A Feedback Based Quality Assessment to Support Open Source Software Evolution: the GRASS Case Study

Salah Bouktif, Giuliano Antoniol and Ettore Merlo
Department of Computer Science, École Polytechnique de Montréal,
C.P. 6079, succ. Centre-ville Montréal (Québec) H3C 3A7
{salah.bouktif, giuliano.antoniol, ettore.merlo}@polymtl.ca

Markus Neteler
ITC-irst – Istituto Trentino di Cultura
Via Sommarive, 18 - 38050 Povo (Trento), Italy

Abstract

Managing the software evolution for large open source software is a major challenge. Some factors that make software hard to maintain are geographically distributed development teams, frequent and rapid turnover of volunteers, absence of a formal means, and lack of documentation and explicit project planning. In this paper we propose remote and continuous analysis of open source software to monitor evolution using available resources such as CVS code repository, commitment log files and exchanged mail. Evolution monitoring relies on three principal services. The first service analyzes and monitors the increase in complexity and the decline in quality; the second supports distributed developers by sending them a feedback report after each contribution; the third allows developers to gain insight into the "big picture" of software by providing a dashboard of project evolution. Besides the description of provided services, the paper presents a prototype environment for continuous analysis of the evolution of GRASS, an open source software.

1 Introduction

The laws of software evolution developed 30 years ago by Lehman [13] represent an attractive theory for the software engineering community. To accept, refute, formulate or redefine these laws presents a major challenge to many empirical studies on software evolution. However, there is a consensus in the software engineering community to consider certain Lehman laws as guidelines for understanding evolution problems and proposing innovating solutions. In this study we rely on three Lehman laws as guidelines to address certain evolution problems and propose solutions. Our focus is on Open Source Software (OSS) evolution.

Indeed, many studies were done to understand the evolution of OSS and to evaluate the applicability of Lehman laws on OSS evolution.

Godfrey and Tu [6] studied the Linux Kernel from 94 to 99 and discovered a super-linear growth rate of its size (more than two million SLOC). The same finding was confirmed for Vim text editor. In another in-depth study investigating 96 releases of Linux Kernel [22], it was reported that the total coupling between Linux modules grows exponentially. The same study concluded that without effort to alter the coupling explosion, Linux Kernel will become unmaintainable.

Antoniol, Merlo and otherss also investigated the evolution of Linux Kernel by considering code similarity at function level between versions [2, 16]. Analysis of evolution of mSQL was reported in [1].

Another OSS, Gnome, was studied to discover a causal relationship between the increasing number of developers and the super-linear growth rate. Mockus `et al.` in a study on Mozilla and Apache [17], states that the approach of coordination used in Apache project (100 KSLOC) to fit the work of each developer into the whole system works well for small projects. In contrast, the Mockus study suggests that the Mozilla core team needs to have more formal means of coordinating its work.

According to Scacchi [21], the growth of X-window and GCC (GNU Compilers Collection) is less rapid than other OSS because these were originally developed in an earlier software developing era (pre-Web).

From these studies, we conclude that continuing change, increase in complexity and decline in quality are common phenomena in OSS evolution. Specific causes for these problems, related to a specific OSS in its environment, are stated in the previous studies and can be eventually generalized for the whole OSS domain. Besides these causes, we report other opinions and interpretations of other pioneers of software engineering. Kemerer [10] attributed the previous problems of software evolution to a lack of knowledge of the evolution/maintenance process and of the causal relationships between software evolution tasks and their outcomes. Jones reported in [9] that according to Pfleeger: "there is not one person who has an overview of the whole project", and more time must be spent to communicate the big picture of project to everyone in every position because "The people working on the pieces need to know how their one piece fits into the entire architecture."

We believe that the causes of the software evolution problems related to the Lehman laws are more present in the OSS and their consequences are more common.

Several researchers have addressed the problem of remotely monitoring some software characteristics. In [20] OSS has been remotely measured and analyzed by CVSAnalY tool which accesses the version control repository of a system and provides measurements and analyzes automatically and non-intrusively. Historical data about the project and its contributors can therefore be processed and results displayed using a Web interface. Analyses mostly involve statistical analyses and inequality coefficients computation on some evolution distributions.

Distributed Continuous Quality Assurance of OSS has been investigated in [14, 23] where the task of quality assurance has been distributed on several sites based on availability, and results are then merged. The advantages include short development cycles, consistency of the global view of system configuration constraints, and ensuring coherency and reducing redundancy in QA activities.

Remote dependability has been investigated in [7]. Statistical models are used to predict failures in software under execution which is remotely monitored, and corrective action is taken when required to increase the residual life of some software executions.

Remote monitoring tasks can be split across several instances of the software to be analyzed so that lightweight instrumentation can be effective and execution information can be merged to compose the overall monitoring picture as described in [5, 8].

