrate the feasibility of our approach. softChange was used to extract their MRs and defect information.

We then proceeded to select a subset of MRs that were either maintenance or functionality improvement, and looked in detail into their characteristics. We also selected two periods of development: one dedicated to maintenance, and one dedicated to functionality improvement and compared their corresponding graphs.

Finally, we described a light-weight method to detect additions or deletions of functions in a given file revision (and therefore in a MR) by using lexical analysis.

Acknowledgments

This research was supported by the National Sciences and Engineering Research Council of Canada, and the Advanced Systems Institute of British Columbia. The author would like to thank the Evolution development team and the reviewers of this paper for their valuable comments.

References

[1] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering*, pages 90–101. IEEE Press, November 2003.

[2] M. Fischer, M. Pinzger, and H. Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32. IEEE Computer Society Press, September 2003.

[3] M. Fisher and H. Gall. MDS-Views: Visualizing problem report data of large scale software using multidimensional scaling. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, September 2003.

[4] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories*, pages 17–21, May 2004.

[5] D. M. German. Using software trails to rebuild the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, to appear, 2004.

[6] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softChange. In *Software Engineering Knowledge Engineering (SEKE'04)*, June 2004.

[7] D. M. German and A. Mockus. Automating the Measurement of Open Source Projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, May 2003.

[8] A. E. Hassan and R. C. Holt. The chaos of software development. In *Proceedings of IWPSE 2003: International Workshop on Principles of Software Evolution*, September 2003.

[9] J. Lerner and J. Triole. The Simple Economics of Open Source. Working Paper 7600, National Bureau of Economic Research, http://papers.nber.org/papers/w7600, March 2000.

[10] Y. Liu and E. Stroulia. Reverse Engineering the Process of Small Novice Software Teams. In *Proc. 10th Working Conference on Reverse Engineering*, pages 102–112. IEEE Press, November 2003.

[11] A. Mockus, R. T. Fielding, and J. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[12] X. Wu. Visualization of version control information. Master's thesis, University of Victoria, 2003.