2009 IEEE/ACM International Conference on Automated Software Engineering

An Open Source-Based Approach to Software Development Infrastructures

Yuhoon Ki and Meongchul Song
Advanced Software Research Center
Samsung Electronics Co., Ltd.
Suwon, Korea
{yuhoon.ki, meong.song}@samsung.com

***Abstract*— As software systems become larger and more complex, automated software engineering tools play a crucial role for effective software development management, which is a key factor to lead quality software systems. In this work, we present TRICA, an open source-based software development infrastructure. The name of TRICA represents its features such as Traceability, Relationship, Informativeness, Cost-effectiveness, and Automation. Essentially, in TRICA, a continuous integration tool is coupled with a software configuration management tool and an issue tracking tool. We provisioned a mechanism to connect the open source tools in TRICA so that project members use the collaborated information to solve various issues and implementation problems efficiently, and easily share forthcoming issues during the course of the project. We show that TRICA can help to decentralize risks throughout the software development cycle and achieve successful software development.**

Keywords-component; software engineering tools, continuous integration, SCM, issue tracking, open source

I. INTRODUCTION

With the ever increasing size and complexity of software systems, software projects are required to be well-managed in order to achieve better quality and productivity so that they can stay competitive in the market. Furthermore, as the trends of technology change quickly and user demands for products are diversified, many organizations collaborate with distributed teams, utilize commercial off-the-shelf products (COTS), and even adopt open source software so as to develop and deploy quality products in time. As a result, large software systems that consist of a great number of components are typically developed, tested, and maintained by many people encompassing several organizations and sometimes located in geographically distributed places.

In large scale projects, developing software systems is almost impossible without effective management [1]. Automated software engineering tools play crucial roles in managing software development process efficiently and effectively. They help managing project schedules, tracking defects or events, leaving feedback and various reports, controlling software configuration, and automating build and deployment process [2]. Automated software engineering tools benefit projects extensively when they are incorporated with the development process and complement other tools.

When many teams and developers collaborate with others, integration is critical to lead success or failure of the project. According to [3] and [4], integration errors take up to approximately 40% of program errors and 50% to 70% of testing effort is spent on integration-level test. However, due to the limitation of time and budget, many software projects still end up with *Big-Bang* integration of which risks are well understood. Therefore, it is difficult to find referential examples of applying automation tools and/or systems to avoid such integration risks in industry.

In this paper, we present our experience on open source-based software development infrastructure in industry. The rest of the paper is organized as follows. Section II briefly discusses open source and proprietary automated software engineering tools. Section III presents the structure of TRICA, our infrastructure, and discusses its features. Section IV discusses our own experiences in adopting open source tools. Finally, Section V gives future work and conclusions.

II. RELATED WORK

Software engineering tools can be categorized as software configuration management (SCM), issue tracking, continuous integration (CI), requirements engineering and testing tools depending on their use. However, we do not consider testing tools such as static analyzers, coding rule-checkers, or dynamic testing tools in this work.

SCM tools keep record of software change history, and merge and integrate changes in different local snapshots back to the main branch of the code base. Concurrent Versions System (CVS), Revision Control System (RCS), Subversion[1] are well-known open source SCM tools while Perforce and ClearCase are their counterparts for the proprietary software [5].

RCS is a simple SCM tool and useful, especially for projects developed in a single user development environment. Other aforementioned SCM tools incorporate the client-server architecture. CVS has a long history and is probably one of the most widely used SCM tools. Linux kernel, Apache, OpenOffice, and all of the open source projects in Sourceforge.net have been developed using it. According to [6], Subversion is well-known in open source community and recognized as the sole leader in the Standalone SCM category. Perforce features comprehensive

[1] http://subversion.tigris.org/

1527-1366/09 $29.00 © 2009 IEEE
DOI 10.1109/ASE.2009.73

SCM capabilities and has been licensed to more than 250,000 users at 4,500 organizations as of 2007 [6]. ClearCase is a well-known tool developed by the IBM Rational.

