T U M

I N S T I T U T F Ü R I N F O R M A T I K

Proceedings of OOPSLA 2003: 1st Workshop on Open-Source Software in an Industrial Context (OSIC'03)

Marc Sihling (Ed.)

TUM-I0319
Dezember 03

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

TUM-INFO-12-I0319-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

Table of Contents

Modeling Open Source Software Development Processes: Issues and Experiences

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425 USA
949-824-4130, 949-824-1715 (fax)
Wscacchi@uci.edu
http://www.ics.uci.edu/~wscacchi

Overview

This study presents selected analyses and findings from a multi-year study into the development processes, work practices, and community dynamics that arise in open source software development (OSSD) projects. Previous results from this study have identified socio-technical development processes that shape OSSD projects [13]; the use of software informalisms as both OSSD artifacts and communication media in developing the requirements for OSSD projects [14]; investigation of the comparative advantages that arise in OSSD versus traditional software engineering [15]; and others [1, 16-18].

In this paper, emphasis is directed to presenting preliminary findings as to the kinds issues that arise in the modeling of the techno-social processes found in different OSSD projects. We highlight results from the examination, discovery, and modeling of software engineering processes [15] within the Apache HTTP (Web) server, Mozilla Web browser, and in particular, the NetBeans integrated development environment (IDE) project as an example of an OSSD project in an industrial context.

It is important to qualify the findings reported here in ways that have previously lacked adequate attention. In particular, it is important to recognize that efforts like Apache and Mozilla are *composite projects* that consist of many loosely-coupled component system development projects. For example, the Apache Software Foundation consists of a dozen or so top level projects such as the HTTP Server, ANT, Cocoon, and Jakarta, while each of these contains their own sub-projects. For instance, the Jakarta project, focusing on the creation of commercial-quality, server-side solutions using Java, contains about two dozen sub-projects, such as Tomcat and Lucene. The Apache HTTP Server also has multiple system component sub-projects. Therefore, what is observed and reported as findings in one sub-project may not be indicative of the practices or processes found in other sub-projects within its parent projects, or the overall composite project. Such a condition therefore merits consideration when examining the results from studies of large composite projects that have been published elsewhere, so

as to appropriately gauge or qualify the scope of the reported findings.

In the remainder of this paper, attention is focused to issues and findings from an effort to empirically discover and model software engineering processes that can be observed arising in particular OSSD. In this case, the choice is to examine issues arising in the NetBeans IDE projects. NetBeans is more of a large-scale (monolithic) OSSD project that we find in many ways resembles traditional distributed software engineering projects, compared to common small-scale OSSD projects, or to the composite OSSD projects like Apache, Mozilla, and Linux.

OSSD Process Discovery

Previous studies that report on OSSD projects like Apache Web server and Mozilla Web browser [6, 12] provide informal narrative descriptions of the overall software development process that typify how these systems are developed. However, they do not treat these processes in terms that associate tools or the roles that people play in performing different development tasks. Similarly, they do not identify which project or Web site repositories are employed to store the artifacts that document or embody these systems [cf. 14]. Finally, neither Apache, Mozilla, NetBeans, or any other OSSD projects that have been examined provide documents on their project Web sites that explicitly describe what development processes are employed in the projects, nor how they are to be performed. As a result, any software engineer or developer seeking to either join one of these projects, or to start an independent OSSD project, cannot find explicit descriptions or models of the software processes that are performed to support the production of OSS systems.

Instead, if one seeks to understand and perform OSSD processes, then these processes must be discovered either through ad hoc trial-and-error, or by systematic investigation, Web site navigation, coding, and modeling. The former choice offers no leverage, while the latter implies labor-intensive research methodology, such as virtual ethnography [3, 14, 19]. The latter approach is taken here.

To capture and code the results from such an inquiry, it is necessary to organize and associate (e.g., hyperlink) observed development roles, tools, and development tasks. One such method for doing this is to employ a *rich picture* [7] to visualize the observed associations in the context of a specific project. In the rich picture of the NetBeans OSSD project shown in Figure 1, project participants play roles including Users, CVS administrator, Web site administrator, the Release Manager, The Board, SUN Microsystems, etc. Associated with each participant role are their stakeholder concerns or interests (indicated in the bubble clouds), development activities (download release, report bugs, etc.), tools (CVS, etc.) and associated artifacts (software version release, bug reports, etc.) as hyperlinked Use Cases (not shown).

Together, the rich picture and the Use Cases serve to capture and depict contextual features of the NetBeans development effort in terms of the processes, concerns, and overall structure of relations that serve as a prelude to construction of a formal model. Aspects of the resulting model are described in the next section.

Beyond this, a companion effort describes recent progress in the development of automated mechanisms to support and streamline this process discovery effort [4].

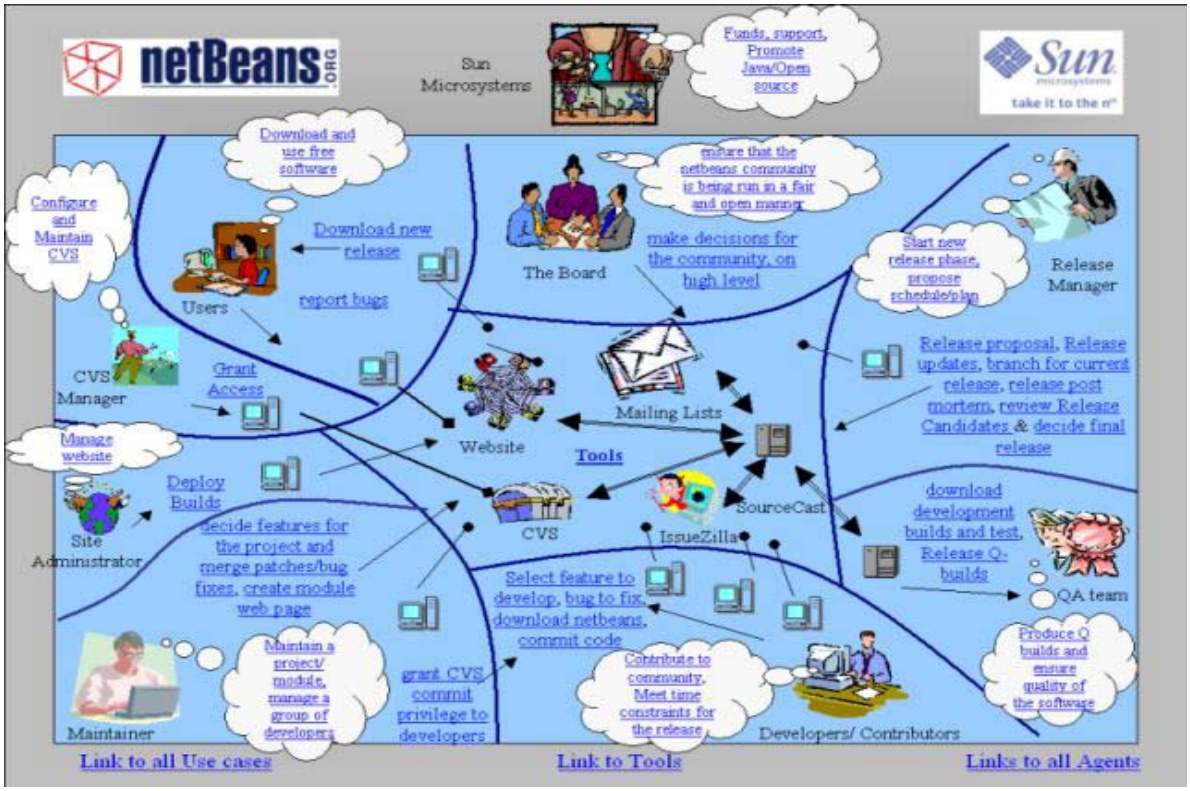

Figure 1. A rich picture depicting relationships between developer (agent) roles, artifacts, and tools associated with the software release process observed in the NetBeans project.

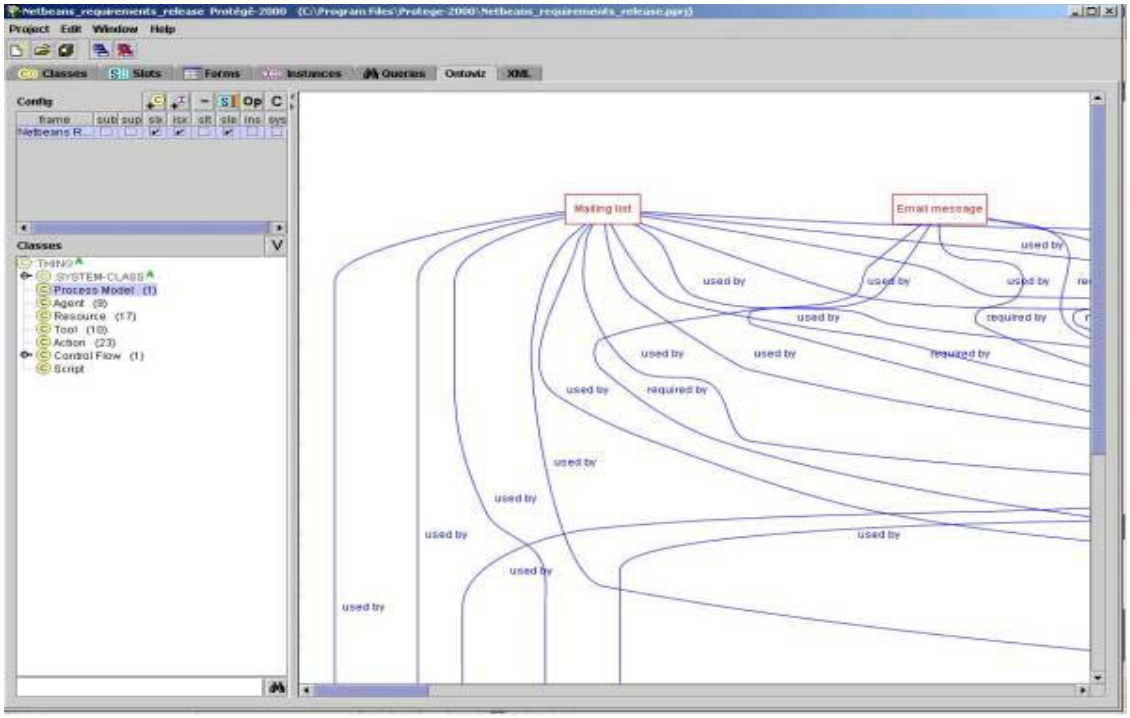

Figure 2. A partial view of a Protégé-based model of objects (artifacts, tools, roles) and activities (links) that form a formal model of the software release process for the NetBeans OSSD project.

OSSD Process Modeling

The available studies of Apache and Mozilla projects noted above also do not provide either a visual flow or a formal computational model of these processes. As such, the available narrative descriptions of these processes can not be easily analyzed, compared, visualized, computationally enacted, or transferred for (re)use in other projects [cf. 9,14].

The goal of this study is to discover and codify empirically observed OSSD processes as formal, computationally enactable models using the PML process modeling language [9]. Software processes modeled in PML employ the ontology of a process meta-model as their semantic foundation [5], and this foundation has been installed and configured to operate within a semantic object modeling framework called Protégé [10]. Figure 2 displays a partial view of a formal model of the software release process within the NetBeans OSSD project that conforms to this process meta-model [2, 11].

Among other things, Protégé can be programmed to capture (manually), edit, and transform models of software processes into notational forms like PML, as well as XML and SQL. Further, it should be noted that Protégé is itself an OSS tool for creating knowledge base models and ontologies.

Overall, one of the principal goals for modeling software engineering processes is to establish informal narrative, semi-structured hypermedia (i.e., a Web-based rich picture image map), and formal computational renderings that can be analyzed, compared, and shared within both the research and practice communities. Modeling also provides the foundation for continuous process improvement. The OSSD community has up to this time not recognize the potential value or use of "open source" software process models. As a result, OSSD projects like NetBeans must rediscover or reinvent software engineering processes, rather than follow the practice of open sharing, modification (improvement), and redistribution of these processes in ways that should enhance the productivity and quality of OSSD projects. The effort introduced in this study perhaps marks a first step in this direction.

Conclusions

This paper introduces and examines some of the issues that arise when seeking to discover and computationally model the software engineering processes that arise in a sample of different open source software development projects. Capturing and modeling of these processes may help developers new to an established composite OSSD project, or starting a new OSSD project, to more rapidly learn and become productive in the enactment of these processes. The discovery and modeling of these processes can also be an effective enabler of continuous process improvement techniques, as well as serve as another coordination resource that globally distributed developers can employ to stabilize and synchronize their loosely couple software development activities, roles and artifacts.

Acknowledgements

The research described in this report is supported by grants from the National Science Foundation #IIS-0083075, #ITR-0205679 and ITR#0205742. No endorsement implied. Contributors to work described in this paper include John Georgas, who tailored the Protégé tool for use in software process modeling; Mark Ackerman at the University of Michigan Ann Arbor; Les Gasser at the University of Illinois, Urbana-Champaign; John Noll at Santa Clara University; Margaret Elliott, Chris Jensen, Mark Bergman, and Xiaobin Li at the UCI Institute for Software Research; and Julia Watson at The Ohio State University are also collaborators on the research project described in this paper.

References

1. ELLIOTT, M. and SCACCHI, W.: Free Software Development: Cooperation and Conflict in A Virtual Organizational Culture, to appear in S. Koch (ed.), *Free and Open Source Software Development*, IDEA Press, 2004.

2. GEORGAS, J.: Software Process Modeling with Protégé, Institute for Software Research, UC Irvine, May 2002.

3. HINE, C.: *Virtual Ethnography*, SAGE Publishers, London, 2000.

4. JENSEN, C. and SCACCHI, W.: Simulating an Automated Approach to Discovery and Modeling of Open Source Software Development Processes, *Proc. Software Process Simulation and Modeling Workshop* (ProSim'03), Portland, OR, May 2003.

5. MI, P. and SCACCHI, W.: A Meta-Model for Formulating Knowledge-Based Models of Software Development. *Decision Support Systems*, **17**(4), 313-330, 1996.

6. MOCKUS, A., FIELDING, R. and HERBSLEB, J.: Two Case Studies on Open Source Software Development: Apache and Mozilla, *ACM Trans. Software Engineering and Methodology,* **11**(3), 309-346, 2002.

7. MONK, A. and HOWARD, S.: The Rich Picture: A Tool for Reasoning about Work Context, *Interactions*, 21-30, March-April 1998.

8. NOLL, J. and SCACCHI, W.: Supporting Software Development in Virtual Enterprises, *J. Digital Information*, **1**(4), February 1999.

9. NOLL, J. and SCACCHI, W.: Specifying Process-Oriented Hypertext for Organizational Computing, *J. Network and Computer Applications*, **24**(1), pp. 39-61, 2001.

10. NOY, N.F., SINTEK, M., DECKER, S., CRUBEZY, M., FERGERSON, and R.W. MUSEN, M.A.: Creating Semantic Web contents with Protege-2000, *IEEE Intelligent Systems*, **16**(2), 60-71, Mar/Apr 2001.

11. OZA, M., NISTOR, E., HU, S., JENSEN, C., and SCACCHI, W.: A First Look at the Netbeans Requirements and Release Process, Institute for Software Research, UC Irvine, June 2002.

12. REIS, C.R. and FORTES, R.P.M.: An Overview of the Software Engineering Process and Tools in the Mozilla Project, *Proc. Workshop on Open Source Software Development*, Newcastle, UK, February 2002.

13. SCACCHI, W.: Software Development Practices in Open Software Development Communities, *Proc. 1st. Workshop on Open Source Software Engineering,* Toronto, Ontario, May 2001.

14. SCACCHI, W.: Understanding the Requirements for Developing Open Source Software Systems, *IEE Proceedings - Software*, **149**(1), 24-39, 2002a.

15. SCACCHI, W.: Process Models in Software Engineering, in J. Marciniak (ed.), *Encyclopedia of Software Engineering, Second Edition*, 993-1005, Wiley, 2002b.

16. SCACCHI, W.: *Open EC/B: A Case Study in Electronic Commerce and Open Source Software Development*, technical report, Institute for Software Research, UC Irvine, July 2002c.

17. SCACCHI, W.: Free/Open Source Software Development Practices in the Computer Game Community, (submitted for publication), 2003.

18. SCACCHI, W.: When Is Open Source Software Development *Faster, Better,* and *Cheaper* than Software Engineering? to appear in S. Koch (ed.), *Free and Open Source Software Development*, IDEA Press, 2004

19. VILLER, S. and SOMMERVILLE, I.: Ethnographically Informed Analysis for Software Engineers, *Int. J. Human-Computer Studies*, **53**, 169-196, 2001.