In this paper, we propose a general approach based on remote and continuing analysis of OSS evolution related to [20] to mitigate software degradation and risks. This approach works by analyzing, identifying and prioritizing evolution problems and needs in order to better control evolution problems and risks such as increasing complexity. Regardless of the geographical location, it allows the OSS developer/maintainer to be informed about the big picture of the entire system. This picture includes quality indicators of the OSS, potential open problems, information about developers' contributions, etc. Our approach allows the developer/maintainer to get near real-time feedback after working on a piece of code and to know how it fits into the whole architecture.

A particular solution (a prototype) of our remote, continuing and feedback-driven analysis environment is proposed for the OSS GRASS GIS.

The contribution of this work can be summarized as follows:

- mitigation of the ripple effect of the OSS evolution such as those stated by Lehman laws.
- architecture and plugins ensuring the remote control and analysis of OSS evolution activities
- quality remote monitoring
- scheduling of quality improving refactoring activities
- real-time feedback to developers about their contributions

The remainder of this paper will be organized as follows: Section 2 describes our remote, continuing and feedback-driven approach for OSS evolution analysis. Section 3 presents a particular solution of our approach for the OOS GRASS GIS. Section 4 presents the results of the facilities provided to the GRASS developer/maintainer team. Section 5 presents the lessons learned and future work. A conclusion is drawn in Section 6.

2 Remote Open Source Software Evolution Analysis Approach

2.1 Principle and objective

When studying the evolution problems of OSS, analyzing causes and searching for interpretation of their subsequent phenomena, we identify, three weaknesses in the OOS development/maintenance approach; (1) lack of a mechanism to control the rapid growth, continuing increasing complexity and declining quality of OSS systems, (2) lack of automated feedback reporting the effects of a developer/maintainer contribution on the quality of the system and (3) lack of formal means to learn at least periodically about the big picture of the whole OSS system. To solve these problem, our re-

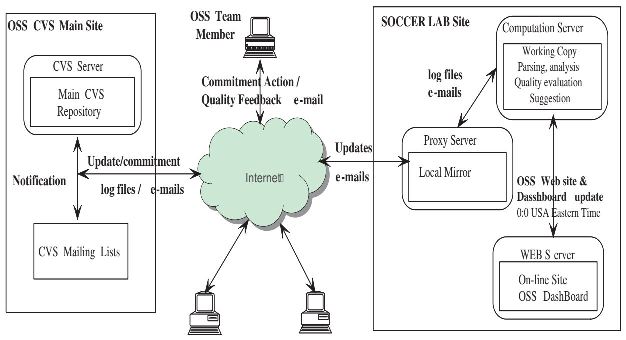

Figure 1. Infrastructure for remote OOS evolution analysis

mote OSS evolution analysis approach provides a set of services using the CVS versioning system repository as data source. The information provided by CVS log files for a commitment include date, changed file name, developer name, revision number, etc. The services provided by our approach use as software infrastructure a Plugin-based Architecture of software Maintenance (PAM), which provides a set of basic software components for general maintenance purposes. It also allows the plugin of other specialized components for specific problems, projects or objectives of maintenance. As physical infrastructure, our remote analysis approach uses the Internet as communication support between main CVS site and the SOftware Cost-effective Change and Evolution Research (SOCCER) laboratory (See Fig. 1). The CVS site is sheltering the OSS system versioning repository and the SOCCER lab. is providing automated evolution services and quality control and improvement facilities. Three main services are proposed in our approach to avoid problems related to Lehman laws; the growth, complexity and quality control mechanism (Section 2.2), the Feedback-driven communication service (Section 2.3) and the OSS evolution dashboard service (Section 2.4).

2.2 Growth, complexity and quality control mechanism

Growth, complexity and quality controlling mechanism, in brief, the quality controlling mechanism, is activated periodically or after a number of developer interactions (i.e., commitments). The time between activations or/and the number of commitments can be decided heuristically, based on statistics collected on the deployed architecture or with respect to the importance of the developer change, which can be determined by using information from the commitment log file.

To control the growth of the code size and the increase of the complexity, we parse the code, and compute at different levels (i.e., function, class or file level) a set of metrics such as size, complexity, in and out coupling, cohesion indicator, etc. Then, we use three methods to control the new metric values. The first method consists of computing the metric value delta and tolerating a certain percentage of variation. This is followed by a more robust and statistical method of checking where each metric value is positioned in the corresponding box plot computed before the current change(s). The third method, which reflects an extreme programming attitude, consists of verify how far the metric is from standard values (e.g., 20 LOC for the size). For metrics at system level, such as the total size of the OSS system, can be controlled by verifying whether their variation is sub-linear, linear, super-linear or exponential. Besides metric values control, which is a preventive process for decline in quality, the quality control mechanism also provides an intelligent feature of scheduling, prioritizing and optimizing evolution activities in order to improve quality.

The quality control mechanism uses the plugins of the PAM architecture, namely a code parser, metric extractor and viewer, statistical analyzer, quality modeling, evolution effort estimator, activity scheduling, and optimizing plugin. These plugins are described in detail in [4]. It is worth noticing that the quality control mechanism is extensible and can use other plugins in the PAM architecture depending, for example, on the quality factor being controlled (See Fig. 2).