TABLE I. EXAMPLE AUTOMATED SOFTWARE ENGINEERING TOOLS

Category	Open Source	Proprietary
SCM	RCS, CVS, Subversion	Perforce, ClearCase
Issue tracking	Trac, MantisBT, Bugzilla	ClearQuest
CI	CruiseControl, Hudson, TinderBox, BuildBot	Team Foundation Server

Issue tracking tools offer means to provide a collaborative hub for distributed project members implementing new features, correcting faults, and retaining references for maintenance. Trac[2], MantisBT, and Bugzilla are Web-based open source issue tracking tools [7]. Trac has features including project management, ticket system, and Wiki. It also serves as a Web interface to SCM tools. MantisBT is a PHP-based bug tracking system. Bugzilla is originally developed and used by the Mozilla project. ClearQuest is a proprietary workflow automation tool developed by IBM Rational. Originally developed for tracking defects, ClearQuest also supports task management. It integrates well with ClearCase.

CI tools provide methods to build and test code, and report build status. CI is the practice of integrating early and often, so as to avoid the pitfalls of so called integration hell [8][9]. It reduces rework and helps cope with errors from early phase of the development. Thus it helps save cost and time to develop software systems. CruiseControl is a Java-based open source framework. It provides a Web interface to view the details of subsequent builds. Hudson[3] is also a Web-based open source tool written in Java. Tinderbox is a software suite that provides CI capability and used for the Mozilla Project. BuildBot is originally developed as a light-weight alternative to the TinderBox and is now used at Mozilla. Microsoft Team Foundation Server is a proprietary product that offers source control, data collection and reporting, and project tracking.

In addition to the tools listed in Table I, UML modeling tools such as Telelogic Tau and Rational Rose, and requirements management tools such as Telelogic DOORS are also widely used.

III. SOFTWARE DEVELOPMENT INFRASTRUCTURE

A. Background

The goal of our project is to develop a software platform for a large embedded system that yet to be developed. The project involves external, i.e., out-of-the-company, contractors in abroad as well as different organizations from within. As a result, the implementation of code and integrations are accomplished at various locations – several domestic laboratories in Korea and the external contractors' offices in abroad. Some of the software components are COTS and shipped in binary forms, some are to be ported from previous projects, and some are to be developed from scratch. With such background of the project, we anticipated the followings as serious potential hindrances to our project.

1) Managing deliverables produced from distributed working environment: We need to put all the deliverables under control, especially for integration and maintenance. Without a repository for configuration management, project members may have different code bases, which make the integration hard. Also they may experience difficulties to fix errors and maintain releases when they need to refer others' source code or retrieve an appropriate version.

2) Integration errors due to different development environments: Each developer would run his or her own build on one's own working machine. This may cause a well-known problem, 'It works on my machine!'[8]. Because developers use different IDEs and configurations, integration is vulnerable to fail when a developer's build process does not match with those of others'. Also, human error is one of frequent reasons to create failures during practicing the build process, especially when the integration gets complicated.

3) Big-bang integration: If a defect found during the big-bang integration, whether it is critical or minor, entails implementation changes, it easily causes delay of the project schedule. Moreover, impromptu tasks and sudden changes negatively impact on the quality of the software system and they tend to relate with failures in the market. Because our project is large and very complicated, this can be a relatively bigger risk.

4) Disagreement between the design documents and the implemented code: During the design phase, we carefully designed the software architecture to support efficient distributed development. However, design changes frequently since our project is in its research phase. Without effective sharing of up-to-date design documents, developers may work on programming based on incorrect design information.

5) Difficulties in communication and progress synchronization: Since project members participate from different locations, even some abroad, we anticipated effective communication would remain as an outstanding issue throughout the project. Insufficient understanding or miscommunication can be a latent problem which might show up in the late phase of the development cycle. When the problem is finally revealed, the result may be significantly different from other organization's expectations.