Automatic Categorization Tool for Open Software Repositories

Shinji Kawaguchi† Pankaj K. Garg†† Makoto Matsushita† Katsuro Inoue†

†Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka
560-8531, Japan
{s-kawagt, matusita, inoue}@ist.osaka-u.ac.jp

††Zee Source
1684 Nightingale Avenue, Suite 201
Sunnyvale, California, 94807, USA
garg@zeesource.net

ABSTRACT

The world of Open Source software has demonstrated the remarkable appeal of *communal software development.* Large number of software projects can leverage, reuse, and coordinate their work through Internet and web-based technology. For example, SourceForge currently hosts about sixty thousand software systems. Similar strategies have been suggested for corporate software development, through notions like Corporate Source and Progressive Open Source [6, 7]

When used in a corporate setting, infrastructures for project information sharing present new opportunities. For example, one would like to know all projects that have something in common, so that the project groups can collaborate and share their work. With thousands of projects, manually locating related projects can be difficult. Hence, we propose to use automatic software categorization to find clusters of related software projects, using only the source code from projects. Our experiments with a small set of C programs demonstrates potential for automatic categorization of software systems without human aid.

1. INTRODUCTION

The rapid use of Internet and Web-based technology has given rise to a novel, global software archiving service, pioneered in the Open Source community through SourceForge [17]. More recently, several large corporations are realizing the benefits of such services for their own, proprietary software development. For example, Hewlett-Packard Company, IBM, Motorola, Nokia, and Xerox, are some of the corporations that are known to have deployed such archival service for their own internal corporate network.

For large software archives, categorizing their contents for browsing and searching is essential for effective utilization of the software archive. Automatic categorization would be helpful in several ways:

- Several *similar* software can be grouped together in a category for ease of browsing. For example, SourceForge [17] categorizes software according to their function (editors, databases, etc.), and also has the notion of *software foundries* for related software.
- Developers working on a software system may be informed about related software. Finding related software systems has following advantages.
  1. Developers can learn "best practices" and programming idioms from existing software systems. From related software systems, they can get strategies or hints for software evolution. They can even evolve their software systems based on related software systems, and not have to create it from scratch.
  2. Developers can leverage each other's work and promote more reuse. This becomes specially useful in situations like Corporate Source [7], where global groups in companies may not be aware of the relationship among their work [9].

In the past, such relationships have been determined by hand. Manual categorization generally requires deep understanding of not only the target software system, but also other software systems and their classification policy. With the increase in the number of software systems, e.g., SourceForge now has over sixty thousand software systems registered and continues to evolve, such manual identification is not enough.

Automatic categorization of software systems is a novel and intriguing challenge on software archive evolution. Past work in software engineering (e.g., see [4, 16]), has focused on determining *intra-component relations* of one given software system. We, however, propose finding *inter-component relations* of many software systems.

In this paper, we propose software automatic categorization system based on Latent Semantic Analysis(LSA). LSA is a method for extracting and representing the contextual-usage meaning of words by statistical computations applied to a large corpus of text [11]. LSA

has found a variety of uses ranging from understanding human cognition [11] to data mining [5]. Also, it is used for clustering components in a software system [14] and recovering document-to-source links [15].

We apply LSA for determining categories of software systems. We implemented the proposed method, and report on experimental results.

2. RELATED WORKS

Maarek et al. applied free-text indexing approach for software classification [13, 8]. They retrieved information from Unix man pages and classified Unix tools.

While quality and granularity of Unix man pages are highly uniform, the amount and quality of documents differ with software systems. Some software may have complete documentation, while others may have no or few documentation for their implementations.

From the viewpoint of retrieving information from source code, some existing clustering methods cluster one software into some functional parts for program understanding. Such software clustering methods use Latent Semantic Analysis [14], Self-Organizing Map [2], file structure and file names [1] or structure of program like call graph [3, 12].

In our previous work [10], we have tried measurement methods of software similarities. The hope was that we will relate "similar" systems together, and determine orthogonal categories like "editors," "databases," and so forth. We applied LSA to software systems and found that software similarity values are reflected only by *most influential aspects* of software systems. For example, the similarity value between database software with GTK interface and editors using GTK is very high. Although this phenomenon is not what we had hoped for, as be report in the rest of the paper, it is not necessarily bad.

3. CATEGORIZATION METHOD

The result of our previous work cited above indicates that software systems have multiple 'functional aspects'. Functional aspects are, for example, "compiler", "editor", "database", "runs on Windows", "supporting regular expression," and so forth. Consider an editor on Windows. This editor has not only "editor" functional aspect, but also "runs on Windows" functional aspect.

The software systems can be categorized with functional aspects on a nonexclusive basis. If a categorization is mutually exclusive, the categorization may capture only a few functional aspects.

We focus on identifiers (variable name, function name and so on) included in source code to retrieve a functional aspect. For example, "`gtk_window`" identifier represents some window, and source codes near the identifier would contain GUI operation.

As stated above, identifiers may represent a part of functions implemented in the program. If relationship between identifiers are found, they would represent one functional aspect. To determine relationships between identifiers, we use Latent Semantic Analysis(LSA), an information retrieval method explained below.

3.1 Latent Semantic Analysis (LSA)

Latent Semantic Analysis, LSA, is a practical method for the characterization of word meaning. LSA produces measures of word-word, and passage-passage relations which are well correlated with semantic similarity [11]. The method creates a vector description of documents. This representation is used for comparing and indexing documents, and various similarity measures can be defined.

Consider the six simple documents in Figure 1. In LSA, these documents are represented by a matrix shown in Table 1. Each column means a document and each row represents a word which may appear in the documents. Cell entries show the occurrence of the word in the document.

c1: Human machine interface for ABC computer applications
c2: A survey of user opinion of computer system response time
c3: Relation of user perceived response time to error measurement
m1: The generation of random, binary, ordered trees
m2: Graph minors IV: Widths of trees and well-quasi-ordering
m3: Graph minors: A survey

Figure 1: Example Input Documents

	c1	c2	c3	m1	m2	m3
computer	1	1	0	0	0	0
user	0	1	1	0	0	0
response	0	1	1	0	0	0
time	0	1	1	0	0	0
survey	0	1	0	0	0	1
trees	0	0	0	1	1	0
graph	0	0	0	0	1	1
minors	0	0	0	0	1	1

Table 1: An Example of LSA Matrix

Each row vector of this matrix indicates the characteristics of the word through the whole documents occurrences. This row vector can be used to determine the similarity of two words. A simple similarity definition used here is $cosine$ of two vectors.

In LSA, single value decomposition (SVD) is applied to the matrix. SVD is a form of factor analysis and acts as a method for reducing the dimensionality of the matrices. Why does LSA apply such translation? This is because a simple term-by-document matrix does not capture relationship among terms. Two documents show high similarity only when the documents have some same words; however, there are many synonyms. Thus similar documents do not always share completely same words. They may contain many synonyms. Using SVD, LSA can retrieve such undirectional relationship among documents. For more details, please refer [11].

3.2 Overview of Classification Method

Our method consists on 7 parts, explained in brief below:

1. Extract identifiers.

   First, we extract all identifiers from source code of software systems. We don't use reserved words in programming language and words in comments. Reserved words are meaningless from the viewpoint of function. Comments are abstract description, but amount and quality of comments in each software systems vary widely. Thus we cut out reserved words and comments.

2. Create identifier-by-software matrix.

   We create an identifier-by-software matrix, similar to the word-by-document matrix of Table 1.

3. Remove meaningless identifiers.

   Before performing LSA, we remove identifiers that appear in only one software system, or in more than half of software systems. Identifiers appearing in only one software are not meaningful in LSA. And, identifiers appearing in more than half of software systems are probably a general term and have no affect on categorization.

4. Perform LSA.

   We perform LSA for the identifier-by-software matrix without meaningless identifiers.

5. Compute $cosine$ of each identifiers and perform cluster analysis.

   From the matrix of LSA result, we compute $cosine$ values of each identifiers. Thereafter, we apply cluster analysis using calculated similarities. Cluster analysis is statistical analysis method that cluster individuals into clusters based on similarity among individuals.

6. Make software clusters from identifier clusters.

   From each identifier clusters, we retrieve software systems that contain one or more identifiers in the cluster, and make them a corresponding software cluster.

7. Make titles of software clusters.

   We obtain software clusters by previous steps, however, each software cluster needs description that explains what software systems are included. As titles, we use the ten highest score identifiers in the clusters.

4. EXPERIMENTS

As an implementation of our method, we created a prototype system. We experimented categorization of software systems using the prototype. The overall goals of our experiment were: Does our prototype categorize proper by target systems compared with existing manual categorization? Can our prototype categorize by the libraries use in the system?

4.1 Experiment Process

We collect sample data from SourceForge. We selected 41 C programs in five categories from SourceForge. The list of categories and software systems are in Table 2. Then we ran our categorization tool on the 41 programs.

4.2 Result

Table 3 shows a part of the categorization result by our method. Each row represents one cluster.

We got 40 clusters in total. The target systems in 18 clusters fall in the same categorization as SourceForge categorization. There are 8 clusters in which all software systems depend on same library or have same architecture. In top 20 clusters, 17 clusters fall in the same categorization in SourceForge or same library.

For example, cluster 3, 8 or 9 are clustered since the software systems in those clusters use the same library. Cluster 3 contains software systems using YACC(Yet Another Compiler-Compiler). Cluster 8 and 9 contains software systems using GTK. In the same manner, we can get the following clusters.

Cluster 22 Software systems using regular expression pattern matching library.

Cluster 25 Software systems implementing JNI(Java Native Interface).

Cluster 30 Software systems using getopt, a function which parses a command line argument.

Cluster 32 Software systems implementing Python/C.

Cluster 35 Software systems using YACC(Yet Another Compiler-Compiler).

On the other hand, there are 12 software systems that are not classified into any categories.

4.3 Discussion

Comparing with existing categorization, many clusters of our result follow existing categorization however, our result does not cover whole existing categorization. This is because our result has software systems that are not classified any categories. Our method categorizes based on appearance frequency of identifiers. This makes software systems that have few tokens tend to be not classified any categories.

We verify that our method can retrieve new categorization that are not considered by existing categorization. Such categorizations are: (1) by libraries (GTK, yacc, etc.), and (2) by depending architecture (JNI, Python/C, etc.). Our method does not need any human knowledge. Thus, if a new library appears, our method can follow such change automatically.

About cluster titles, there are some unidentifiable titles like cluster 1; however, cluster 4 and 6 have clear-cut titles “AVI” and “board, ply”. The clusters with software systems using same library tend to have clear-cut titles.

5. CONCLUSION

In this paper, we have proposed automatic categorization method for many software systems. Our method finds categorization clusters and classifies software systems based on the clusters. We have shown this method can classify without any knowledge about target software systems.

For future work, we will seek how to determine parameters and retrieving intuitive cluster titles. Furthermore, we will add large-scale experimentation. To do this, we need to improve system performance and scalability.

6. REFERENCES

[1] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *International Conference on Software Engineering,(ICSE'98)*, pages 84–93, Apr 1998.

[2] A. Chan and T. Spracklen. Discoverying common features in software code using self-organising maps. In *International Symposium on Computational Intelligence (ISCI'2000)*, Kosice, Slovakia, August 2000.

Category	Software
boardgame	Sjeng-10.0, bingo-cards, btechmux-1.4.3, cinag-1.1.4, faile_1_4_4, gbatnav-1.0.4, gchch-1.2.1, ics-Drone, libgmonopd-0.3.0, netships-1.3.1, nettoe-1.1.0, nngs-1.1.14, ttt-0.10.0
compilers	clisp-2.30, csl-4.3.0, freewrapsrc53, gbdk, gprolog-1.2.3, gsoap2, jcom223, nasm-0.98.35, pfe-0.32.56, sdcc
database	centrallix, emdros-1.1.4, firebird-1.0.0.796, gtm_V43001A, leap-1.2.6, mysql-3.23.49, postgresql-7.2.1
editor	gedit-1.120.0, gmas-1.1.0, gnotepad+-1.3.3, molasses-1.1.0, peacock-0.4
videoconversion	dv2jpg-1.1, libcu30-1.0, mjpgTools, mpegsplit-1.1.1
xterm	R6.3, R6.4

Table 2: The list of sample software systems

[3] K. Chen and V. Rajlich. Case study of feature location using dependency graph. In *8th International Workshop on Program Comprehension (IWPC'00)*, pages 231–239, Limerick, Ireland, June 2000.

[4] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, Jan 1990.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[6] J. Dinkelacker and P. Garg. Corporate Source: Applying Open Source concepts to a corporate environment (Position Paper). In *Proceedings of the 1st ICSE workshop on Open Source software engineering*, Toronto, Canada, 2001.

[7] J. Dinkelacker, P. Garg, D. Nelson, and R. Miller. Progressive Open Source. In *Proceedings of the International Conference on Software Engineering*, Orlando, Florida, 2002.

[8] W. B. Frakes and T. Pole. An empirical study of representation methods for reusable software components. *IEEE Transactions on Software Engineering*, 20(8):617–630, 1994.

[9] J. Herbsleb and A. Mockus. An Empirical Study of Speed and Communication in Globally-Distributed Software Development. *IEEE Transactions. Software Engineering*, 2003.

[10] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic categorization algorithm for evolvable software archive. In *2003 International Workshop on Principles of Software Evolution(IWPSE 2003)*, Sep 2003.

[11] T. K. Landauer and S. T. Dumais. Latent Semantic Analysis and the Measurement of Knowledge. In *Educational Testing Service Conference on Natural Language Processing Techniques and Technology in Assessment and Education*, princeton, 1994.

[12] G. A. D. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. D. Carlini. Comprehending web applications by a clustering based approach. In *Proc. of 10th International Workshop on Program Comprehension(IWPC'02)*, pages 261–270, Paris, France, June 2002.

[13] Y. S. Maarek, D. M. Berry, and G. E. Kaiser. An information retrieval approach for automatically constructing software libraries. *IEEE Transactions of Software Engineering*, 17(8):800–813, 1991.

[14] J. I. Maletic and A. Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'00)*, pages 46–53, November 2000.

[15] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering(ICSE2003)*, pages 125–135, Portland, OR, May 2003.

[16] R. Schwanke. An intelligent tool for re-engineering software modularity. In *Proc. of 13th International Conference on Software Engineering*, pages 83–92, Austin, Texas, USA, May 1991.

[17] SOURCEFORGE.net. `http://sourceforge.net`.