2.3 Feedback-driven communication service

The feedback-driven communication service, in brief, feedback service, is automatically activated after each commitment of a developer to send him a feedback. The idea behind this service is that the developers need to know how well their changes, additions, or general contributions fit into the whole OSS structure. On the other hand, the also service starts when users/developers suggest or comment on some evolution actions as a feedback to other developers/users. The reason behind the second activation is that developers/users can identify problems that they cannot resolve because they do not fit their skills and knowledge. The feedback-driven communication service is

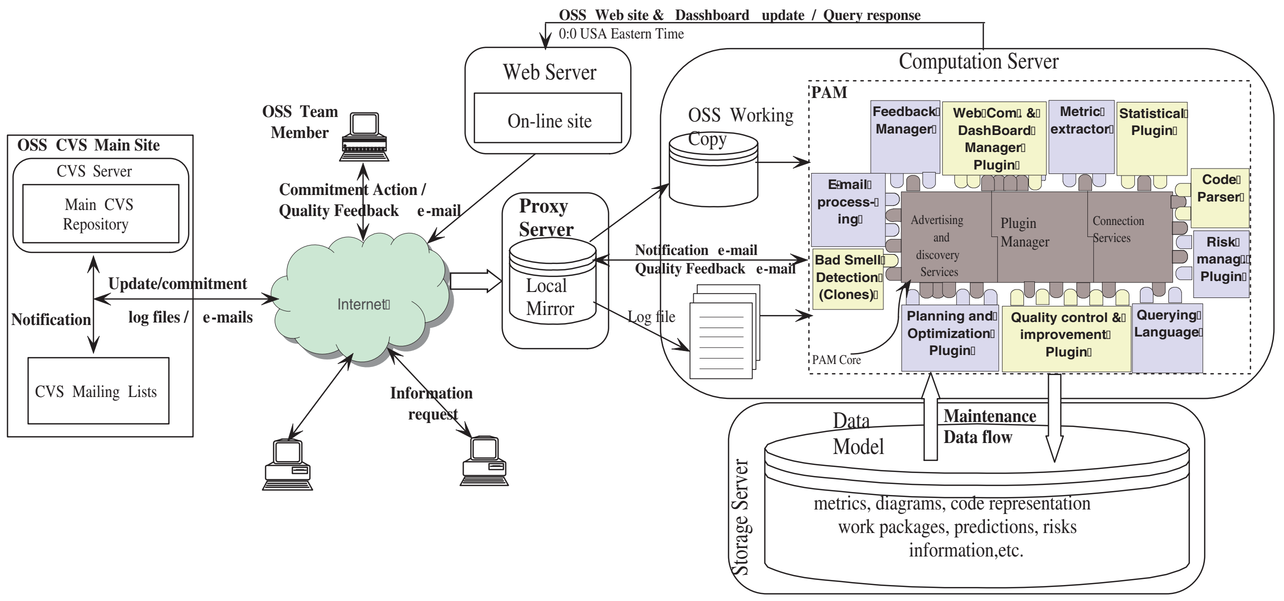

Figure 2. Remote OOS evolution analysis using PAM architecture

basically implemented by sending and receiving message to and from the message processing plugin in the PAM architecture (See Fig. 2).

After each developer's contribution, various metric and quality indicator values are updated and analyzed (see Section 2.2), and then a message is sent to the developer reporting the effects of the contribution on the complexity and quality of the OSS system. The reflected effects can be negative if there are abnormal behaviors such as adding clones and very complex functions qualified as *monsters* (all metrics beyond the outer limits) or decline of a quality factor. This service uses the OSS mailing list as a source of developer information.

On the other hand, the message processing plugin collects feedback from users/developers about new suggestions or comments of evolution actions. These are analyzed to determine the most valuable new changes to be planned in the next period, which can be published by sending message to all developers or consulted via the dashboard (See the next Section).

2.4 OSS evolution dashboard service

OSS evolution dashboard allows everyone, regardless physical location, to learn about the overall picture of the whole system and its evolution activity.

Like the quality control mechanism, the dashboard service relies on a number of periodical activities. For example, quite detailed snapshot of the OSS system and its evolution is captured periodically. The first source of dashboard information is the OSS code, which is periodically parsed; then three families of metrics (coupling, cohesion, and size-and-complexity) are extracted. Depending on the features to exhibit in the dashboard, metrics values are used in a straightforward way, transformed into other forms by statistical or/and graphical devices, or combined using various techniques to model or predict quality factors. The second information source for the OSS evolution dashboard is the message processing. From the analyzed exchanged messages and feedback between users/developers, valuable suggestions and comments can be formulated in terms of new functionalities and changes to be carried out with different priorities. These requirements give a better idea of the desired evolution of the OSS system. A third source of information for the dashboard is extracted from the commitment log files in the CVS repository system including important dates of major changes, the activity involvement of developers, etc.