Finally, we decided to construct an infrastructure to overcome possible barriers of the project. During the software architecture design phase, we derived following requirements for the infrastructure:

[2] http://trac.edgewall.org/
[3] http://hudson.dev.java.net/

- *Project management:* Project managers should be able to set development milestones, and activity monitoring and management.
- *Source configuration management:* Developers working at geographically different locations implement and commit their code. The managed artifacts include various documents, e.g. architecture documents and various guides, as well as code. In addition, managed design documents and source code should be in agreement.
- *Automatic build and continuous integration:* Since the project is in its research phase, developers frequently modify their code. So, automatic build and continuous integration is one of the necessary practices to ensure the quality of the software system.
- *Progress and issue sharing:* Project members need to share issues and communicate easily with each other, regardless of their locations or teams.
- *Software licenses:* Since we collaborate with outside contractors, license for proprietary software is an important issue. Adopting open source tools is a strong possibility.

B. Approach

Our infrastructure, called TRICA, consists of SCM, issue tracking, and CI tools. Firstly, we chose Subversion over CVS as the SCM tool. Although CVS is a popular tool, Subversion is more adequate for our project. In Subversion, branch and merge transactions are light, and update and tag transactions are fast. Also, it keeps atomic transactions of commits and manages snapshots of whole working directory not those of each file [5]. These advantages are very important for distributed environment to make developers commit and update source code safely and conveniently.

Next, we chose Trac as the issue tracking tool. MantisBT was another issue tracking tool under consideration. However, Trac is more appropriate for our project because it provides convenient links to Wiki and Subversion. Moreover, it enables to share and keep track of ideas and issues as well as bugs [7].

Lastly, our choice was Hudson for the CI tool. It is simple to use and configure while it provides almost all the important features of CruiseControl, which was a strong candidate for us. Moreover, Hudson can be easily installed on our Linux-based server.

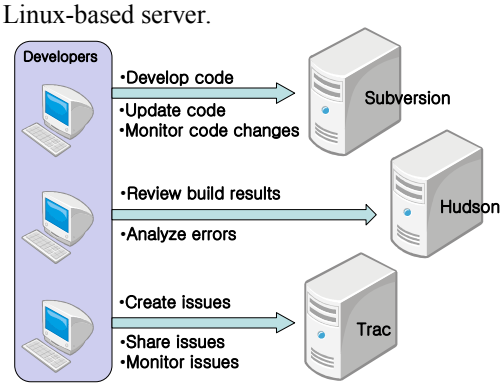

Figure 1. Subversion, Hudson, and Trac as standalone tools.

Fig. 1 shows how developers work with each tool. Developers can use the functionalities of each tool independently but our goal is to integrate them so that we can maximize the advantages of each tool, improve the efficiencies of co-located automated tools, and reduce time to find connections among different types of information. After investigating our requirements and existing features of each tool, we found a few complements to the following considerations are necessary for TRICA.

- **Each tool does not connect to the other tool as is:** There has to be something that plays a role as a link and configuration that defines a protocol among them.
- **Subversion only leaves raw information of changes such as time, date, and the person in charge:** The infrastructure should provide a method to leave annotations for changes. This makes it possible the change history maintained by Subversion is practically shared by other tools. Without the annotations, change history can lead developers into confusion because of the swarming information.
- **Automated build conducted by Hudson requires carefully planned build scripts:** The build scripts should be systematic and need to be based on the development policy. In this way, Hudson is able to get source code from the Subversion repository; build and execute the code according to the defined rules.
- **Subversion cannot automatically make source code accordant with design documents:** Although source code and design documents are under control, source code changes are not updated in design documents unless developers manually do the task. When interface or design changes are occurred in source code, it should be readily reflected in design documents which are shared by developers.
- **Issues maintained by Trac should include due date and be reported separately to each organization as well as a whole:** Simply creating an issue is not a good approach for issue tracking. Tracking issues with due date and by organization can help developers and project managers follow ongoing situations more actively under positive and light pressure.

As a result, our goal was to make the three tools as an integrated system by filling up insufficient parts of each tool. The infrastructure can be much more powerful when they work together. Project members are able to work on so called virtual working environment [10] within the TRICA infrastructure.

C. TRICA

Based on the analysis described in the previous chapter, we resolved the problems as follows:

- **Connect Subversion and Hudson with Trac**: By appropriate server configurations and additional software installations, we made the tools connected with each other. More specifically, Subversion

connects the change history maintained in its repository to Trac and creates links to related issues; Trac's issue history has links shared by the Subversion snapshots. It helps relevant issue and change sets are analyzed together to find problems and trace errors efficiently. Also, the combined information from Subversion and Trac is integrated to Hudson, i.e. each build has information of what has been integrated, who and what issues are related. With the mapping of change history to corresponding issue history, developers blamed for a failed build check corresponding change sets first, not from scratch.