No.	Title of cluster	Software	The number of tokens
1	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc	8597
2	CASE_IGNORE, CASE_GROUND_STATE, screen, CASE_PRINT, CASE_BYP_STATE, Widget, TScreen, CASE_IGNORE_STATE, CASE_PLT_VEC, CASE_PT_POINT	xterm/R6.3, xterm/R6.4	2160
3	YY_BREAK, yyvsp, yyval, DATA, yy_current_buffer, tuple, yy_current_state, yy_c_buf_p, yy_cp, uint32	compilers/gbdk, database/mysql-3.23.49, database/postgresql-7.2.1	223
4	AVI, cinfo, OUTLONG, avi_t, AVI_errno, hdrl_data, OUT4CC, nhb, ERR_EXIT, str2ulong	videoconversion/dv2jpg-1.1, videoconversion/libcu30-1.0, videoconversion/mjpgTools	177
5	domainname, msgid1, binding, msgid2, domainbinding, pexp, __builtin_expect, transmem_list, codeset, codesetp	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1	165
6	board, num_moves, ply, pawn_file, npiece, pawns, moves, white_to_move, move_s, promoted	boardgame/Sjeng-10.0, boardgame/cinag-1.1.4, boardgame/faile_1_4_4	154
7	xdrs, blob, DB, UCHAR, XDR, mutex, key_length, logp, page_no, bdb	database/firebird-1.0.0.796, database/mysql-3.23.49	118
8	domainname, N_, binding, gchar, GtkWidget, PARAMS, codeset, gpointer, loaded_l10nfile, argz	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1, editor/gnotepad+-1.3.3, editor/peacock-0.4	118
9	GtkWidget, gchar, gpointer, gint, widget, gtk_widget_show, N_, g_free, dialog, g_return_if_fail	boardgame/gbatnav-1.0.4, editor/gedit-1.120.0, editor/gmas-1.1.0, editor/gnotepad+-1.3.3, editor/peacock-0.4	104
10	AOP, emitcode, esp, IC_RESULT, IC_LEFT, obstack, aop, mov, aopGet, IC_RIGHT	compilers/clisp-2.30, compilers/gbdk, compilers/sdcc	100
11	tuple, uint32, plan, int32, lsn, elm, rec, interp, TCL_ERROR, finfo	database/mysql-3.23.49, database/postgresql-7.2.1	79
12	xdrs, blob, DB, UCHAR, XDR, mutex, key_length, logp, page_no, bdb	database/firebird-1.0.0.796, database/mysql-3.23.49	73
13	UCHAR, relation, stmt, trigger, yyvsp, yyval, t_data, plan, dbname, USHORT	database/firebird-1.0.0.796, database/postgresql-7.2.1	68
14	fout, interp, TCL_ERROR, typ, YY_RULE_SETUP, List, DATA, Tcl_Interp, id, YY_BREAK	compilers/freewrapsrc53, compilers/gbdk, compilers/gsoap2, database/postgresql-7.2.1	50
15	GtkWidget, gchar, gpointer, dlg, gint, g_free, gtk_widget_show, gtk, GList, GTK_BOX	editor/gedit-1.120.0, editor/gmas-1.1.0, editor/gnotepad+-1.3.3	46
16	UCHAR, relation, stmt, trigger, yyvsp, yyval, t_data, plan, dbname, USHORT	database/firebird-1.0.0.796, database/postgresql-7.2.1	43
17	AOP, emitcode, mfp, ic, uchar, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT	compilers/gbdk, compilers/sdcc, database/mysql-3.23.49	36
18	adr, FX, word, stm, ED, xt, REF, prop, term, FP	compilers/gprolog-1.2.3, compilers/pfe-0.32.56	35
19	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc, database/firebird-1.0.0.796	31
20	dyn, FPRINTF, process_id, p_offset, ctl, rab, que, io_ptr, prior, PRINTF	database/firebird-1.0.0.796, database/gtm_V43001A_src_linux	29
21	dyn, FPRINTF, process_id, p_offset, ctl, rab, que, io_ptr, prior, PRINTF	database/firebird-1.0.0.796, database/gtm_V43001A_src_linux	27
22	regparse, dbp, mech, reginput, flagp, NOTHING, tuple, db, __P, regnode	boardgame/btechmux-1.4.3, database/leap-1.2.6, database/mysql-3.23.49	26
23	rectype, argp, rec, fileid, save_errno, data_len, qp, argpp, int4, dbp	database/gtm_V43001A_src_linux, database/mysql-3.23.49	26
24	AOP, emitcode, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode, iCode, etype	compilers/gbdk, compilers/sdcc, videoconversion/mjpgTools	26
25	jobject, JNIEnv, JNICALL, JNIEXPORT, jint, jstring, interp, TCL_ERROR, objv, TCL_OK	compilers/freewrapsrc53, compilers/jcom223, compilers/pfe-0.32.56, database/mysql-3.23.49	24
26	entrypoint, USHORT, TEXT, yyvsp, raddr, R, UCHAR, yyval, blob, REQ	compilers/clisp-2.30, database/firebird-1.0.0.796	17
27	int32_t, dbp, cinfo, net, unpack, argp, sinfo, cur1, purpose, mysql	database/mysql-3.23.49, videoconversion/mjpgTools	17
28	AOP, emitcode, mfp, ic, uchar, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT	compilers/gbdk, compilers/sdcc, database/mysql-3.23.49	16
29	USHORT, UCHAR, blob, REQ, NULL_PTR, hIcon, SCHAR, interp, wndclass, bdb	compilers/freewrapsrc53, database/firebird-1.0.0.796	16
30	optind, nextchar, __P, optstring, last_nonopt, option_index, uchar, optarg, pfound, dbp	boardgame/ttt-0.10.0, compilers/clisp-2.30, database/mysql-3.23.49	15
31	int4, ctl, tn, rec, semid, blkno, ti, oprtype, save_errno, AH	database/gtm_V43001A_src_linux, database/postgresql-7.2.1	14
32	notify, mech, PyObject, fargs, Node, Name, pset, zone, tprintf, NOTHING	boardgame/btechmux-1.4.3, database/postgresql-7.2.1	11
33	interp, notify, dbp, tuple, mech, PyObject, uint32, plan, int32, buff	boardgame/btechmux-1.4.3, database/mysql-3.23.49, database/postgresql-7.2.1	10
34	adr, stm, AOP, emitcode, operands, ASSERT, IC_RESULT, pred, lg, REF	compilers/gprolog-1.2.3, compilers/sdcc	9
35	yyvsp, yyn, PARAMS, codeset, domainname, msgid1, binding, msgid2, yylsp, domainbinding	boardgame/gbatnav-1.0.4, boardgame/gchch-1.2.1, compilers/clisp-2.30	9
36	ERREXIT, picture, pool_id, USHORT, get_buffer, output_buf, cinfo, xxx, UCHAR, streams	database/firebird-1.0.0.796, videoconversion/mjpgTools	9
37	REF, dyn, USHORT, vec, path_name, clause, STATUS, E, UCHAR, CSB	compilers/gprolog-1.2.3, database/firebird-1.0.0.796	8
38	AOP, emitcode, pfile, ic, IC_RESULT, IC_LEFT, aop, aopGet, IC_RIGHT, pic14_emitcode	compilers/gbdk, compilers/sdcc, database/postgresql-7.2.1	7
39	ic, ply, npiece, score, AOP, pawn_file, uchar, bking_loc, wking_loc, emitcode	boardgame/Sjeng-10.0, compilers/gbdk	7
40	clause, cinfo, pred, ci, Group, Np, word, X, A, tmp4	compilers/gprolog-1.2.3, database/postgresql-7.2.1, videoconversion/mjpgTools	6

Table 3: 41

A Software Production Infrastructure for the New Millennium

Alberto Sillitti
DIST - Università di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
ph: +39(010)353-2173
fax: +39(010)353-2154
alberto@dist.unige.it

Giancarlo Succi
Libera Università di Bolzano
Piazza Domenicani, 3
I-39100 Bolzano, Italy
ph: +39(0471)315-640
fax: +39(0471)315-649
Giancarlo.Succi@unibz.it

Tullio Vernazza
DIST - Università di Genova
Via Opera Pia, 13
I-16145 Genova, Italy
ph: +39(010)353-2793
fax: +39(010)353-2154
tullio@dist.unige.it

ABSTRACT
Software development involves several people with very different skills: customers, managers, accountants, business analysts, systems analysts, designers, developers, etc. Each one of them tends to focus on his or her specific aspect and to consider it the cornerstone of the whole development process. To involve these people in the software development there are many different views of the same knowledge base at the time of the involvement of a specific person. The build and synchronization of these views is a complex task and include a remarkable overhead in the process. This paper presents a set of problems and an overview of a possible solution to the development of an integrated development platform.

Categories and Subject Descriptors
D.2.6 [**Programming Environments**]: *Integrated environments*.

D.2.11 [**Software Architectures**]: *Domain-specific architectures*.

General Terms
Design, Languages.

Keywords
Software development, open source, tools integration.

1. INTRODUCTION
In software development, different people focus on different topics: developers focus on source code, designers focus on architecture, manager focus on progresses and team management, etc.

The code is what satisfies the customer needs and generates revenue and wealth. However, the code alone has proven ineffective as a mechanism to drive development. There is a consistent body of knowledge emphasizing the role of analysis and design, to derive the code. Recent trends in software engineering, such as Agile Methodologies also emphasize the role of requirement capture and tests and the interaction with the business context.

In any case, software development requires an integration of different activities, each using a different formalism. Traditional practices extract requirements, perform analysis, write the code, etc. Agile Methodologies put the accent on a strict integration of code with semiformal requirements documents – user stories, metaphors, etc., and with systematic tests in close collaboration with customers and users.

Tool support appears an invaluable form of aid to software production, improving quality and productivity by highlighting and, when possible, correcting errors, maintaining consistency across different activities, automating tasks, modeling financial outcomes and opportunities.

Almost all people involved in the development process are competent in using their own formalism: managers in planning using Gantt etc., analysts using formal models, coders in writing the code. The same reasoning can be generalized to most of the kinds of profiles involved in the overall production.

There are effective tools addressing the specific, consolidated activities related to development, say, project management tools, analysis tools, design tools, etc. Such availability refers both to proprietary tools and to open source tools. However, only limited support is available for integrating different activities and new practices and methodologies. For instance, there is minimal tool support for Agile Methodologies, just a bunch of nice but small tools such as xUnit, Cruise Control, etc., and the integration of such tools into proprietary system has required some effort.

The paper is organized as follows: section 2 summarizes the state of the art of tools integration; section 3 describes a set of ideas for an improvement in tools for software development; finally, section presents future scenarios.

2. STATE OF THE ART
The integration of simple tools is a well-known practice in the Unix community, including in the Linux one [1] [11]. There is a huge set of very specialized and efficient programs, each of them

addressing a single need [4]. The glue to connect these simple programs is a shell script that uses the pipe mechanism to redirect the input and output streams.

A different way of software integration is based on the development of a graphical interface that can simplify the use of a set of command line utilities as it happens to burn CDs in the Linux environment with the K3b tool [5].

A more complex integration is performed in tools such as Emacs. This tool is highly customizable through the Lisp language so that it is able to transform the text editor into every text-based application such as an e-mail client, a development tool, etc.

Moreover, the research community has investigated the problem over many years. One example is the PCTE (Portable Common Tool Environment) project supported by the European Union through ESPRIT in the '80s [7]. The aim of the project was the definition of an interface specification for tools in order to make them communicate but this framework has not been commercially successful. Some reasons could be found in the complexity of the approach and in the lack of standard tools and languages to manage data exchange, integration and workflows.

A way to develop new applications through integration is the package oriented programming (POP). This technique exploits mass-market applications, like Microsoft Office, as large components integrating their functionality to provide a user familiar new products with low costs. This technique is usually platform dependant because it is not possible to integrate components running on different platforms [12].

This is an important limit because each one provides different and specific applications that could be used as high quality components.

3. TOOLS INTEGRATION

Web service technology is completely based on the XML language as a lingua franca to make services communicate. To achieve this goal, a transformation language (XSLT) is available and a language specific for integration [2] has been recently developed. A protocol to integrate programs could be based on this set of already available protocols and adapt them to programs integration [3]. This choice is based on the growing amount of tools and software libraries already available with an Open Source licence. Moreover, programmers are becoming familiar with these technologies and almost all tools already support such data format.

A data interchange format is not enough but it should be a starting point to develop an integration protocol without strong requirements for basic operation because such tools are already available, even if they have been developed to achieve a quite different goal.

In the past there have been several open source projects that have been a success just because of the open source/open standard approach.

Emacs is an example of a tool that owns part of its success to its integrability and its expansibility to new environments and tools [11]. Emacs was first developed in 1975 and therefore is still used after more than 27 years of its inception. Almost every command in Emacs is a small LISP program that acts upon the document. To extend Emacs, existing programs are simply modified or new ones added which made it possible that the evolution of the application was fluid and very user-driven [10].

Emacs is not the only development system that allows integrating plug-ins, commercial software like Visual Studio from Microsoft allow that, too [8]. Also open source platforms that have come from individual corporations, such as the Eclipse tool integration platform, support plug-ins.

Unfortunately, Emacs is now not any longer adequate for today's needs. It is anyway important to learn from such 25+ years success story and to identify its key features with respect of other corporate or open source solutions.

Emacs is centered on **(a)** an open protocol and on **(b)** a protocol upward and downward compatible used by all plug-ins developed using elisp. An open, upgradeable, fully upward and downward compatible protocol like the one of Emacs is the key technology to bring to success several open source projects.

Development platforms like Eclipse or Visual Studio are both extendable; they allow the development of integrated plug-ins, but they are tailored to their specific architecture and are not upward-compatible in the sense that sometimes new versions of the underlying products require new versions of their plug-ins.

An example is the embedded programming environment Visual Basic for Applications that is part of Microsoft Office. It changed substantially over the last versions of Microsoft Office. With Microsoft Office 2003, .NET will be introduced as the next generation programming environment, making much of the current development effort useless [9]. Moreover, the way different events are caught within Microsoft Applications is not totally homogeneous: for instance, on-activate and on-deactivate is handled differently in Microsoft Word XP and Microsoft Excel XP.

An open, multi-purpose, upward- and downward-compatible protocol can address effectively such problem. Such protocol would leverage the effect of the several tools and in this way increasing substantially the productivity of the development activity. In essence, an open, upgradeable protocol supports "platform transparency" and facilitates cross-platform development, which may help significantly the industry as a whole.

The analysis can be extended from protocols to tools. It is interesting to analyze the economic reasons behind the instability of corporate solutions. The open source community has no interest in changing underlying protocols because this would create problems for many other developers and users. On the other side, commercial software producers are interested in selling new versions of their system and may have an advantage in making the new system incompatible with previous versions. If a change is unavoidable, e.g., to correct bugs, an open source standard makes it easier to adapt a tool or plug-in that stopped working after the new release because the new and the old protocol are known.

In addition, a key characteristic of open source development is that it is focused on diffusion; therefore, it is in its best interest to keep full upward and downward compatibility in its subsequent releases. This is not the case for commercial products where short time-to-market development cycles have priority, and there are incentives to get people to jump on new versions of tools as soon as possible, if possible disregarding old versions of tools. An

example for this is again Office 2003 that requires at least Windows 2000 and will not work on Windows 9x [6]. After all, versioning is the only way to keep a cash flow coming, in a world where the information good does not "naturally" age.

4. OPEN ISSUES

Tools integration is a basic requirement to improve the development process in order to reduce the overhead required by data synchronization and information exchange among incompatible or partially compatible tools.

At present, there are no established architectures to address this problem due to the complexity, corporate strategies, and failed attempts.

Open source protocols and technologies can be used to propose a widely adopted standard to perform such integration and provide both open source and commercial communities a common way to develop inter-operating systems.

5. REFERENCES

[1] Bach, M. Design of the Unix Operating System, Prentice Hall PTR, 1986.

[2] Business Process Execution Language for Web Services. http://www-106.ibm.com/developerworks/library/ws-bpel/

[3] DeRemer, F., H.K. Kron. Programming-in-the-Large Versus Programming-in-the-Small. IEEE Transactions on Software Engineering, 2(2): 80-86, 1976.

[4] Frakes, W.B., B.A. Nejmeh, C. Fox. Software Engineering in the Unix/C Environment, Prentice Hall, 1991.

[5] K3b homepage: http://k3b.sourceforge.net/

[6] Keizer, G. Drops Support For Older Windows, 2002. http://www.informationweek.com/story/IWK20021031S0008

[7] Long, F., R. C. Seacord. A Comparison of Component Integration Between JavaBeans and PCTE. International Workshop on Component-Based Software Engineering, 2003. http://www.sei.cmu.edu/cbs/icse98/papers/seacord.html

[8] Microsoft Visual Studio – Creating Add-ins and Wizards, 2003. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxconcreatingautomationobjects.asp

[9] Microsoft Office 2003 for Developers, 2003. http://www.microsoft.com/office/preview/developer/default.asp

[10] Raymond, E.S. The Cathedral and the Bazaar, 2000. http://www.catb.org/~esr/writings/

[11] Raymond, E.S. The Art of Unix Programming, 2003. http://www.catb.org/~esr/writings/

[12] Succi, G., W. Pedrycz, E. Liu, J. Yip. Package-Oriented oftware Engineering: A Generic Architecture. IEEE IT Professional, 3(2), 2001.

Supporting Distributed and Decentralized Projects: Drawing Lessons from the Open Source Community

Justin R. Erenkrantz, Richard N. Taylor

Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
{jerenkra,taylor}@ics.uci.edu

ABSTRACT

Open source projects are typically organized in a distributed and decentralized manner. These factors strongly determine the processes followed and constrain the types of tools that can be utilized. This paper explores how distribution and decentralization have affected processes and tools in existing open source projects with the goals of summarizing the lessons learned and identifying opportunities for improving both. Issues considered include decision-making, accountability, communication, awareness, rationale, managing source code, testing, and release management.

Categories and Subject Descriptors

K.6.3 [**Management of Computing and Information Systems**]: Software Management - *software development*

General Terms

Management, Design

Keywords

Decentralization, Distribution, Open Source

1. INTRODUCTION

Organizational distribution and decentralization alter critical factors in the software development process. Historically, centralized organizational structures have prevailed. A single organization, or part of an organization, has been fully responsible for a project, bearing the ultimate responsibility for shaping the deliverables and selecting the tools and processes used in development. Communication and consensus building within the organization has been facilitated by physical proximity.

For numerous business reasons, over the past decade and more, many organizations have moved to *distributed* development — a single administrative authority operating over physically distributed subgroups. This change has been supported by improvements in communication and networking technologies. Nonetheless, with participants no longer physically collocated, the processes and tools of development have had to change to attempt to cope with the difficulties so incurred.

Development of applications by *decentralized* organizations adds an additional wrinkle into the problem. By decentralized development we mean that no single organization controls the project; rather that all decisions related to the goals and objectives for the project must be made multilaterally. The motivation for decentralized development is akin to the motivation for participation in standards bodies: the common weal can be advanced while independence is retained. Each organization holds ultimate authority for its internal processes and tools. To the extent that interaction between participating organizations occurs, selection of the involved tools and processes must also be done multilaterally.

The premise of this paper is that we can gain some insight into how to effectively meet the challenges that face decentralized and distributed development organizations by examining the practices of the open source community, as these projects are most often both distributed and decentralized. The intended beneficiaries of this work is, largely, new open source projects, through several of the observations have applicability outside the open source domain.

Some work along this line has already taken place. Progressive open source[6], for instance, has been introduced in some commercial entities. This model is primarily geared towards applying open source practices within the community of internal employees or specific strategic partners rather than the public. This model is not fully decentralized, however. Implicit in the notion of progressive open source is a controlling authority that can dictate development.

There has also been a large body of work related to distributed software development. One particular area that has been carefully studied by Herbsleb, *et.al*. is the communication between participants in a distributed software project[15]. In this case study, participants were spread across several countries and developed a project collaboratively. One of the significant results was that there was a clear bias towards communicating with people in proximity, rather than communicating with remote peers, even when supported by good communication technology. This study does not fully explore the effects of decentralization on development, however, as all of the participants essentially worked for the same organization and were clustered in relatively large groups at a small number of sites. In a highly decentralized and distributed software project, few develop-

ers may be in proximity and belong to the same organization.