Indeed, the OSS evolution dashboard can contain graphs, indicators of quality, previous major changes and changes roadmap, evolution activity planned for the next period. It can also show information about the major problem being treated in the current period, the more active development team members, the most stable part of code, the most changing (unstable) code, etc. Besides these features, the dashboard offers a user friendly interface that allows developers/maintainers to interact with the code and all the relevant data for the OSS evolution in order to support their contributions. This is possible thanks to a query-based language that can, for example, respond to queries for all the functions using "Map" as argument type.

Plugin Category/Name	Used by
Code source parser	All three services
CVS log file processing	All three services
Metric extractor	All three services
Statistical analyzer	All three services
Quality control	All three services
Message processing	Feedback
Feedback Manager	Feedback
Querying language	Dashboard
Web visualization	Dashboard
Dashboard Manager	Dashboard

Table 1. Plugin categories for Remote and feedback-driven OSS evolution analysis

2.5 PAM architecture

To help the process of designing a software evolution environment, we propose a generic CASE tool architecture that make it possible to integrate services supporting quality improvement and risk mitigation. Inspired by the hardware system field and others, this architecture, called Plugin Architecture for Maintenance (PAM) is based on plugin components and provides an increasing number of diversified automated tasks, i.e., plugins supporting software evolution.

Alternative approaches to achieve distribution while offering services and flexibility include architectures based on web services, J2ee, Java beans, process interacting through sockets, and so on.

Although from an architectural point of view, it would be interesting to investigate and compare different architectures to support maintenance, we believe that a plugin based distributed architecture promotes extensibility, flexibility, evolvability and openness. PAM offers the possibility to add new automated activities by extensions usually called architecture snap-ins or plugins. The components are plug-able to the core of the architecture. In order to integrate a remote and feedback-driven analysis for software evolution, PAM is used as a software infrastructure for the quality control mechanism, feedback service, and dashboard service. Fig. 2 shows a PAM composition prototype used to support the remote analysis of GRASS OSS project evolution.

In addition to the plugins discussed in Sections 2.2, 2.3 and 2.4, the categories of the plugins needed for remote and feedback-driven analysis are summarized in Table 1 and used in the PAM architecture.

3 Case study

As stated in the introduction, our remote monitoring architecture has been applied to monitor *GRASS*, a large scale open source GIS. The following Section summarizes the history of GRASS, the monitoring project, and describes implemented plugins.

3.1 The Monitoring Project Startup

GRASS was originally developed by the U.S. Army Construction Engineering Research Laboratories (USA-CERL, 1982-1995), as a tool for land management and environmental planning by the military. It has evolved into a powerful utility with a wide range of applications in many different areas of scientific research and is currently used in academic and commercial settings around the world.

GRASS provides an ANSI C language API with nearly 1000 GIS functions used by *GRASS* commands to read and write maps, compute areas and distances for georeferenced data, and visualize attributes and maps. Details of *GRASS* programming are covered in the "*GRASS* 6.1 Programmer's Manual" [18]. GRASS6 characteristics are summarized in Table 2.

The GRASS6 development team perceived that, given the substantial effort toward quality improvements, different means were needed to ensure a continuous monitoring and close to real time feedback on GRASS6 quality status.

It was also perceived that the imported code base has some quality related issues such as old style Kernighan and Ritchie (K&R) code, mixed use of wrapped and unwrapped dynamic memory allocation functions (e.g., GRASS6 uses a wrapped `malloc` called `G_malloc` to guard against null pointers), or a mixed use of symbolic and numeric constants in array declarations. These specific issues, though known and relevant, were considered so frequent to be too expensive to be manually fixed at the entire project level.

Between August–November 2005, several meetings between GRASS developers and SOCCER Lab researchers took place and it was decided to evaluate the feasibility of monitoring GRASS6 evolution on the basis of a previous collaboration [19]. The focus of the monitoring environment was identified in quality and evolution monitoring. The environment desiderata were stated as being non intrusive (i.e., not change or alter the GRASS6 CVS repository performances or content), with the ability to schedule evolution activities (e.g., clone removal planning) and provide feedback with reactive behavior, close to real time, after source code modification.

A first development milestone for the environment was established in December 2005. In particular, upon development of the first core plugins, the *GRASS* 6.1-CVS development snapshot of December 5, 2005[1] was used as a starting point to initiate our PAM architecture, discuss priorities with GRASS developers and beta test the distributed monitoring architecture.

Directories	541	Functions	8054
C Files	2486 K&R	Functions	820
Header Files	591	Perfect Clones	274
C KLOC	510	Near Duplicated Clones	720
Libraries	45		
Applications	580		

Table 2. GRASS6 key statistics (CVS Dec 5, 2005).

3.2 Implemented Plugins

As stated, the first analysis of the monitoring problem revealed that two behaviors were needed; the first is a reactive behavior activated when the GRASS6 main CVS repository is modified or a developer accesses PAM services via WEB, the second is the ability to run routinely scheduled tasks (e.g., each weekend re-compute the schedule of evolution activities). Furthermore, it was obvious that the CVS main repository, hosted at Osnabrueck, Germany, could not be exploited to host PAM services or hacked to simplify PAM implementation. Finally, a lightweight and fast update of information on machines hosting quality and evolution data was required.