Another body of work has focused on enhancing technologies specifically for supporting distributed development. CSCW technologies fit into this category, as well as enhancements to the web. In [7], for instance, enhancements were discussed that could make the current web tools better facilitate collaboration. However, it limited itself to web-based artifacts and does not lay out a guideline for the processes best suited to these tools.

It almost goes without saying that not all projects require heavyweight processes and tools due to their limited scope or participation. Introducing unnecessary processes and tools may stifle a small project. It is also possible that participants do not desire expansion beyond a specific threshold. These classes of projects do not truly fit the distributed and decentralized criteria. In the following discussion, therefore, we will only concern ourselves with projects that are sizeable or complex enough to warrant tool and process support and which are developed in a collaborative, distributed, and decentralized fashion.

We begin the remainder of the paper with discussion of a survey of open source projects, showing similarities that have arisen in tool usage. Discussion then turns to characterizing the ways distribution and decentralization can constrain processes and tools. We then begin to summarize lessons from the open source experience, starting with identifying the management and coordination needs. We continue with an examination of techniques for satisfying these various needs. We conclude with a discussion of potential future work.

2. BASIS PROJECTS

Since most open source projects display significant degrees of distribution and decentralization in their organization, they provide a good foundation for study. Some prior examinations have been conducted into the tool usage of open source projects[12]. While most open source projects are not directly related to each other in terms of the subject of their production, a commonality of supporting tools has emerged in many cases.

In [12], eleven open source projects were surveyed to determine what tools are used to support the development model of the project. The survey was conducted to determine the quality aspects of open source projects and determine how to improve the project deliverables. The surveyed projects are spread across several different domains - including compilers, web servers, programming languages, and desktop environments.

The surveyed projects are among the most successful open source projects available. The Apache HTTP Server is currently in use by about sixty percent of all websites[20]. Servers shipping with the Linux kernel amounted for fourteen percent of all servers shipped in the first quarter of 2003[11]. Tomcat is the official reference implementation for the Java Servlet and JavaServer Pages technologies[2]. Therefore, these projects provide a reasonable basis for examining how successful distributed and decentralized open source projects should be conducted.

As Figure 1 depicts, these projects also represent a wide range of

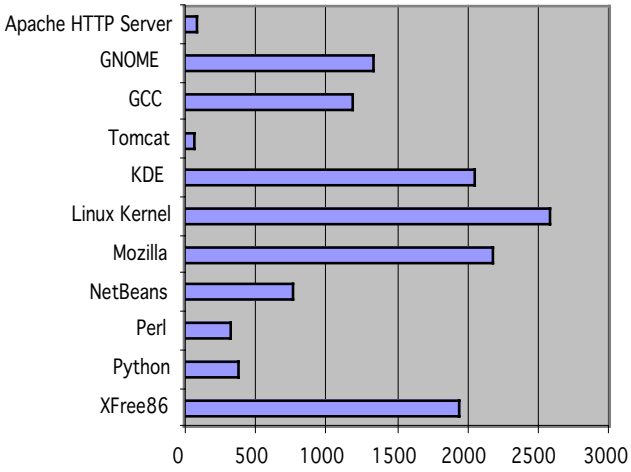

Figure 1. Lines of Code in Basis Open Source Projects (1000s)

source code size. One project had as little as 55 thousand lines of code (Tomcat), while another surveyed project supports 2.5 million lines of code (Linux).

Each project has independently chosen the tools and processes that best fit it. Most of the surveyed projects do not have any common developers, so there is no direct relationship. However, in some areas, a consensus appears to have been reached concerning the proper tools to use.

In the survey, all of the projects shared the same source control system, CVS. However, since the publication of the survey, Linux has adopted BitKeeper as its source control system[4]. While there does currently appear to be a consensus regarding CVS, a number of other replacements to CVS are actively being developed. These include such tools as Arch[1] and Subversion[5]. Therefore, this consensus may not be stable over the long-term as newer products attempt to replace CVS.

In other areas, there is no single tool that predominates; rather, a small number of tools are commonly used. One such area is in mailing list software; two tools currently dominate - ezmlm[3] and Mailman[10]. While two of the surveyed projects used other tools, the rest used one of these two tools.

In yet other areas, such as web portals, there is extreme variations in the tools used. No two projects shared the same tool for updating their website. At this point, most projects are creating customized tools for their website that fit their individual needs rather than relying upon a pre-built solution for content management.

The variations of tool similarity across problem domains presents an interesting statement. In some areas, open source projects have found a particular tool that seemingly fits their development model well. This is evidenced by consensus concerning a particular tool. This consensus may be due to an inherent property of the way the project is organized whereby this tool is the only obvious choice. Or, perhaps the adoption of a particular tool is a matter of historical accident. If the adoption is related to historical accident rather than

a solid fit, the introduction of tools that are better suited to a distributed and decentralized development model may be able to replace the current consensus. However, in order to encourage better tools and processes, we must understand the constraints placed upon an open source project by decentralization and distribution.

3. CONSTRAINTS

The presence of decentralization and distribution in a software project places a number of new constraints on what processes and tools can be effectively utilized. In order to obtain a clearer picture of what may work in these environments, we need to be able to identify these constraints.

3.1 Decentralization

The decentralization aspect of development requires processes to consider multiple interested parties. The involved developers may act towards their own goals, rather than the goals of the entire project. Therefore, not all developers will necessarily be aligned on all items and tasks. Yet, the processes and tools used should try to promote working towards a common beneficial goal while meeting the individual goals.

Due to decentralization, developers may not all work for the same physical organization. However, one organization may fund a portion of the developers on a project. If this organization removes its funding, their associated developers may leave the project. Therefore, the project needs to be able to withstand such losses or risk having the project abandoned. This risk promotes processes and tools which maintain continuity and shared communal knowledge.

When the individual goals of organizations collide, care should be taken in resolving these concerns. If these concerns are not met to everyone's satisfaction, dissatisfied organizations may leave the project. Depending upon the influence of the departing subset, it may place the project in jeopardy. Therefore, processes promoting compromises should be strongly emphasized to minimize such departures.

3.2 Distribution

Distributed software development places a strain on the project's communication mechanisms. When developers are not collocated, it is no longer possible to have frequent face-to-face meetings. Therefore, other communication mechanisms and tools must be deployed to fill this void.

As noted earlier, prior studies into the nature of distributed software development have indicated that it is hard to facilitate communication to the right person at the right time across site boundaries[15]. In order to address this problem, processes and tools need to be in place to allow timely identification of key contacts. Herbsleb, *et.al*., for instance, identifies needs in the areas of awareness, rich interpersonal interaction, and support for finding experts.

Since developers are not physically collocated, it may cause problems with synchronous communication as developers may be scattered across timezones. If synchronous methods are used, some participants may not be able to contribute to a discussion. Therefore, asynchronous communication mechanisms are usually preferred.

4. MANAGEMENT AND COORDINATION NEEDS

This section identifies management and coordination needs that decentralized and distributed project organization imposes. If these needs are not properly addressed at the outset, then repercussions may arise as the project matures.

4.1 Goals

Before embarking on a project, there is usually a need for a clear statement of goals that the project needs to accomplish in order to be successful. Upfront identification of goals allows for examination by prospective participants. It may be that the initial goals may not suit all interested parties. Therefore, the goals may need to be altered to support other interests. If the interests are made to correspond, the groups can begin to proceed to coordinate development tasks. If their interests are irreconcilable, the parties may proceed separately. A confrontation may occur later if an implicit differences in goals is later revealed.

Furthermore, if only a few parties wish to participate, the potential cost of decentralization may not add sufficient value to the project. It may be that the project does not have enough outside attraction to reach a critical mass to support a viable community. Unnecessarily adding the overhead of decentralization may end up harming the viability of the project.

4.2 Coordination of Initial Development

Once a goal has been determined, the interested parties need to identify how to reach these objectives. This roadmap can be valuable in planning development activities. A project may have an initial donation of code to build upon, or the new project needs to start the development process.

4.2.1 Inherited Code

A project may inherit code based upon a prior effort that has decided not to further development, or, one of the interested parties may be willing to donate code to begin the development effort. In either case, interested parties should be aware of the implications of the decision.

When using inherited code, the primary task becomes enhancement and evolution. At first, the project may be able to bypass the design stage of the software life-cycle. The majority of tools and processes will be geared towards maintenance. Depending upon completeness of the donation, design artifacts may need to be reproduced to promote understanding of the inherited code.

As the project matures, limitations may be found in the initial design that require substantial refactoring. The initial developers may desire a reasonable expectation that the inherited code allows for ample extensibility. Otherwise, efforts to evolve the code may encounter an early roadblock that forces reconsidering the usage of this code.

4.2.2 Initial Code

When a project begins afresh, the initial processes and tools will primarily be design-oriented rather than maintenance-oriented. In order to work in a distributed environment, the processes and tools must be able to support collaborative design. As the project evolves, the processes may alter to primarily supporting implementation and maintenance tasks.

A common occurrence in open source projects is that a publicly documented standard is implemented. These documents are typically written by a separate standards organization. These documents serve as the initial requirements and often specify interoperability characteristics of the deliverable. Once agreeing upon a standard to implement, developers can then devise a plan to carry out the architectural design and implementation.

By minimizing the requirements stage of the software life cycle by leveraging pre-existing standards, more effort can be directed towards the design and implementation. However, many projects implementing public standards also provide feedback to the standards committee based on real-world implementation experience.

4.2.3 Effect on Design and Requirements

Since some of the most prominent open source projects inherited code which implements a public or well-known standard, it may stand to reason that the processes involved with design and requirements gathering are not as well developed as maintenance and extensibility of code in open source projects.

However, in these particular cases, the requirements and initial design have already been determined in a very rigorous manner. In the case of projects which implement Internet RFCs, these requirements have been created in a very decentralized fashion. However, once these requirements have been established, various parties will form groups to implement the standard.

Therefore, while it may seem that some open source projects lack an emphasis on requirements and design, we may be able to rationalize that on the strict division of requirements and implementation in the traditional standard-making bodies of the Internet.

4.3 Common procedures

In decentralized communities, the interested parties may establish a common set of rules for running the project. These rules will take the place of a controlling authority which dictates such rules. Furthermore, this will allow all parties to operate within stated organizational parameters.

The parties should have already agreed on the goals and may have agreed on the initial design, but they must now also agree how to reach the desired result in a formal manner. If a conflict between members of the project arises, there needs to be a predetermined mechanism for resolving these conflicts.

If these steps are ignored and such a process does not exist, it may introduce tension between parties. By creating and following these guidelines, the belief is that most conflicts will be resolved peacefully. These procedures should also attempt to not introduce unnecessary overhead in the development process.

If these mechanisms fail and an impasse develops, then the community may be *forked*. One of the most prominent examples of the forks of open source projects are among the BSD-derivatives[17]. Despite having a common ancestry, the vision of each BSD-derivative is slightly different. In essence, each BSD-based platform has the goal of creating a Unix-like operating system. However, each of these derivatives has a different technical or procedural vision of how this goal should be accomplished.

Therefore, we suggest that decentralized projects are self-correcting though at a cost of wasted resources. When a difference of vision occurs between developers and organizations, projects can be forked to maintain the integrity of the private goals. In the end, each constituency retains their private goals, and may be willing to separate from other participants if an impasse is reached. Only when their private goals are being met will participation continue.

4.4 Tool requirements

When multiple parties participate in decentralized development, special attention should be made to the requirements of the tools that support the processes. The selection of tools should recognize that not all developers have equal resources to acquire specialized tools. Open source projects may not be directly funded, but when all participants are funded, these requirements may not be as stringent.

Since open source projects are traditionally open to all developers regardless of organizational affiliation, the tools used are commonly open source as well. By relying on free tools, this alleviates financial barriers to participation as not all developers may receive direct compensation for their work on a project. It may be unreasonable to expect developers to purchase tools in order to work on a project.

Due to the variety of developer preferences, most required tools need cross-platform support. In the open source community, a good tool will not require developers to switch their operating system to use a special tool. This allows developers to work on platforms with which they are most comfortable.

Since a project may attract developers of different skillsets, it may be unreasonable to expect developers to have special training in a tool or a technique. To offset this, projects may need to provide clear documentation on techniques that will help unfamiliar developers. Furthermore, since the participants are self-selecting, not all participants may have formal computer-science backgrounds, so some more advanced techniques may not be accessible to all participants.

5. PROCESS AND MANAGEMENT TOOLS AND TECHNIQUES

This section describes process and management techniques that may be used in distributed and decentralized projects. These techniques attempt to satisfy the needs discussed previously. We will examine how open source projects are solving these constraints and identify potential areas of improvements for each concern. Table 1, at the end of the paper, will summarize these techniques.

5.1 Delegation and Decision-Making

A concern for distributed and decentralized projects is delegation of assignments and leadership. Since the participants do not necessarily share the same reporting structure, traditional management techniques may not apply.

Similarly to other management models, there may either be a flat or hierarchical structure within this decentralized organization. Ultimate authority may rest with a specific individual, or decision-making responsibility may be shared by the interested participants. In the case of a single authority, this individual may set policies unilaterally. However, these policies must still promote participation by others. This requires the creation of a benevolent dictatorship where participants are willing to yield authority to a central authority — explicitly backing away from decentralization.

A prominent example of this central authority organizational model is seen in the Linux kernel. Linus Torvalds was the initial designer and developer of the Linux kernel. The rest of the participants have allowed him to maintain control over the project. Linus has the ultimate say on what changes make it into the kernel.

Designating a single individual with ultimate authority may create an organizational bottleneck. Therefore, a hierarchical organizational structure usually accompanies these structures. In Linux, most substantial components of the kernel have an associated maintainer. Rather than submitting a change directly to Linus, changes should be submitted to the responsible maintainer. If the maintainer agrees with the patches, the patches can then be submitted to Linus.

Linus places a certain degree of trust in his maintainers that they will follow his process for submitting patches to him and deal with most of the overhead for that component Yet, due to the supreme nature of Linus's role, he can overrule the maintainer of a component. It is possible to circumvent a maintainer and send a patch directly to Linus. If he decides to apply the changes without receiving prior input from the maintainer, he retains that right.

Another model commonly used by open source projects, one more in tune with decentralization, is the meritocracy model. This is exemplified by the Apache HTTP Server Project[9]. All members share equal power, so there is no direct leader of the project. Under this flat organizational model, people gain power by sustained contributions over time. The power of the developer is enabled by grants of write access to the shared repository and the ability to veto changes.

Until a developer gains commit access, they are considered a contributor. While they may participate freely on the mailing lists, an intermediary with appropriate access must review and commit their suggested changes. They may also cast non-binding votes on issues before the community. Since these votes are non-binding, developers with binding votes may choose to disregard such votes.

As the voting developers are exposed to a new participant, they are examining the quality of the contributions and how the participant works within the community. Then, one voting developer will nominate the contributor for voting privileges to the rest of the voting developers on a private discussion list. If the group considers the contributions beneficial and the participant trustworthy, voting privileges will be offered.

While no single person can control the project, each voting developer has *veto* authority to stop undesired changes from being merged into the shared repository. While these vetoes can be cast on any patch, there must be a valid technical reason for stopping this change. There is also no way to override a veto - this organizational model enforces consensus-building.

5.2 Accountability

Accountability may become an issue in a decentralized organization. If there is a problem with the software, users may desire a contact to resolve this problem. Open source projects have typically addressed this concern in two fashions: creating for-profit corporations that provide commercial support or creating non-profit foundations that provide a perpetual point of contact. These solve the issue by a direct step away from decentralization.

Some organizations that participate in open source projects provide for-fee support as a source of revenue. For example, this is common in the relational database domain. Two open source databases, PostgreSQL and MySQL, both have strongly related corporations that sell support to end-users.

These commercial entities will often provide support plans that assist users in setting up the product. These companies may also respond to direct support questions concerning the product. By having a revenue stream, these companies are able to fund development of the associated open source project by directly financing developers. These developers may add enhancements that the organization's client base has requested, or fix problems that have been identified by support personnel.

As an alternative to providing a commercial support, some open source projects have established non-profit foundations. These foundations are the owner of the code and do not have any explicit commercial interests. Therefore, it is expected that these foundations will be able to oversee and maintain accountability for the code. Two prominent examples of this are the Free Software Foundation and FreeBSD Foundation.

In these cases, a non-profit foundation is usually responsible for providing the infrastructure of the project. They will typically provide the services that allow development to occur. These foundations do not usually provide end-user support or directly fund developers. However, the foundation is expected to be eternal, while a for-profit corporation may be forced to dissolve due to financial considerations.

5.3 Communication

Due to the introduction of distribution, there may be varying degrees of developer collocation. Since the projects are also typically decentralized, developers may not work for the same physical organization. Therefore, the development process must allow for communication between people not at the same location and not belonging to the same physical organization. Therefore, the major-

ity of communication should be at the virtual organization level, rather than the physical organization. By relying upon asynchronous forms of communication rather than synchronous communications, a higher proportion of global developers can be supported. Yet, relying upon asynchronous communication introduces a delay factor[8,15].

In order to facilitate communication to the right person at the right time, mailing lists are commonly used. This reduces the number of contacts that are required. Almost every open source projects uses public mailing lists to promote subscription by non-developers and to encourage contributions by new developers.

Multiple mailing lists may also be used to further segment the mail traffic. These mailing lists may be dedicated to a particular subtopic. By reducing the scope of a mailing list, it allows for separate communities to form within the same project. This may be beneficial for encouraging growth in large projects. It also moves discussion away from a more generic mailing list where there may not be as many interested people in the discussion.

It has been stated that email is predominately used because it is the least common denominator[8]. One problem with email is that it requires a common language to be used. Mechanical translation services have not yet proved to be sufficient to address technical translations. This may promote developers who are only fluent in the main language of the developers.

A possible avenue to research would be to investigate projects where developers do not share a common language. In these cases, it would be useful to analyze how developers communicate when they do not share a language. This may promote islands of developers that do not often communicate.

In addition to relying upon asynchronous communication, some projects use a variety of synchronous communication (such as real-time chats or instant messaging). Yet, this is only effective when developers are located in similar time zones. If not all developers can participate in synchronous communications, it is essential that some archival of the communications be made. Otherwise, key participants may be left out of a critical discussion.

5.4 Awareness

Awareness is an understanding and coordination of what participants are doing. Since the personnel of a decentralized and distributed community may be rapidly changing, it may be hard to even identify who is currently active. This aspect of development processes has been remarkably underdeveloped. Most coordination efforts remain ad-hoc and short-term.

However, mailing lists provide a rudimentary tool for coordination. A developer can post on the mailing list indicating that they are planning to perform some activity. But, there is no enforcement of this plan. This leads to a problem when a participant says they are going to accomplish some task, but does not complete it.

Some projects may also use shared information repositories for awareness information. For example, the Apache HTTP Server Project relies on a STATUS text file that lists outstanding issues; this file is emailed weekly to the main developers mailing list. Participants may make a notation as to which issues they are addressing. However, it may require frequent refreshing of this file to ensure that the information is not stale.

Many open source projects also require that large changes be discussed before implementation starts. This allows other developers to provide feedback on proposed implementation strategy. Leveraging the feedback of developers may allow potential design problems to be detected earlier than if review occurs after implementation.

5.5 Historical Rationale

Since turnover may be high in decentralized projects, a collective history should be maintained and documented. By examining past communications and activities, new developers can begin to understand decisions made at a certain point in the past. It also allows developers to learn from prior decisions.

It is essential to use communication mechanisms that allow for long-term archival. Most asynchronous forms of communication lend themselves well to archiving - such as public mailing list archives. Therefore, the delay factor introduced by asynchronous communication has an advantage of allowing capture of historical rationale.

However, spontaneous synchronous communications are often not archived. This may often be seen in projects where a number of developers are physically co-located. In these environments, face-to-face communications may have an unusually strong bias[15]. In addition to not allowing full participation of the group, these sorts of communication may be detrimental to distributed projects because artifacts of these conversations are rarely recorded.

A common problem in open source projects is that new developers often repeat or bring up old discussions. This demonstrates a lack of tools that encourage review of past discussions. If tools for reviewing prior discussions were readily available, developer time spent rehashing prior topics could be minimized.

To help with this, Perl has created Perl Design Documents (PDDs) which lay out the rationale for certain decisions made in the development of Perl 6 and Parrot[28], This allows new developers to annotate and reexamine prior decisions in a central location. It may happen that a new developer has added insight that was not noted in the prior conversation. If post-mortem annotation of discussions is allowed, it may achieve a balance between stifling and encouraging reexaminations.

5.6 Design Rationale

In addition to allowing for discovery of important historical conversations, it may be critical to understand the design rationale of certain components. In a distributed and decentralized environment, it may not be possible to contact the original author of a section of code. Therefore, mechanisms need to be in place to communicate rationale to future participants.

One way to communicate rationale is by creating developer docu-

mentation. Some open source projects, such as AbiWord[25], keep interface definitions and notes in-line with the source code. Documentation can then be published with tools such as doxygen[14]. By synchronizing the location, when major changes are made to the source code, the belief is that the documentation will be updated. This makes it easier to produce developer documentation which reflects the current code.

Depending upon whether the project did the initial design, other artifacts such as design documents and diagrams may be available. In projects that provide an extensible interface, it is also common to produce well-explained and concise examples as a way to illustrate the interface in action. This helps new developers of an interface understand the code by looking at examples.

There has been research into encouraging software reuse, but these tools have not yet been integrated into the mainstream. There are tools available that provide relevant interface information in a personalized manner[30]. There has also been work towards harvesting the structural and semantic information of code[19]. Encouraging adoption of already existing tools may make capturing design rationale easier.

5.7 Participation

In projects where the personnel on a project may change frequently, it is important to have a published set of developer guidelines. These guidelines allow familiarization with the processes and tools used in a project. New participants can review them and contribute to the project in an intelligent manner.

Sites such as the KDE Developer's Corner provide a wealth of information that allow new participants interested in KDE to learn how to contribute[18]. The site contains introductory tutorials for developers new to the internals of KDE. Information about the development tools required to compile KDE and how to obtain the latest KDE snapshots are also available.

It has been mentioned that having an established set of guidelines shared by projects can reduce the redeployment costs of developers[6]. If all projects shared the essential guidelines, it would make it easier to contribute to new projects. If each project had its own set of unique guidelines, it would be difficult to transition to new projects. Therefore, it would be beneficial to encourage standardization of participation guidelines across projects.

5.8 Controlling Participation

A corollary to encouraging participation in decentralized and distributed software is that participation by new people must be managed by the current participants. Depending upon the access policies of the project, new participants may only have limited access to making changes to the project. Therefore, processes and tools need to be in place to support facilitating such contributions.

Tools such as the SourceForge's patch manager used by the Python project can be extremely useful[21]. These tools allows participants to submit patches to be applied, then developers with the appropriate commit access can integrate the changes. This particular tool also allows for annotations to be stored.

However, these current tools suffer from a lack of integration with the rest of the development process. Some projects enforce a policy where a certain number of positive reviews must be received before a change can be integrated[26]. Contributions may also grow stale as the project code base evolves. Furthermore, if none of the active developers deem an issue important, it may be a challenge to motivate integration. The tools used to control participation should ease the burden of merging the changes.

5.9 Managing Source

Since the developers are distributed, it is often a requirement to have a unified view of the source code. If a unified view is not available, it may be possible for developers to not be aware of the current state of affairs. Therefore, most projects will adopt some sort of collaborative software configuration management system (SCM). The processes and tools need to balance that each developer should be able to work independently while allowing them to remain consistent with the rest of the team.

As discussed previously in [12], the predominate SCM in use by open source projects is currently CVS. There has been a recent trend in seeking tools that can replace CVS[1, 4, 5]. CVS is based on a centralized repository model with one repository holding all of the content. Some of the newer SCM tools that have been introduced are keeping the centralized model of CVS[5], while others are attempting to decentralize the repository structure[1, 4].

However, depending upon the accountability structure of the project, it may make sense to keep a centralized repository even in a decentralized project. If a project has a non-profit organization which holds the copyright, then this organization should administer the master repository. However, if the project has a loose accountability structure, a decentralized repository structure may be more efficient.

Most SCM tools currently in use also promote an optimistic conflict resolution model rather than a pessimistic conflict resolution mode[16]. An optimistic locking strategy allows source conflicts to be resolved at commit-time, while a pessimistic locking strategy uses locking to prevent others from making changes while a change is being developed. A pessimistic locking strategy may interfere with parallel development as it prevents two developers from working on the same file at the same time. Only using pessimistic locking may have an impact upon the effectiveness of distribution.

5.10 Issue Tracking

One of the stated advantages of open source projects is that it is easier to fix problems since the source code is freely available[22]. However, it may still be difficult for non-developers to fix problems as they may not have the appropriate background required to resolve a defect. Therefore, processes and tools are required to report problems to the people who can help resolve defects.

Due to the presence of decentralization, it may be difficult to solicit participants who can resolve reported defects in a timely manner. Some participants may be wary of working with end-users, or are too busy to deal with acquiring the relevant informa-

Table 1. Summary and Avenues for Enhancements

Issue	Techniques	Project Exemplar	Avenues for Enhancements
Decision-Making	Project leader, meritocracy	Linux, Apache	Understanding consequences
Accountability	For-profit support, non-profit ownership	PostgreSQL, FreeBSD	Introducing clarity
Communication	Discussion lists, asynchronous	All	Balancing granularity
Awareness	Frequent status updates, Discussion before implementation	Apache	Creating better tools
Historical Rationale	Archival of communications, design documents	Perl	Creating better tools
Design Rationale	Developer-centric docs, examples	AbiWord	Enforcing synchronization
Participation	Clear tutorials, guidelines	KDE	Creating standards
Controlling Part.	Feedback, annotating contributions	Python	Integrating into other processes
Source Code	Public source repository, optimistic conflict resolution	All	Investigating decentralization
Issue Tracking	Soliciting developer assistance	Mozilla	Creating easy-to-use tools
Documentation	Distinct personnel, annotations	PHP	Separation of code and docs
Testing	Code reviews, automated tests	Subversion	Optimizing test executions
Release Management	Mirroring, versioning	Debian	Managing distributions

tion from the reporter. Therefore, the tools need to be able to support novice users and expert developers efficiently.

Standardizing on issue tracking tools, such as Mozilla's Bugzilla[27], may increase the familiarity of both users and developers with these tools. However, as these tools are adopted by more projects and enhancements requested, feature creep must be resisted. If the issue tracking tool becomes too complicated to use effectively, its usefulness is diminished.

5.11 Documentation

Since not all users of a project are developers that can understand code, a project must also be able to deliver quality user documentation. Otherwise, users may find the product too complicated to use properly. A significant challenge for distributed and decentralized projects is to have documentation at an equivalent quality to the code.

At best, documentation can be viewed as a form of source code. Therefore, many of the processes that apply to source code can also apply to documentation. Documentation may be written in a collaborative environment using similar tools and processes as the ones used to write code.

A problem in any software project is how to keep the end-user documentation synchronized with the current version of the source code. Oftentimes, developers are hesitant or reluctant to write user documentation. Therefore, when they make a change that is visible to a user, the developer may not update the relevant documentation. This leads to the documentation becoming out of sync with the code.

Some open source projects have addressed this by having separate documentation teams. One example of this separation is in PHP's documentation[29]. By isolating the documentation process from the development process, it enforces another perspective on the usability aspects of the code. This may result in an increase of quality of both the code and end-user documentation.

Another characteristic of the PHP documentation process is that it allows users to annotate the documentation on the website. As users spot errors in the documentation, they may append a correction comment to the website. Then, PHP documentation participants can harvest the changes into the main documentation.

5.12 Testing

There is often a strong desire to ensure that a project meets both the functional and reliability goals previously established. Therefore, testing processes and tools should be developed and encouraged throughout the life cycle of the project. There are two classes of methods that are typically used in open source projects: code review and testing.

Since it is difficult to conduct regular meetings in a distributed workplace, it is not possible to conduct periodic code review sessions. Therefore, code reviews must occur as the changes are conducted. Developers are usually asked to make small verifiable changes rather than large changes. By asking all developers on a project to review the changes as they happen and asking for the most concise changes possible, it may make it easier to identify problems sooner.

Besides relying upon manual inspection, some projects have a suite of automated tests for the project. These automated tools allow all participants to run the same set of tests at their discretion on their specific platform. One such project that utilizes automated tests is Subversion[5]. The test suite in Subversion is extensive and tests almost all functionality of the system. Furthermore, no releases can be made without first passing the automated tests. It may be possible to integrate some recent research into optimizing which regression tests are executed to improve the performance of the test suites[13].

5.13 Release Management

Since users ostensibly wish to use the deliverables of a project, quality releases must be produced. Therefore, a viable release strategy must be determined. If the project does not have a coherent process, it may have problems attracting users or achieving a reputation for stability.

In order to achieve widespread distribution, an infrastructure must be in place to allow public consumption. Some projects rely upon mirrored servers to handle the load of delivering releases to end-users. A critical concern is how to select these mirrors - should they be self-selected or should they be limited only to trusted individuals.

One such project that relies upon mirrors to deliver releases is Debian[23]. Debian balances the load across many geographically dispersed self-selected servers. However, Debian has several push-primary mirrors that are chosen because of their reliability. Self-selected mirrors can then pull releases from one of the pushed mirrors rather than accessing the master site directly.

Projects may also place meanings on the versions that deliverables are labeled with. This allows a shared understanding of the expected reliability. At times, it is helpful for a project to have a development branch that is not intended for widespread usage. These releases can also be used to perform dry-runs of the release process. This can be especially helpful when a project is trying a new release process. By explicitly labeling a version as unstable or development, it can help match the expectations of users with the expectation of the developers.

For example, Debian always has at least three versions that are actively maintained: stable, testing, and unstable[24]. The stable distribution is the one that is recommended for widespread usage. Then, the testing distribution consists of packages that are waiting to be included in the next stable release. Then, the unstable distribution is meant for developers and not meant for production quality.

6. SUMMARY AND FUTURE WORK

Adopting a decentralized and distributed organization for developing software requires rethinking fundamental process and tools. We have attempted to examine the consequences of supporting decentralization and distribution by seeing how open source projects have addressed these concerns. Table 1 provides a summary of the issues, techniques, and projects discussed in this paper. It also lists avenues for enhancement that have been identified where the current processes and tools could be improved to better support distributed and decentralized software projects.

If projects can create a clear line of accountability that is separate from all of the participants, it may foster a sense in the users that responsibility will be maintained. A decentralized project should be able to withstand the departure of organizations gracefully. If this is not present, then users may become wary of the project falling out of active maintainership.

By limiting the scope of discussion lists, it makes it easier for participants to understand what is currently going on in areas of the project. This level of granularity must be balanced with having too many mailing lists that makes it difficult to find the appropriate forum for discussion. However, when the right balance is achieved, this allows participants to easily partition discussion based upon agreed topical lines.

One concern for distributed software development is that a set of standards is required in order to ease participants shifting from one project to another. This may manifest itself as a common vocabulary shared between projects. If participants do not share a common language, it becomes hard to communicate effectively. The creation of standards and accepted best practices can help ease migration between projects.

A common problem in a distributed software project is understanding what other participants are currently working on. The creation of tools to promote awareness between developers can address this need. Furthermore, tools that promote capturing of historical rationale may make it easier for new participants to enter a project.

Another area for tool improvement is introducing a way to capture the rationale for a decision in the documentation. Currently, it is hard to identify why a particular change is made. The artifacts for determining this may not be centralized. Creating a tool that indicates relationships between artifacts to encourage rationale understanding may be critical.

The current tools for controlling participation are ad hoc and not well integrated. This makes it difficult to lower the burden upon the participants in a project in dealing with the contributions by casual participants. If the tools for handling contributions were better integrated into the standard processes, it would make this task significantly easier.

7. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0205724.

8. REFERENCES

[1] *Arch - Revision Control System*. <http://arch.fifthvision.net/>, HTML, 2003.

[2] Apache Software Foundation. *The Jakarta Site - Apache Tomcat*. <http://jakarta.apache.org/tomcat/>, HTML, 2003.

[3] Bernstein, D.J. *ezmlm*. <http://cr.yp.to/ezmlm.html>, HTML, 2000.

[4] Bitmover. *BitKeeper*. <http://www.bitkeeper.com/>, HTML, 2003.

[5] CollabNet. *Subversion*. <http://subversion.tigris.org/>, HTML, 2003.

[6] Dinkelacker, J., Garg, P.K., Miller, R., and Nelson, D. Progressive Open Source. In *Proceedings of the International Conference on Software Engineering (ICSE)*. p. 177-184, 2002.

[7] Fielding, R., Whitehead, E.J., Anderson, K., Oreizy, P., Bolcer, G.A., and Taylor, R.N. Web-based Development of Complex Information Products. *Communications of the ACM*. 41(8), p. 84-92, 1998.

[8] Fielding, R.T. and Kaiser, G. The Apache HTTP Server Project. *IEEE Internet Computing*. 1(4), p. 88-90, 1997.

[9] Fielding, R.T. Shared Leadership in the Apache Project. *Communications of the ACM*. 42(4), p. 42-43, 1999.

[10] Free Software Foundation. *Mailman*. <http://www.list.org/>, HTML, 2003.

[11] Fried, I. Sales Increase for U.S. Linux Servers. *CNet News.com*. February 10, 2003. <http://news.com.com/2100-1001-984010.html>.

[12] Halloran, T.J. and Scherlis, W.L. High Quality and Open Source Software Practices. In *Proceedings of the Meeting Challenges and Surviving Success: 2nd Workshop on Open Source Software Engineering*. May, 2002.

[13] Harrold, M.J., Jones, J.A., Li, T., Liang, D., Orso, A., Pennings, M., Sinha, S., Spoon, S.A., and Gujarathi, A. Regression Test Selection for Java Software. In *Proceedings of the ACM Conference on OO Programming, Systems, Languages, and Applications (OOPSLA 2001)*. p. 312-326, Tampa, Florida, October, 2001.

[14] Heesch, D.v. *Doxygen*. <http://www.doxygen.org/>, HTML, 2003.

[15] Herbsleb, J.D., Mockus, A., Finholt, T.A., and Grinter, R.E. An Empirical Study of Global Software Development: Distance and Speed. In *Proceedings of the International Conference on Software Engineering (ICSE)*. p. 81-90, 2001.