These consideration led to the conceptual physical layout shown in Fig. 1. GRASS6 CVS central repository is mirrored on a SOCCER Lab machine; this mirroring is an exact and identical copy kept synchronized. All computations, including configuration, compilation or metrics extraction are performed on the *working* copy, which holds all intermediate and final results. As shown in Fig. 1 a further instance of the GRASS6 CVS repository is used. This is essential particularly when several files are changed at the same time and a non negligible time to update the working copy is needed (e.g., to extract software metrics). In such a case, two accesses, close in time, could provide very different information. To avoid this, was decided to keep on line a stable copy of the information and to update it as soon as a new stable state is reached by the working copy. This layout ensures a fast update of CVS mirror, information coherence, and non interference between different update tasks and parallel computations. As the monitoring system is being active and on line, Fig. 1 organization has been replicated to allow developing, debugging and testing of components and plugins.

[1]Downloadable from `http://grass.itc.it`

Two families of plugins have been developed, namely *General Purpose Plugins* and *Specialized Plugins*. Among general purpose plugins are the following: Metrics extraction, statistical analysis and visualization; Code transformation and code instrumentation; Activity planning and optimization; WEB communication and WEB services and Feedback plugins. These are likely to be common to several projects, domains and environments. The second category, specialized plugins, consists of technology, team, and problem dependent plugins. They are essential to actually implement evolution activities. They can depend on any element of the evolution environment and are specialized in some activities. Examples in this category include: effort modeling plugins, parser tools and tester plugins.

In the following we will provide a brief description of already developed or in progress plugins, general purpose and specialized, involved in the evolution monitoring analysis project.

Metrics extraction, statistical analysis and visualization. These plugins can be considered as composed of three services: extraction, statistical analysis (e.g., box plot or outliers identification) and visualization of software metrics. At the time of writing distinct families of plugins are available for project related metrics (e.g., number of changed files per day or week, changed volume per unit of time or programmer), object-oriented languages (e.g., Java, and C++), procedural languages (e.g., C), interface definition and communication languages (e.g., CORBA - IDL). Metric visualization deals with information presentation; several proposals exist to represent and navigate in large amounts of information [3, 12]. Visualization is also essential to grasp the overall picture as well as the details. At present we provide basic statistical analysis via histogram, multiple box plots, bar charts, and time series visualization (e.g., evolution of clone percentage over a given period of time, see Fig. 3).

Code transformation and code instrumentation. Like the previous plugins, these can be considered as general purpose from a high level point of view. Code transformation plugin is limited in specifying the interface in that actual transformations depend on language, project, idioms and technology. We are currently working on developing plugins specialized in the integration of available transformation engines.

Activity planning and optimization. The rationale of this plugin is to support the new approach to software evolution, namely quality improvement, risk mitigation and cost reduction. Indeed the resolution

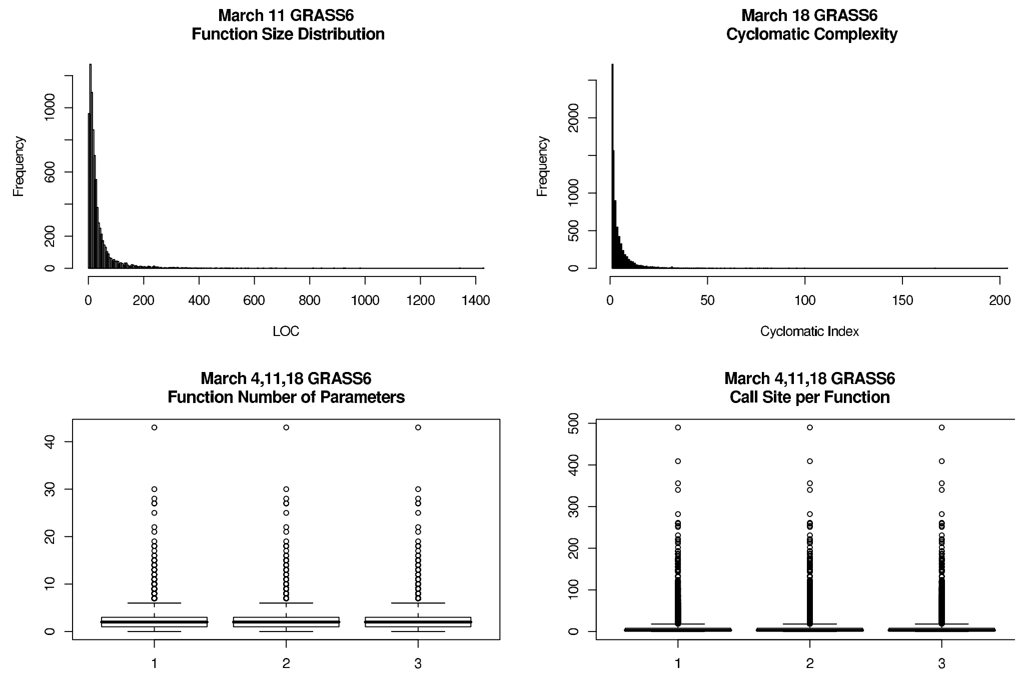

Figure 3. GRASS6 March 2006: various dashboard formats.

of many problems in this area consists of making decisions and compromises. In particular, managing maintenance activities in cost-effective ways while ensuring high quality is a very difficult problem that can be tackled via optimization and search-based approaches. Activity planning and optimization is valuable support of a decision-making process. This plugin implements very well known search-based techniques such as genetic algorithms, simulated annealing and taboo search. Examples of supports provided by this plugin are: a rigorous schedule and planning of maintenance activities, etc. optimizing resources allocation;

Bad code smell detection. Any large software system evolving over time is likely to develop unwanted and undesired characteristics such as duplicated code regions, unstructured or poorly structured code, anti-pattern etc. We have developed a C specialized plugin to detect two families of **bad code smell**: duplicated code also known as clones [11, 15] and C functions with complexity, size or coupling largely exceeding the median of the metric value for the given project (i.e., three times the inter-quantile above the 75% percentile).

WEB communication, WEB services and WEB dashboard. To experiment with our architecture and to develop and deploy plugins, we rely on standard Unix/Linux platforms. In particular, we used an Apache web server acting as gateway, SOAP transport, Java, Perl and Python modules. These modules are the backbone of the system. We rely on WEB communication and WEB Dashboard to perform analysis, update information and present results. WEB dashboard also provides per user authentication and secure access to confidential data.

The actual architecture with its essential elements has been deployed on the SOCCER laboratory network. At present we are monitoring, analyzing and supporting the evolution of GRASS system.

Feedback plugins. These are the core of the reactive part of the system devoted to providing developers with personalized feedback. It relies on metrics extraction, statistical analysis tools and standard Unix e-mail facilities. In particular we rely upon procmail, fetchmail, sendmail and R for the statistical part. E-mail messages are composed automatically and sent back to the developer each time a source file modification is committed. As shown in Fig. 4 the message provides information on metrics, the absence or presence of code duplication (with respect to the entire code base), and a quality ranking (e.g., metric values identified as outliers for the GRASS6 project). At the time of writing no secure or authentication mechanism was put in place for the e-mail. To minimize the risk of possible man-in-the-middle attacks we plan to add a digital signature and to switch to a certified mailer server.

Refactoring Effort estimator plugins. These plugins are specialized in estimating the effort needed for particular types of refactoring activity. In our case we already developer an effort estimator for refactoring of duplicated code at function level.

```
Hello Markus,

here a report on your recent GRASS-CVS change:

Threshold GRASS6 key metrics values at 2006-03-25 are:
---------------------------+-----------+---------+
                     median | upper     | outlier |
                            | quartile  | limit   |
---------------------------+-----------+---------+
Complexity (Cyclo):       5 |        25 |      36 |
ParamNBR:                 2 |         6 |       8 |
CalledNBR:                9 |        55 |      78 |
LOC:                     39 |       181 |     254 |
---------------------------+-----------+---------+
Measures (see below Web page for details):
 Complexity (Cyclo): a complexity index
 ParamNBR: number of passed parameters
 CalledNBR: number of called functions
 LOC: Lines Of Code

Message interpretation:
 OK : Nothing to say at all
 *  : above 75 % of value warning
 ** : this is getting close to be out of range, consider revising the code
 ***: this is a monster, definitely you should revise this function

ANALYSIS

you have changed 1 files/functions:

./grass6/vector/v.in.ascii/points.c functions

points_to_bin (Begins at: 187 Ends at: 300)
--------------+---------+-------+---------+
Complexity:   |      15 |       |      OK |
ParamNBR:     |      13 |   *** | MONSTER |
CalledNBR:    |      53 |     * |         |
LOC:          |     114 |       |      OK |
--------------+---------+-------+---------+
1 metric are OUT OF RANGE!

Clone analysis:
This file does not contain  cloned functions
```

Figure 4. Example of generated feedback (The e-mail was edited to fit the space).

4 Results and interpretation

GRASS developers were asked to provide guidelines to set up quality improvement goals. Discussion led to the decision to separate the problem of K&R code *ansification* from the longer term quality improvement goal. Unfortunately the public domain *ansification* tools don't provide an accurate code transformation, so we developed our own *ansification* toolkit. Information collected while parsing C code on K&R function location is very accurate and it was used to perform precise (including saving comment position) and semantic preserving code transformation. The process required about 6 hours of an expert in parsing and code transformation, and 8 hours of an expert GRASS programmer. Out of the 6 hours, about 5 were needed to develop the *ansification* plugin. *Ansification* was performed on a per-directory basis; GRASS developer time was needed to semi-automatically verify performed transformations, recompile the system, perform basic tests, and commit changes in the central CVS repository. This means that all 820 K&R functions were translated into ANSI style in less than two working days. At the time of writing, no error has been discovered related to such a major change. It is however worth noticing the relation with quality issues: in K&R style function parameter declaration, at function definition, is optional. Out of the 820 functions, about 20 had undeclared parameters; most of these dangerous missing declarations were unknown to developers. The type of parameters is assumed by the compiler; however, there is no guarantee on assumptions made by different compilers. The simple rebuilding of an application with different compiler may thus lead to crashes or, in the worst case scenario, to undetected data corruption.