[16] Hoek, A.v.d. Configuration Management and Open Source Projects. In *Proceedings of the 3rd International Workshop on Software Engineering over the Internet*. Limerick, Ireland, June 6, 2000.

[17] Howard, J. The BSD Family Tree. *Daemon News*. April, 2001. <http://www.daemonnews.org/200104/bsd_family.html>.

[18] KDE e.V. *Developer's Corner*. <http://developer.kde.org/>, HTML, 2003.

[19] Maletic, J.I. and Marcus, A. Supporting Program Comprehension Using Semantic and Structural Information. In *Proceedings of the 23rd International Conference on Software Engineering*. p. 103-112, Toronto, Ontario, Canada, May, 2001.

[20] Netcraft. *Netcraft Web Server Survey*. <http://www.netcraft.com/survey/>, HTML, 2003.

[21] Python Software Foundation. *Patch Manager*. <http://sourceforge.net/tracker/?group_id=5470>, HTML, 2003.

[22] Raymond, E.S. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 2001.

[23] Software in the Public Interest. *Debian Mirrors*. <http://www.debian.org/mirror/>, HTML, 2003.

[24] Software in the Public Interest. *Debian Releases*. <http://www.debian.org/releases/>, HTML, 2003.

[25] SourceGear Corporation. *AbiWord Documentation*. <http://www.abisource.com/doxygen/>, HTML, 2003.

[26] The Apache HTTP Server Project. Apache HTTP Server Project Guidelines and Voting Rules. <http://httpd.apache.org/dev/guidelines.html>, HTML, 2003.

[27] The Mozilla Organization. *Bugzilla Project Homepage*. <http://www.bugzilla.org/>, HTML, 2003.

[28] The Perl Foundation. *Parrot and Perl6 PDDs*. <http://dev.perl.org/perl6/pdd/>, HTML, 2003.

[29] The PHP Group. *PHP: Documentation*. <http://www.php.net/docs.php>, HTML, 2003.

[30] Ye, Y. and Fischer, G. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *Proceedings of the 24th International Conference on Software Engineering*. p. 513-523, Orlando, Florida, May, 2002.

Towards a Method to Instantiate a Product Model for Open Source Software Development in a Commercial Environment

[Extended Abstract]

David Cruz, Sascha Vogel
Software & Systems Engineering
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
{cruz, vogels}@in.tum.de

ABSTRACT

Developing OSS in a commercial environment requires a process model, which combines the flexibility and creativity of OSS development with the structured process models of traditional software development.

A previous paper defines a metamodel which is suitable to build specific process models for developing OSS in a commercial environment. The following Paper gives another brief overview of the product model and exemplifies the construction of a process model. A characterization of project types is performed and specific adaptions for each project type are shown.

1. INTRODUCTION

The characteristics of OSS development [1, 7, 8] are very different from the ones of traditional SW development as used in many companies. Yet lots of companies have an interest in developing OSS. The interests may be of various natures with different focuses (e.g. better quality SW) (cf. [4]), in all cases the wish to develop OSS exists. Nevertheless many companies have difficulties in carrying out their intents, because the process of doing so is not simple nor visible. The product model defined in [2, 10] and [3] provides a means of defining a process to develop OSS in various environments.

This paper explains how to use the product model to create a process model and how to adapt the resulting process model according to special project characteristics.

At first the different kinds of possible OS-projects are subdivided into categories. The impact on common process attributes is shown and methods to adapt the process model in order to account for these impacts are proposed. This is part of section 2.

Then the product model itself and how it can be used in order to create a process model is shortly described in section 3.

Finally section 4 provides an example scenario and shows the use of explained techniques within that scenario.

2. PROJECT CATEGORIES

A large variety of factors have an impact on how to design the process of software development. Factors to be taken into account here could be software architecture, license model, business model, available tool support, etc. When constructing a process model, all of these factors have to be taken into account and the process model has to be adapted accordingly. For some of these factors the product model already provides advice on how to deal with them. In the following two important factors are explained and combined into types. The types influence different project characteristics, some of which will be mentioned after the explanation of the types. These characteristic and the need to cope with them will later influence how the process model is built.

2.1 Methodology

The first factor to be taken into account is the methodology by which to develop the SW. This might be a little confusing, but the thought that closed source SW must always be developed using traditional process models and OSS must always be developed in an open source environment, is not quite correct. It is perfectly possible to develop a SW completely within a company using the 'waterfall' model (cf. [12]) and then license the SW with the GPL, just as it is perfectly possible to develop a SW using OSS development techniques and not license it with an OS license.

In fig. 1 the two possibilities of what methodology to use to develop SW are shown. It is important to notice that open and closed source within the figure refers to the methodology to be used and not to the license chosen, though of course the license may have an influence on the choice, which methodology to use to develop the SW. Further explanation of

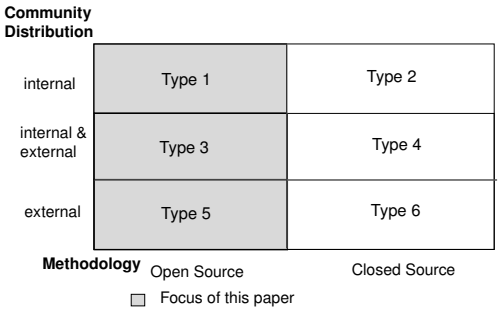

Figure 1: Project Classes

the figure follows in the section about community distribution.

2.2 Community distribution

One of the factors that has the most influence on the development process is how the community is distributed. When developing OSS in an industrial context there are three cases to take into account:

1. The company wants to develop OSS without the external community.

   This means that the company wants to develop OSS using an OSS development process without letting the OS community take part in the development process. As in these cases the development process is an OS process, the product model can be used to build a process model. The corresponding types within the product model are shown in fig. 1, type 1.

2. The company wants to develop OSS together with the community.

   Type 3 in fig. 1 corresponds to this scenario. This constellation also influences how the process model emerges from the product model. The explicit handling is explained in the tailoring mechanism in section 3.3.

3. The company lets the community develop an OSS.

   In this case (type 5 in fig. 1) the company interacts with the OS community without participating in the actual development process. This could be the case if the company where part of the organization. This case is very special in handling and hence the product model cannot be applied to it.

The development of closed source can also be divided into three cases, depending on the subdivision of participating developers:

- Type 2: The company develops SW without external participation.
- Type 4: The company develops SW in a distributed environment.
- Type 6: The company instructs external companies or developers to develop their SW.

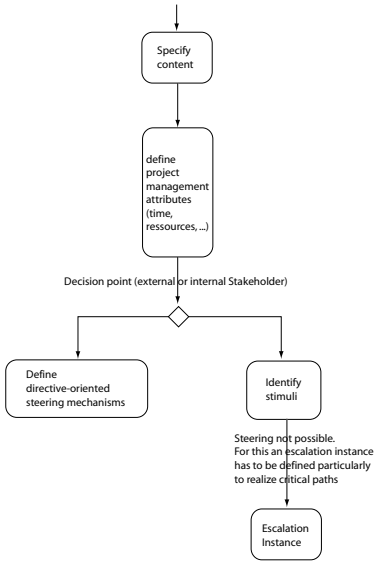

Figure 2: Planing Workflow

The last three cases can also not be solved using the presented product model.

2.3 Scope of this paper

The aforementioned categories allow to differentiate a lot of alternative project types. In this paper we focus on project types 1 and 2 of fig. 1, because that is what this paper understands as open source software development in a commercial environment.

In particular, the relation between OSS and traditional SW development is established by introducing planing, controlling and steering mechanisms in the OSS development process. It is easy to see that these management activities are necessary, but also different to introduce in a process model, as not all participating developers can be forced to work on their assigned tasks by directives.

For the mentioned project types, we introduce a tailoring mechanism, that defines a process based on activities, artefacts and project types. In this paper this definition of processes is called instantiation. The foundation for the following ideas is given by the product model described in [3].

2.4 Type Attributes

Depending on the aforementioned project types, different attributes of the process model should be stressed in different ways. In the following some of these attributes are shown and explained.

- Planing effort

  In most industrial processes someone wants to see a result at a fixed date, therefore planing is essential. Depending on the project type, planing can be adapted in order to meet the requirements imposed by an OS environment.

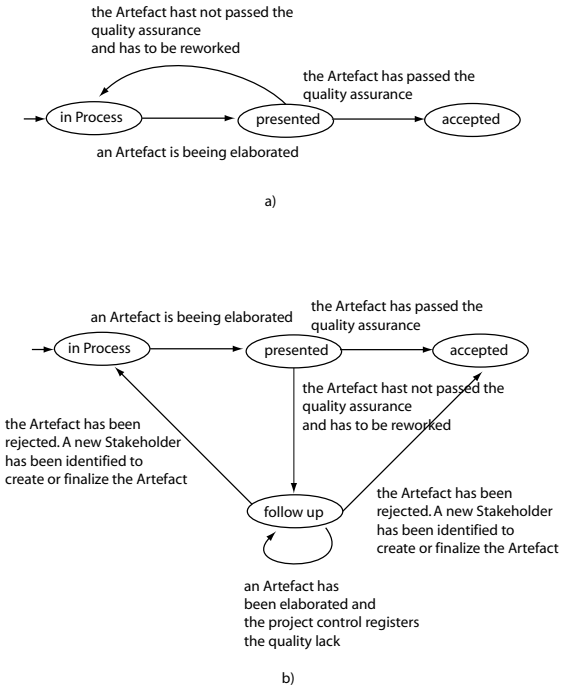

Figure 3: Artefact states

The realization of the planing activities depend on the stakeholders which are assigned to activities which have to be performed. In this paper we offer an OSS specific workflow which helps to plan projects even with external stakeholders. In general external stakeholders are not part of the organization itself. Therefore, traditional planing and steering mechanisms cannot be applied. For this we propose to define stimuli for stakeholders to motivate them.

The needed workflow is illustrated in fig. 2. It can be used for both of the identified project types. In project type 1 the workflow will always take the same path, whereas in project type 3 the stakeholder has to be determined. Before choosing a stakeholder additional information (e.g. time to process an activity) is inserted into the attributes which can later be used to control and steer the process as explained in the attributes motivation and steering.

If problems arise during the project steering, it is important to have "fall back solutions". By this we mean project instances such as further resources, alternatives, etc. to still achieve the given project goals.

- Quality characteristic

  The importance of quality assurance also changes with the project type. Quality assurance is not only concerned about the quality of the software, but about all artefacts built within the development process.

  The artefacts can be assigned to state machines, which store the quality situation the artefact is currently in. This machine has different appearances, which depend on the responsible role. If this role is an internal role, the machine consists of three more or less "traditional states" (cf. [5, 6]): in Process, presented and accepted as describe in figure 3a.

  If the role is an external one, the machine has to be adapted. As seen in Figure 3b it is expanded with the state: follow up. This state documents, that an artefact has not fulfilled the quality requirements.

  Depending on the project steering process it is possible to stay in the state follow up for further iterations of rework. Each iteration influences the standing of an external stakeholder within the project.

  Based on this state it is possible to reach two different states. Firstly, the artefact can be defined as in Progress and secondly, it can be marked as accepted. The Redefinition of in Progress illustrates that an artefact has been rejected by the project control. This implies, that the responsibility for this artefact has changed. Either another external stakeholder has been assigned to this artefact or an internal one. The more serious the finalizing of an artefact is, the more internal responsibilities should be introduced, because of to the possibilities of directive-oriented project management within one specific organization.

- Motivation and steering attribute

  The attributes motivation and steering only need to be accounted for, when the community is partly external, for then special measures for motivation have to be taken. When the community is entirely internal, there is no need to adapt the product model, as the ways to motivate an internal employee are not OS specific.

  As mentioned above the planing attribute inserts time information into the activities. This information is used in both motivation and steering.

  The standard procedure provided to motivate external developers is to offer incentives. When an activity is finished more than a fixed percentage early, the developer is publicly offered an incentive. There is no need to provide any measures if the developer hands in the work late, for any kind of penalization would most probably result in a negative reputation within the community.

  To be able to steer the project according to the plan, two artefacts that are only to be handled by internals can be added. The two artefacts are a positive and a negative list. Similar to when motivating an external developer, after each activity the resulting time score is stored in one of the lists. These lists can then be used to identify external developers which are reliable and assign them critical problems.

3. PROJECT SPECIFIC PRODUCT-MODEL

This section firstly introduces the product model given in [?]. Thereby, the model itself describes the type of a process model. This is described in the subsections "underlying product metamodel" and "products and views". Secondly, the instance of this process type is given. For this, we match the given example of a software development project to the product model. This matching mechanism is described in the subsection "tailoring". At the end, based on the scenario and project types, the product model is instantly applied to the

defined tailoring mechanism. In this way, a role assignment is performed.

3.1 Underlying Metamodel

The metamodel describes the elements of a work product and their relations. Generally, a work product consists of artefacts, activities and roles. Artefacts characterize physical elements which are produced during the development process. This includes documents or code fragments. The artefacts are created modified or removed by activities. In this context, specific activities are defined to realize one of these activity categories. For instance the artefact "code" can be modified as a consequence of a review comment or a faulty test run. Here, the activity means to correct and adapt the code according to the test specification. The roles determine either responsibilities or assistants of the artefacts.

Products can be related. For this, a relation-type exists which correlates artefacts and activities. This relation can be a consumer-producer- or producer-producer-relation. It depends on how the artefacts are used by the associated activities. An artefact which is on the one hand produced or modified throughout an activity and consumed on the other hand describes a "producer-consumer relation" between the products including the activities. If both activities produce or modify an artefact, we call the resulting relation a "producer-producer relation". In a similar way further relations can be identified.

3.2 Products and Views

One further aspect of the metamodel are views. They are used to structure the work products according to typical aspects of open source development. In the product model we identified the views: project initialization, software usage, code development cycle and release management. However, one work product can be part of different views as necessary. For instance, the work product "code depot" is used in different occurrences in different views. Whereas the repository is emphasized in the "project initialization", it focuses on the network structure of documents in the "software usage".

3.3 Tailoring Mechanism

As we have seen in the previous sections 2, the stakeholder significantly influences the project type attributes. Based on the introduced ideas a tailoring mechanism can be characterized. Here, tailoring means to adapt the product model according to the community distribution and the product relations.

Product relations arise as shown by activities creating or using artefacts. These relations exist not only within one product but can also exist between different products. Each kind of relation is important for the tailoring mechanism as described below.

However, to manage an OSS-project it is important to track these relations to avoid problems during the life cycle. For instance, assuming there are different stakeholders involved to modify and finalize an artefact for the quality assurance and one is waiting for the other.

In this case it is necessary, that the project management is able to strictly steer the process. Therefore, an appropriate

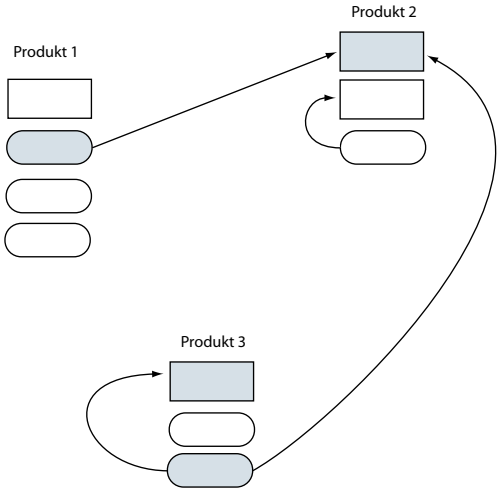

Figure 4: Tailoring based on the product criticality

planing or an adequate steering mechanism as mentioned in section 2.4 is important. If different responsibilities appear, the products should be harmonized in terms of assigning the same community class to associated product elements.

The aim is to minimize delay. The rule for doing so is: the more critical an artefact is the more manageability it should be and the more possibilities for steering activities should be present. In these cases, ideally, the role should be assigned to stakeholders within an organization.

4. CASE STUDY

This case study defines a scenario in which the tailoring mentioned above can be demonstrated.

4.1 Scenario

The scenario used to exemplify how to obtain a process model is as follows: A large company wants to develop the operating system for its mobile phone hardware using embedded linux. Different departments of the company will develop the software together and in interaction with the open source community. The open source community has been included, because the resources within the company are limited and not sufficient. The reason why this scenario was chosen is because of a number of interesting effects can be identified:

- different departments of the company building up embedded linux work together. For instance, the department developing hardware on the one hand and on the other hand the department for software and systems engineering as well as the integration unit.

  This kind of OSS cooperation is quite different from the hierarchical inter-department cooperation that is usual for big companies (cf. [9]). This has an interesting effect on the tradeoff of organizational rules and know-how interchange.

- The cooperation with the OSS community shows that internal and external stakeholders participate in the development process.

Both aspects show that traditional development processes are not applicable. People are motivated if they can be creative and have room for own decisions. A strict organized development process would reduce this motivation and thus efficiency in the development process. Otherwise planing and steering are indispensable in a commercial environment [12, 11]. A trade off between traditional and open source software development as introduced in this paper would help coping with this dilemma.

For the given scenario we can identify project type 3 to be suitable. The community includes internal and external stakeholders and the project method should contain open source elements.