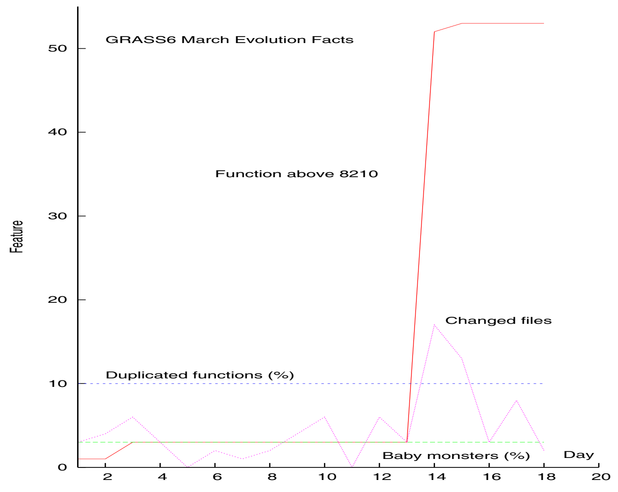

Figure 5. GRASS6 March 2006: feature evolution

GRASS developers set up priority on a code region perceived as critical. Moreover, concern was expressed on the quality of the largest and most complex functions. Complexity was in turn represented as number of parameter and cyclomatic complexity. These priority values together with initial model parameter values and data collected by two developers were then used by a calibration plugin to customize the quality and activity planning plugin. Following the initial calibration, a beta version of PAM was put on line at the beginning of February 2006. After a couple of weeks collected data and statistics were considered reliable.

Fig. 3 shows reports produced to populate the dashboard. The upper left histogram shows the function size measured in LOC on March 11, while the right histogram depicts the cyclomatic complexity histogram. No substantial variation was observed in the first month of monitoring. Summary statistics for collected function level metrics remain stable as also shown by the two Fig. 3 boxplots.

The trend observed between March 1–20, 2006 for the number of modified C files, the number of C functions, the percentage of duplicated functions and the percentage of *baby monster* is shown in Fig. 5. A function is referred as to a *baby monster* when at least one of the monitored metrics falls above the outlier threshold (e.g., the cyclomatic complexity is above 32); these are about 300 functions. Pathologic functions, *monsters*, (all metrics above the outlier limits) are both of the order of ten, did not exhibit a substantial change in the first three weeks of March. Also GRASS6 duplicated code about 10% did not change substantially. A slight reduction between December 5, 2005 and February 2006 was experienced. However, between February and March a slight increase was observed. This is also related to the fact that the clone analysis was almost unknown in the GRASS community. When exact copies are counted the rate drops to about 2.5%.

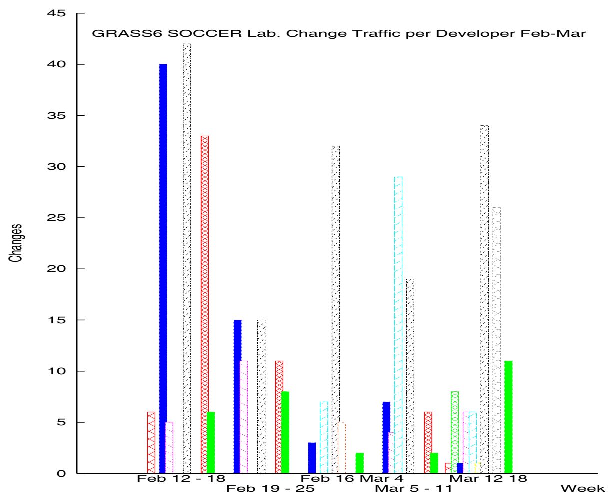

Figure 6. GRASS6 March 2006: CVS source file traffic per developer

About 45 programmers are registered as GRASS developers however, in reality, there are fewer than 20 core contributors; as shown by Fig. 6, most contributions can be attributed to an even smaller pool of five people. Names are not reported for confidentiality.

5 Lessons learned and future work

We recall that the objectives for the described environment are the remote monitoring of software evolution to avoid quality deterioration, to increase developers' productivity and efficiency, and to reduce the cost of evolution. At first, gaining confidence of developers was an issue, but, once achieved, developers were more than willing to supply help and make suggestions to improve PAM architecture for their needs. One key success factor for effective collaboration is to show developers that PAM really adds value to their daily work. Gathering priorities about the projects and the planned development together with inhibition information about parts of the project were quickly identified as an important utility in PAM.

Another success factor has been the non-invasive approach used in PAM. In general, developers receive a suggested schedule of refactoring activities that they are free to follow, reject, or re-schedule. GRASS developers are free to subscribe/unsubscribe to the automatic quality notification plugin. A prescriptive way of interacting with developers would have been rejected in the project.

Indeed, although we should have expected it, we realized that refactoring carried some risks. Uncertainty about real span of refactoring may be a source of risk; unexpected dependencies may carry refactoring farther than expected or than detected by analysis. Also, human error during refactoring can hardly be planned in the scheduler. To reduce risks, we created and implemented a file-level visualization tool that shows the differences between files during refactoring.

The implemented visualization scheme to interact with developers is very useful, but rather primitive and simple. Better schemes should be investigated in terms of ergonomic design of user interface and of supplied visual functionalities. The presented approach allowed the accidental discoveries of candidate bugs in the project, such as array size inconsistencies between somehow related code fragments. This bug discovery opportunity should be further investigated and exploited since it could be an interesting source of quality improvement. A formal definition and catalog of candidate bugs together with appropriate detection techniques could be developed and tested.

Also, although very important, the presented closed-loop between development and analysis in PAM is relatively primitive. A more sophisticated process would consider some communication scheme to specific developers, rather than the blackboard style access to the all the information.

6 Conclusion

We have reported our experience in deploying a plugin-based architecture to monitor software evolution and support programmer activity. Our approach stems from the needs of a team of open source developers in evolving an open source project, the GRASS6 application.

Like other approaches presented in the literature, we rely on parsing technology and standard tools. An initial version of our monitoring environment has been deployed and is currently used to track and manage GRASS6/GIS evolution. Developed plugins allowed GRASS maintainers to effectively carry out automated maintenance and to receive, almost in real time, feedback on performed evolution activities. Moreover, we provide them with an optimal schedule of quality improvement.

Our approach also allowed for the accidental discoveries of candidate bugs in the project, such as array size inconsistencies between somehow related code fragments. This bug discovery opportunity should be further investigated and exploited since it could be an interesting source of quality improvement at a low effort of detection. At present we are developing more sophisticated plugins, such as those required to semi-automatically deal with array boundaries inconsistency.

References

[1] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. *Proceedings of IEEE International Conference on Software Maintenance*, pages 273–280, Nov 6-10 2001.

[2] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44:755–765, Oct. 2002.

[3] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, Apr 1996.

[4] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A plugin based architecture for software maintenance. RT EPM-RT-2006-03, Department of Computer Science, École Polytechnique de Montréal, 2006.

[5] J. Bowring, A. Orso, , and M. Harrold. Monitoring deployed software using software tomography. In *ACM SIGPLAN-SOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Nov 2002.

[6] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings International Conference on Software Maintenance*, pages 131–142, 2000.

[7] K. C. Gross, S. McMaster, A. Porter, A. Urmanov, and L. G. Votta. Towards dependability in everyday software using software telemetry. In *IEEE Workshop on Engineering of Autonomic Systems*, March 2006.

[8] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying classification techniques to remotely-collected program execution data. In *13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Sept 2005.

[9] C. Jones. *Patterns of Software Systems Failure and Success.* Computer Press, Boston, Mass., 1996.

[10] C. F. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, 1999.

[11] B. Lague, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 314–321, 1997.

[12] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *proceedings of the 20^{th} international conference on Automated Software Engineering.* ACM Press, Nov 2005.

[13] M. M. Lehman. Laws of software evolution revisited. In *EWSPT*, pages 108–124, 1996.

[14] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. Schmidt, and B. Natarajan. Skoll: Distributed continuous quality assurance. In *International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, UK, May 2004. IEEE Society Press.

[15] E. Merlo, G. Antoniol, M. D. Penta, and F. Rollo. Linear complexity object-oriented similarity for clone detection and software evolution analysis. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 412–416, Sept 2004.

[16] E. Merlo, M. Dagenais, P. Bachand, J. S. Sormani, S. Gradara, and G. Antoniol. Investigating large software system evolution: the linux kernel. In *COMPSAC*, pages 421–426, Aug 2002.

[17] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[18] M. Neteler, editor. *GRASS 6.1 Programmer's Manual. Geographic Resources Analysis Support System.* ITC-irst, Italy, `http://grass.itc.it/devel/`, 2005.

[19] M. D. Penta, M. Neteler, G. Antoniol, and E. Merlo. A language-independent framework for software miniaturization. *Journal of Systems and Software*, (77):225–240, 2004.

[20] G. Robles, S. Koch, and J. M. Gonzlez-Barahona. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS '04), 26th International Conference on Software Engineering*, May 2004.

[21] W. Scacchi. Understanding open source software evolution: Applying, breaking, and rethinking the laws of software evolution, June 24 2003.

[22] S. R. Schach, B. Jin, D. R. Wright, G. Z. Heller, and A. J. Offutt. Maintainability of the linux kernel. *IEE Proceedings - Software*, 149(1):18–23, 2002.

[23] C. Yilmaz, A. Memon, A. Porter, A. Krishna, D. Schmidt, and A. Gokhale. Techniques and processes for improving the quality and performance of open-source software. *Software Practice and Improvement Journal*, 2006 (To appear).