4.2 Mapping Scenario and Product-Model

There are essentially two steps to take to tailor the product model in order to obtain a suitable process model. The type of the project has to be taken into account and then the actual tailoring can take place.

Once the project type has been determined (see section 2) the attributes to be stressed can be selected and used to go through the product model. In the case of the chosen scenario the community is internal and external. Furthermore the SW to be developed is an Application. Therefore the scenario can be categorized into a type 3 project.

As illustrated in section 2.4 a type 3 project needs to:

- define "fall-back-solutions" for all activities in case a stackeholder does not meet up the expectations imposed on him. If, for example, an external stakeholder does not accomplish the activity of filtering the task list properly, a "fall-back-solution" could be to simply have an internal person available to filter the tasks. A different solution would be to oblidge the developers, only to handle tasks essential for the next release.
- define a state machine for all artefacts depending on the stakeholder's community class. Every artefact starts in the state "in Progress". If an external person is responsible for the artefact "Risk list" and presents it to some other project member, the state of the artefact "Risk list" changes to "presented". If the artefact lacks quality, then the state changes to "follow up" and project control has to decide weather to define a new stakeholder for the artefact. If the artefact constantly lacks quality, then the project control will certainly define a new stakeholder.
- have special mechanisms to motivate and steer external developers. Lists of the fastest developers, the best documented code, etc. could be published on the newsboard to motivate developers to meet coding times or document better.

  If a developer codes the solution for a patch in twice the time that was planed to code the patch and this happens with most of his patches, then an internal negative list could sum up the time of delays. When a critical patch is to be developed this task can then be given to the developer with the most hours on the positive list.

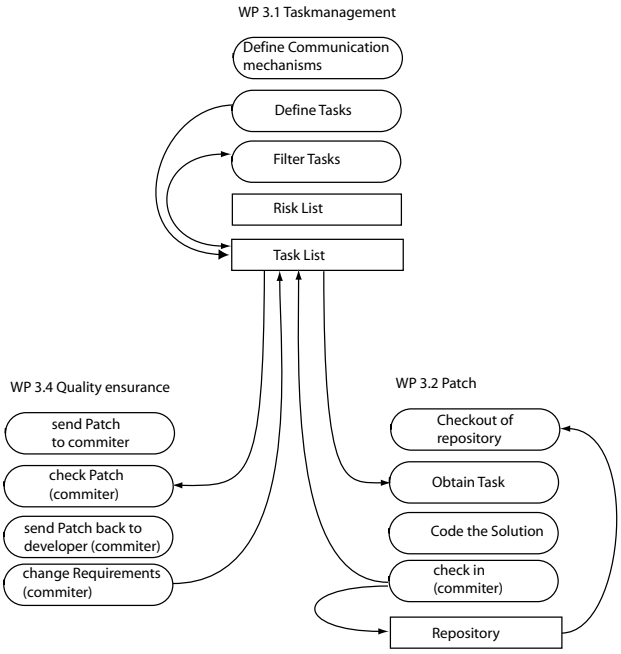

Figure 5: Tailoring the Scenario

Fig. 5 shows a small part of the "patch development cycle"-view (cf. section 3). There are three work products containing activities and artefacts which are partly related. The activity "Obtain Task" for example has a consuming relation towards the artefact "Task List" and the activity "Define Tasks" has a producing relation with the same artefact, whereas the activity "Filter Tasks" has both, a consuming and a producing relation (cf. section 3).

When tailoring the product model to suite the given situation, these relations have to be accounted for. This means that when performing changes in any of the activities or artefacts, the related activities or artefacts have to be analyzed as well. If, for example, the activity "Define Tasks" had to be changed in order to define dates for the planing, the artefact "Task List" has to be changed accordingly.

Also the cross relation tailoring is dependent on the role assignment. In the example that means if the activity "Define Tasks" is assigned to an external developer and the activity "Filter Tasks" is assigned to an internal stakeholder who is waiting for the "Task List" created or updated by the external developer, it would be better to either change the roles or to assign an internal developer to the activity "Define Tasks", too. These decisions always depend on how critical the artefacts are.

5. SUMMARY AND CONCLUSIONS

In this paper we introduced a possibility to apply the product model given in [3]. The model can be project-specific tailored and thus a development process derived easily.

The paper starts with a short introduction and discusses: what the product model is and how it works. It continues

by describing the different types of OSS-development projects within a commercial environment. Different project types have been related to specific attributes. Based on these attributes, a mechanism is characterized which can be used to adapt the product model to the project type and the criticality of the developed artefacts.

The following results have been developed:

- Different project types have been identified and described in detail based on planing, steering and quality assurence attributes.
- An existing product model has been applied to build up an adequate development process. This has been done for a simple case study which only uses a small piece of the overall product model. Nevertheless, the used and described method can be expanded to the complete model.
- Furthermore, the tailoring-mechanism facilitates to regard critical elements during an OSS development project and to cope with problems arising during the life cycle, if external stakeholders are included into the project and particularly the development process.
- Last but not least, a case study has been introduced which briefly illustrates how to build up a very concrete development process in a commercial environment, that is for one of the identified project types.

6. REFERENCES

[1] Apache XML Project. http://xml.apache.org/guidelines.html, Feb 2003.

[2] Bernhard Deifel, Wolfgang Schwerin, and Sascha Vogel. Work Products for Integrated Software Development. Technical report, Technische Universität München, 1999.

[3] Jianjun Deng, Tilman Seifert, and Sascha Vogel. *Towards a Product Model of Open Source Software in a Commercial Environment*. In 3rd International Workshop on Open Source Software Engineering, ICSE 03, May 2003, 2003.

[4] Martin Fink. *The Business and Economics of Linux and Open Source*. Prentice Hall, 2002.

[5] IABG. Willkommen zum V–Modell. http://www.v-modell.iabg.de/index.htm, November 1999.

[6] P. Kruchten. *The Rational Unified Process – An Introduction*. Addison Wesley, 1998.

[7] Linux project. http://www.linux.org/, Feb 2003.

[8] Mozilla project. http://www.mozilla.org, Feb 2003.

[9] Research on open source software development. http://www.isr.uci.edu/research-open-source.html, Feb 2003.

[10] Wolfgang Schwerin. Models of Systems, Work Products, and Notations. In *Proceedings of Intl. Workshop on Model Engineering ECOOP, Cannes France*, 2000.

[11] Software development practices in open software development communities: A comparative case study. http://opensource.ucc.ie/icse2001/scacchi.pdf, Feb 2003.

[12] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1992.

Open Source Software Case Study - BuDDy Library

(This document was created using the open source software Open-Office)

Haim Cohen
Analog Devices DSP Design Center
11 Galgalei Haplada St. P.O.Box
12193, Herzlia 46733, Israel.
972-9-9715406
haim.cohen@analog.com

ABSTRACT

In this paper, we describe an open source software case study. We present the stages we gone through using the software in an EDA tool we developed. The developed tool was a neutral-context language for functional coverage definition, collecting and analysis.

Keywords

Open source software, BuDDy, BDD, ROBDD, EDA, functional coverage.

1. PROBLEM DOMAIN INTRODUCTION

In VLSI design, functional coverage is the task of defining and analyzing the test cases the design was gone through during a test. For example, when testing a DSP chip design model, we would like the DSP to be tested under all possible kinds of instructions. More difficult task will be to cover all the possible sequences of 2 instructions. An EDA tool which define, collect and analyze functional coverage tasks, is by nature dealing with very large sets of binary vectors.

To implement sets of binary vectors with sets operation support such as conjunction, disjunction, subtraction and set size, we used Boolean functions. Using Boolean functions, a set is represented by a Boolean function which evaluated to '1' only for binary vectors which are in the set.

An efficient implementation of a Boolean function is the novel data structure ROBDD [1] . ROBDDs are out of the scope of this document, but we will just mention that this is an efficient and compact data structure to represent a Boolean function, in a way that the complexity of operations depends on the entropy (or disorder) of the Boolean function, and not on the number of binary vectors it satisfies.

2. THE NEED FOR THE LIBRARY

The advantages of using an existing open source library are its immediate availability, its open nature which allows to add new features as required, and (almost most of the time) its quality in terms of bugs, due to its widespread usage (and therefore testing) and usage history.

In our opinion, the main criteria for the usage of an OSS library is the estimated work required to implement it from scratch. Most of the times it will be more efficient to use existing OSS library and invest the required time to evaluate and learn it, than to write it from scratch.

In our case, we estimated it will take too much time to implement the library from scratch. The ROBDD is a data structure with the form of a graph with special ROBDD operations. Implementing a ROBDD data structure in efficient matter with all the supported operations is much more complicated and error-prone than implementing simple data structures such as lists or hash tables. In addition, the usage we made of ROBDDs, was in such manner that bugs can hardly discovered by the end user. Therefore, there was no other option but to find a good source code which already made this job successfully.

3. EVALUATION

We choose the BuDDy [2] library for evaluation based on its previous successful usage in EDA projects presented by the library developer. Another factor for choosing BuDDy was the quality of documentation that comes with it, and its implementation which composed of low level C with C++ interface, which assures high performance together with easy usage.

The evaluation of the library was consist of 2 stages : feasibility check and performance testing. In the feasibility check we gone through the library documentation and made sure that all the ROBDD features required by our tool are supported by the library. During this check, we found out that some data retrieval functionality we need does not supported. Before we made any further progress in the evaluation, we took a look at the source code to verify that we can easily implement the required functionality by ourself. During the performance testing stage, we measured the performance of critical features, according to their planned usage by our developed tool. We estimated how the library performance will influence the tool performance. During this stage we found that we can easily reduce the size of the files the library use to store data.

4. SOURCE CODE MODIFICATIONS

After the successful evaluation of the library, we made two modifications to the source code. The first modification was to add the data retrieval functionality the library was missing. The second modification was to reduce the file size the library use. We reduced the file size using two different techniques : replacing the existing ASCII files format by a more compact binary format, and using another open source library called zlib [3], which supports in-memory file compression . The modified source code files of BuDDy were sent back to the library developer.

5. SUPPORT

Support is a delicate issue in open source software. A support can be divided to features-support, and bugs-support. In our case, we made sure that the interface supplied by the library is rich enough to fulfill all the features that are required and will be required by our tool. We implement by ourself missing features that we thought are easy to implement. The usage of this library by other projects will enable us to have some developers to contact with if some critical problem arise. The original developer of the library is now involved in other projects and no longer support the code.

6. ADDITIONAL OPEN SOURCE LIBRARIES / TOOLS USED

During the development of our EDA tool, we used some other interesting libraries/tools we thing it worth to be familiar with. (We will not mention the trivial yet essential emacs, gcc and gmake.)

6.1 LIBRARIES

Zlib - [3] a library which support in-memory read and writing of files in gzip format. The usage is very simple and the interface is similar to the stdio.h interface.

Boost – [4]A rich library which supports a variety of common facilities.

6.2 TOOLS

Doxygen – [5] A powerful tool to generate documentation from C/C++ source code in various kinds of formats, including HTML.

SWIG – [6] Simple Wrapper and Interface Generator. Using this tool an existing scripting language can be easily added with a special purpose C/C++ library, therefore adding a scripting entry to an existing C/C++ application.

7. ACKNOWLEDGMENTS

I would like to thank Jørn Lind-Nielsen, a Former Ph.D. student at the IT University of Copenhagen Denmark, for the state-of-the-art ROBDDs library he developed.

8. REFERENCES

[1] Randal E. Bryant. *Graph-based algorithms for Boolean function manipulation.* IEEE transactions on Computers, 8 (C-35):677-691,1986

[2] Jørn Lind-Nielsen. *BuDDy - A Binary Decision Diagram Package.* http://www.itu.dk/research/buddy/ .

[3] Jean-loup Gailly (compression) and Mark Adler (decompression) . Zlib - data compression library, which lets you compress or decompress data in memory or read and write files in the *gzip* format. http://www.gzip.org/zlib/

[4] Boost - http://www.boost.org/

[5] Doxygen - Doxygen is a documentation system for C++, C, Java, IDL (Corba and Microsoft flavors) and to some extent PHP and C#. http://www.stack.nl/~dimitri/doxygen/

[6] SWIG - SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. http://www.swig.org/

Applying a Reference Framework to Open Source Process Discovery

Chris Jensen
Institute for Software Research
University of California, Irvine
Irvine, CA, USA 92697-3425
cjensen@ics.uci.edu

ABSTRACT
The successes of open source software development have inspired commercial organizations to adopt similar techniques in hopes of improving their own processes without regard to the software process context that provided this success. This paper describes a reference framework for software process discovery in open source software development communities that provides this context. The reference framework given here characterizes the entities present in open source communities that interplay in the form of software processes, discusses how these entities are encoded in data found in community Web spaces, and demonstrates how it can be applied in discovery.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management– *lifecycle, software process models*

General Terms
Management, Measurement, Documentation.

Keywords
Reference Framework, Process Discovery, Open Source

1. INTRODUCTION
Open source software development has existed for decades, though only more recently has it piqued the curiosity of industry and academia. While many, like Eric Raymond in his essay, "The Cathedral and the Bazaar" and Garg [5], with his work on corporate sourcing, have extolled the virtues of the open source development paradigm, seeking methods of bringing the benefits of open source to industry, we still lack an understanding of the process context that enables such successes and in which these techniques lie. With this understanding, we may analyze other process activities and social and technical factors on which these techniques depend, whether they are compatible with existing processes and the larger organizational landscape, and how process techniques may be configured to realize such benefits as have been seen in an open source forum. However, process engineering activities for such analysis and that guide redesign and (continuous) improvement all require a process specification. Thus motivates our interest in process discovery. In previous work [7], we demonstrated the feasibility of automating process discovery in open source software development communities by first simulating what an automated approach might consist of through a manual search of their online Web information spaces. Here, we discuss an approach to constructing the open source software development process reference framework that helps make such automation possible. This framework is the means to map evidence of an enacted process to a classification of agents, resources, tools, and activities that characterize the process. In traditional corporate development organizations, we may be able to readily determine such things by examining artifacts such as the org-chart and so forth. But open source communities often lack such devices. While components of the framework may be known, no such mapping framework exists that enables open source process discovery.

2. RELATED WORK
Weske, et al. [17] describe what they refer to as a reference model for workflow application development processes, though theirs is more of a software development lifecycle model than a software development reference model and provide no insight for mapping the Web information space to a process.

Srivasta, et al. [16] details a framework for pattern discovery and classification of Web data. The discussion relates site content, topology, session information garnered from site files and logs and applies association rules and pattern mining to obtain rules, patterns, and statistics of Web usage. However, they offer no help in constructing the pattern discovery techniques that process the data to arrive at those usage rules.

Lowe, et al. [8] on the other hand, propose a reference model for hypermedia development process assessment. This model, however lacks their domain model does not reflect software development and their process meta-model is awkwardly configured. Nevertheless, the overlap between hypermedia development and open source software development makes is apparent in comparing their reference model with the one presented here.

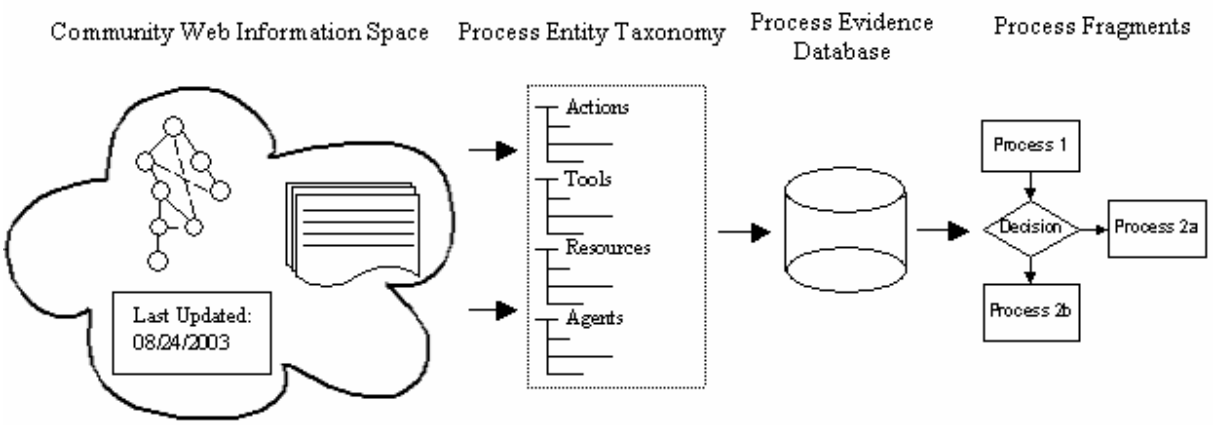

Figure 1. The open source software development reference framework, mapping Web content, structure, and usage/update data to process entities, which are assembled into process fragments via association rules

3. ESSENTIAL ATTRIBUTES OF THE FRAMEWORK

The job of the reference framework is to provide a mapping between process evidence discovered by searching the community Web and a classification scheme of process attributes. Software lifecycle models in combination with probabilistic relational modeling techniques then provide guidance for integrating these relations together into a sequence of process fragments that can be pieced together to form a meaningful model of the development process. Our reference framework is based on the process meta-model of Noll and Scacchi [12]. This meta-model consists of actions, tools, resources, and agents. Whereas Lowe and associates adapted the Spearmint framework, the skill level of the agent is unnecessary for the specification of the software development process. The abstract resource entity is likewise excessive as it caries little semantic benefit in software development process specification. These ingredients are not specific to software development processes, however the reference framework is domain specific. Furthermore, some variance is expected between communities, based on the community size, the extent of its maturity, and preferences of the individuals. Thus, while it is not possible to assert that any given community uses a specific testing suite, it is likewise impossible to say that they use a testing suite at all. However, that is the purpose of process discovery, and not the reference framework. With this in mind, we can discuss the contents of the framework.

Surveys of Apache [1], Mozilla [3], and NetBeans [13] led to a taxonomy [14, 15, 18] of tasks, tools, resources, and roles common in open source development. The approach chosen characterizes these process entities in two dimensions: breadth (e.g. communication tools, code editing tools, etc.) and genericity (e.g. an instant messaging client as a type of synchronous communication tool subset of the larger category of all communication tools). This classification scheme is necessary in order to relate instances of process entities to their entity type, which may then be associated with related entities (such as other tasks, tools, resources, and roles).

4. PROCESS MAPPING OF OSSD WEB INFORMATION SPACES

As noted elsewhere [7], there are three dimensions of the information space that encode process evidence:

- Structure: how the Web of project-related software development artifacts is organized
- Content: what types of artifacts exist and what information they contain
- Usage Patterns: user interaction and content update patterns within the community Web

The structure of the community Web is evident in two forms. The physical form consists of the directory structure of the files of which the site is composed. But, it is also apparent on a logical level, in terms of the site layout, as might be given by a site map or menu. These may or may not be equivalent. Nevertheless, each layer in the hierarchy provides a clue to the types of agents, resources, tools, and processes of the community. Structure hierarchy names may be mapped to instances of tools, agents, resources, and activities found in the open source software development meta-model taxonomy, thus fulfilling the first role of the reference framework. Additionally, directories with a high amount of content, both due to file numbers and file size may indicate a focus on activity in that area. Claims such as these may then be reinforced or refuted based on additional information gathered during discovery. Common to most open source communities are mailing lists and discussion forums, source repositories, community newsletters, issue repositories, and binary release sections, among others. The mere presence of these suggests certain activities in the development process. These also signal what types of data may be contained therein. If we just look at source repositories, we can obtain a process specification of a limited set of activities- those that involve changes to the code, just as issue and bug databases tell us that some testing is done on which the issue reports are based. In some communities, issue reports are also used to file feature requests. Such information may also be found within discussion forums or email lists.

The bulk of the process data is found within the content of Web artifacts. Much of the mapping consists of text matching between

strings in artifacts such as web pages, and email messages and process related keywords as was demonstrated for structure-based data. In the case of web content, we are also looking for items like date stamps on email messages to place the associated events in time, document authors, and message recipients. In some cases, it is possible to uncover "how-to" guides or partial process prescriptions. Like other content, these may not accurately reflect the process as it is currently enacted, if they ever did. Therefore, each datum must be verified by others.

Usage patterns, like content size, are indicators of which areas of the Web space are most active, which reinforces the validity of the data found therein and also what activities in the process may be occurring at a given time. Web access logs, if available, provide a rich source of data. Page hit counters and last update statistics are also useful for this purpose. Work by Cadez [4] and Hong, et al [6] demonstrate two techniques for capturing Web navigation patterns, however neither can be done in a strictly noninvasive manner. The first cannot provide tours of the Web space and the latter requires members to access the community Web through a proxy server used to track trips.

OSSD artifacts vary along these three dimensions over time, and this variance is the source of process events. To effectively discover processes, our reference framework must be able to relate artifacts in the community Web space with process actions, tools, resources, and roles.

5. RESULTS

Our experiences in process discovery have shown this framework to be adequate and effective for use in discovering software development processes in OSSD communities. Nevertheless, open source communities vary drastically in size and process due to factors such as degrees of openness, product, motivations, authority structure, and more. These all affect the development paradigm and, in turn, the process and the landscape of the community Web space. The challenge in process discovery is then, determining relationships between entity instances discovered. A directory such as "x-test results" is positive evidence that some sort of testing is conducted. It is likely that the files in this directory relate to this testing. Additionally, hyperlinks in the content of these artifacts may point to other sources of testing-related evidence as indicated by the context of the reference. Detecting relationships between unlinked or indirectly linked artifacts is more challenging. These connections may be established by analyzing the context of the data collected in light of a priori knowledge of software development practices provided by the process entity classification scheme. For example, the automated XTest results report summary found in the subdirectory "xtest-results/netbeans_dev/200308200100/development-unit" [11] of the NetBeans community Web may be linked to the "Q-Build Verification Report" in the QA engineer build test subdirectory "q-builds" [10] even though there is no hyperlink to relate them by observing a match between the build numbers found on each page, which can, in turn, be matched with a binary file found on the "downloads" page [9]. This shows a relation between automated testing, manual testing, and source building efforts. Date stamps on each artifact give us a basis to assert the duration of each activity. Whereas structure and content can tell us what types of activities have been performed, monitoring interaction patterns can tell us how often they are performed and what activities the community views as more essential to development and which are peripheral.

6. DISCUSSION

Edward Averill [2] states that reference models must be a set of conceptual entities and their relationships, plus a set of rules that govern their interactions. The reference framework described above does this by defining a particular application domain, fully classifying it without prescribing how particular roles, resources, tools, and activities should be assembled, or which meta-model entities are required for a process. In doing so, the reference framework is therefore community and process model independent. It is also discovery technique independent. Though we have applied it to discovery through manual search of the community Web information space, there is nothing in the specification that restricts its application to a more automated approach to process discovery as is our goal.

The reference framework is development process independent but it is not independent of the classes of tools, agents, activities, and resources. If a new role, for example, is incorporated into the development process, it must be added to the framework in order to be found through automated discovery techniques. It is worth recalling that the resulting process model shows an example of a process instance, which is subject to variation across executions. The degree of variation between instances may indicate stability and maturity in the process, as well as showing signs of a direction of evolution.

Though we have outlined a course of framework formation for an abstract open source development paradigm, it is a framework that may easily be tailored to communities with commercial-corporate influences such as NetBeans and Eclipse, as well as corporate source projects, adjusting the meta-model taxonomy in terms of tool instances, roles, etc. to suit the development paradigm.

7. ACKNOWLEDGMENTS

The research described in this report is supported by grants from the National Science Foundation #IIS-0083075 and #ITR-0205679 and #ITR-0205724. No endorsement implied. Contributors to work described in this paper include Mark Ackerman at the University of Michigan Ann Arbor; Les Gasser at the University of Illinois, Urbana-Champaign; John Noll at Santa Clara University; and Margaret Elliott and Walt Scacchi at the UCI Institute for Software Research are also collaborators on the research project described in this paper.

8. REFERENCES

[1] Ata, C., Gasca, V., Georgas, J., Lam, K., and Rousseau, M. The Release Process of the Apache Software Foundation, (2002). http://www.ics.uci.edu/~michele/SP/index.html

[2] Averill, E. Reference Models and Standards. Standardview 2, 2, (1994) 96-109.

[3] Carder, B., Le, B., and Chen, Z. Mozilla SQA and Release Process, (2002). http://www.ics.uci.edu/~acarder/225/index.html

[4] Cadez, I.V., Heckerman, D., Meek, C., Smyth, P., and White, S. Visualization of Navigation Patterns on a Web Site

Using Model Based Clustering. In Proceedings of Knowledge Discovery and Data Mining, (2000) 280-284.

[5] Dinkelacker, J., Garg, P. Corporate Source: Applying Open Source Concepts to a Corporate Environment. In Proceedings of the First ICSE Workshop on Open Source Software Engineering, (Toronto, Canada May 2001).

[6] Hong, J.I., Heer, J., Waterson, S., and Landay, J.A. WebQuilt: A Proxy-based Approach to Remote Web Usability Testing, ACM Trans. Information Systems, 19, 3, (2001). 263-285.

[7] Jensen, C., Scacchi, W. Simulating an Automated Approach to Discovery and Modeling of Open Source Software Development Processes. In Proceedings of ProSim'03 Workshop on Software Process Simulation and Modeling, (Portland, OR May 2003).

[8] Lowe, D., Bucknell, A., and Webby, R. Improving hypermedia development: a reference model-based process assessment method. In Proceedings of the tenth ACM Conference on Hypertext and hypermedia (Darmstadt, Germany, 1999), ACM Press, 139-146.

[9] NetBeans IDE Development Downloads Page, http://www.netbeans.org/downloads/ide/development.html.

[10] NetBeans Q-Build Quality Verification Report, http://qa.netbeans.org/q-builds/Q-build-report-200308200100.html.

[11] NetBeans Test Results for NetBeans dev Build 200308200100, http://www.netbeans.org/download/xtest-results/netbeans_dev/200308200100/development-unit/index.html.

[12] Noll, J. and Scacchi, W. Specifying Process Oriented Hypertext for Organizational Computing. *Journal of Network and Computer Applications* 24, (2001). 39-61.

[13] Oza, M., Nistor, E., Hu, S. Jensen, C., and Scacchi, W. A First Look at the Netbeans Requirements and Release Process, (2002). http://www.ics.uci.edu/cjensen/papers/FirstLookNetBeans/

[14] Scacchi, W. Open Source Software Development Process Model Taxonomy, (2002). http://www.ics.uci.edu/~wscacchi/Software-Process/

[15] Scacchi, W. Understanding the Requirements for Developing Open Source Software Systems, IEE Proceedings- Software, 149, 1 (February 2002). 25-39.

[16] Srivasta, J., Cooley, R., Deshpande, M., Tan, P. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data, CM SIGKDD Explorations Newsletter, 1, 2 (2000). ACM Press, 12-23.

[17] Weske, M., Goesmann, T., Holten, and R., Striemer, R. A reference model for workflow application development processes. In Proceedings of the international joint conference on Work activities coordination and collaboration (San Francisco, CA 1999). ACM Press, 1-10.

[18] Ye, Y. and Kishida, K. Toward an understanding of the motivation Open Source Software developers In Proceedings of the 25th International Conference on Software Engineering (Portland, Oregon May 2003). 419-429.

Using Open Source Industrial Projects

Using Open Source for Competitive Advantage

Jeff Garland
CrystalClear Software, Inc
Phoenix, Az USA
jeff@crystalclearsoftware.com

Abstract

In the last few years, open source software (OSS) has become widespread. Due to the pervasiveness of OSS it has become difficult to develop a large-scale software project without using OSS in some aspect of the project.

This paper will argue that companies that do not use OSS for software development are at a competitive disadvantage. In addition, the paper will discuss different categories of OSS usage and how this can allow for a streamlined selection process of appropriate OSS for industrial projects.

1. Introduction

Compared to just a few years ago, the selection of available open source software has grown dramatically. A quick perusal of sourceforge.net attests to this fact. The growth of OSS has been in almost every category of software ranging from small components to complete operating systems. Major corporations such as IBM have now embraced open source strategies. It seems clear that OSS is a key part of the software development landscape for the foreseeable future.

The growth in OSS means that companies developing projects must determine how best to leverage open source software. At one extreme a company can choose to ban all open source software products from both the development of the product or service. This extreme position puts companies at a disadvantage. The competitive disadvantage is a result of lost productivity caused by the reinvention of existing software or in inefficient operations that could be streamlined using OSS technologies. For example, building a web server for internal project use would be a waste of time and energy when perfectly suitable OSS web servers exist.

On the other extreme a company might require all tools and libraries to be open source. This may also put a company at a disadvantage since the company might have to develop software that might otherwise be cheaply purchased. A good example might be the purchase of a sophisticated 'Tab Dialog' component for building user interfaces that can be purchased for the cost of a couple of hours of programmer wages.

Somewhere between these extremes lies the most likely position for most companies. As a result, the question of which open source software provides the most benefit and the least risk to the company and the development is left open.

2. Attributes for Evaluation of OSS

In many respects there is little difference in

the evaluation of OSS and commercial software for industrial projects. Each project needs to evaluate if a given product is suitable for the task, has compatible licensing, and has appropriate support available. Companies building large software systems have faced this issue for years. Some key questions include:

1) How well does the software meet the need?
2) Is the software of high quality?
3) Is the licensing compatible with the goals of the project?
4) What is the investment required to understand and utilize the software effectively?
5) What do we risk by attempting to use this product in our project?
6) Is the product complex enough and important enough to project success that support will be needed?

The assessment of how well a particular software product meets the project needs is unique to each project. However, some of the other questions aspects of the evaluation might be generalized to reduce the effort of evaluation. The following proposes a categorization of OSS products to simplify the evaluation of licensing and other attributes of OSS.

3. Categories of OSS

OSS products can be broadly divided into libraries and frameworks, operating systems and platforms, languages, and development tools. The following provides some examples of these different categories.

Linux [1] is perhaps the best known open source operating system, but there are many others such as BSD [2]. Platforms include web servers such as Apache [3].

OSS languages include Perl [4], PHP [5], and many others. These languages offer modern features, powerful libraries, and have become quite popular for web programming and many other tasks. Perl is an example of an open source language for which there isn't a commercial equivalent (excluding commercial packaging of Perl, of course) that has the same mix of platform portability, large library, and language features. That's not to say Perl is better than say JAVA [6] for writing portable applications, only that for some applications Perl is an excellent candidate.

Development tools include a wide range of products including compilers such as gcc, version management tools such as CVS (Concurrent Versions System) [7], Make tools such as GNU Make, documentation generators such as Doxygen [8], and collaboration tools such as Wiki [9].

Examples of libraries and frameworks would be libraries such as Adaptive Communications Environment (ACE) [10] and Boost [11] (a collection of open source C++ libraries). There is a huge diversity of OSS libraries and frameworks available today.

4. Breakdown of the Categories

Each of the previously described categories of OSS has different attributes and tradeoffs for industrial projects. The following will attempt to characterize how these categories impact the decision to use OSS.

For the most part, use of platforms, tools, and languages is quite beneficial and low risk for industrial development projects. With tools, platforms, and languages the project intent is not typically to extend or distribute the OSS software, but simply to use it. Thus even the

most restrictive open source licenses are compatible with this sort of commercial use. Companies can freely utilize these OSS products for development and to provide services while keeping their own software source a trade secret. The obvious advantages of this sort of use include reduction in licensing costs compared using commercial tools and platforms. The overall investment to install, understand, and evaluate these various open source tools is usually modest. The risk is minimal since if an OSS tool fails to deliver the project can always replace the OSS tool with an equivalent commercial tool later in the project. On the downside, the documentation and guarantee of support is often less for OSS software compared to commercial software. However, this is far from universal since many of the OSS languages and platforms are far better documented and have a much wider range of support options compared to their commercial alternatives.

Open source languages represent an example of where OSS provides significant advantage to organizations with little risk. OSS languages like Perl are so widely used that many experienced Perl programmers are available. Perl also has an advantage of being available on many platforms. For example, in contrast to scripting in Visual Basic, using Perl to build tools can allow a project to maintain platform neutrality while building powerful custom tools to support project development.

The use of open source libraries and frameworks is the most difficult category of open source to evaluate for commercial use. First off, the licensing must be compatible with the project. Most often this will exclude all libraries developed under the GNU General Public License (see www.gnu.org/licenses/gpl-faq.html) for details. Thus many of the libraries and frameworks are excluded for companies that need to keep their project source proprietary. However, there are many other open source licenses for libraries which do allow for commercial use. The wide range of licenses makes the evaluation of libraries much more difficult. Another aspect of libraries and frameworks is the wide variation in quality. Some of the available libraries are of low quality. On the other hand, some libraries such as Boost are literally setting the standard (in this case the next generation C++ library standard) and thus are of very high quality and of particular significance. Commercial support of libraries and frameworks is much more difficult to obtain than for platforms, tools, and languages.

5. Conclusions

Open source software is now an entrenched part of the software development landscape. Companies that ignore OSS run the risk of ignoring tools and libraries that could provide significant efficiencies at little or no risk. Languages, tools, and platforms can be used by projects despite restrictive licensing. Libraries and frameworks present a significant evaluation challenge, but can also be applied successfully to industrial projects.

6. About the Author

Jeff Garland is the founder of CrystalClear Software - a software consulting firm specializing in large-scale mission-critical software development projects. He has used open source software in many large development projects during the last 7 years of his 17 year career. He is also the primary developer of the Boost date-time library - an open source C++ library. Jeff is the co-author of Large Scale Software Architecture: A Practical Guide Using UML Large Scale Software Architecture: A Practical Guide

Using UML published by Wiley and Sons. He holds a Master's degree in Computer Science from Arizona State University and a Bachelor of Science in Systems Engineering from the University of Arizona.

7. References

[1] Linux www.linux.org

[2] FreeBSD www.freebsd.org

[3] Apache www.apache.org

[4] Perl www.perl.org

[5] PHP www.php.net

[6] JAVA www.sun.com/java

[7] CVS www.cvshome.org

[8] Doxygen www.doxygen.org

[9] Wiki www.c2.com/cgi/wiki

[10] ACE www.cs.wustl.edu/~schmidt/ACE.html

[11] Boost www.boost.org