- **Add macro for annotations:** The macro forces developers to leave comments on committed code and tags of the relevant issues. This enhances information sharing among project members and improves usability and readability of information because low level information such as change history is combined with high level information such as developers' annotation.
- **Employ Cmake[4] for automated build:** Cmake is an open source cross-platform build utility. When Hudson loads present snapshot from the Subversion repository and builds it automatically, it follows build rules defined by Cmake.
- **Adopt Doxygen[5] and let TRICA generate HTML documents automatically from Doxygen documents during the daily build:** Developers are required to leave Doxygen comments in their code. Also, interface specifications are written in Doxygen documents. The Doxygen comments are automatically generated into HTML documents during the build process conducted by Hudson. Therefore, developers are able to implement code correctly based on the latest interface specifications.
- **Configure plug-ins to set due dates and make Trac create reports by the organization:** The reports sorted by organization and due date, project managers are able to encourage team members and manage schedules.

Fig. 2 describes how developers and testers interact with TRICA. We describe the features of TRICA as its abbreviation represents.

Traceability. Because all the tools in TRICA record and maintain history, developers can easily trace to probable cause and related information when errors are detected.

Relationship. TRICA provides a Web-based monitoring and feedback system. In other words, TRICA is a communication window for mutual understanding and sharing issues among project members. It constructs a virtual task relationship and helps to reach consensus on the project.

Informativeness. TRICA integrates information from Subversion, Trac, and Hudson. The linked information helps project members to work more efficiently. Especially, the annotation macro enhances sharing and recording of specific information. Also, issue reports by the organization make it convenient to grasp the status of critical issues.

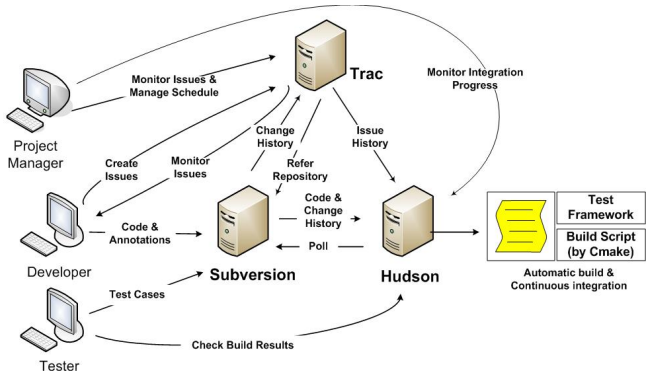

Figure 2. TRICA structure.

Cost-effectiveness. Using TRICA saved license fees for software engineering tools. Also, TRICA automatically integrates source codes incrementally so that it reduces time and efforts for integration which is considered as one of the most time-consuming tasks. In other words, it decentralizes potential risks due to the big-bang integrations.

Automation. TRICA automatically applies source code changes to current integration. As it builds source code regularly and automatically based on build scripts, it reduces load on developers and eliminates potential errors due to manual builds. Automatic generation of HTML documents from Doxygen comments helps keep agreements between the design documents and implemented code.

IV. LESSONS LEARNED AND DISCUSSIONS

Collaboration with external organizations brought in sensitive problems regarding software license and security issues. More specifically, to use proprietary development tools employed in our company, we had to resolve rather sensitive license issues with our contractors. In addition, our server for the proprietary software had to be accessed from outside, which might cause security breaches. This was a difficult problem to resolve.

Due to the reasons, we opted for choosing open source tools in order to construct a software development infrastructure. There was no financial cost to purchase software tools or license. Even though TRICA is based on open source tools, its capabilities are comparable to those of proprietary software-based framework. Many open source communities are tightly related with others and support integration with tools developed by other projects. This helped us resolve support issues that caused by using the open source tools. Furthermore, most widely used open source software has been proven with good quality and their communities are pretty helpful to get references and information based on their participants' extensive experiences [11].

As mentioned in the previous chapter, our project is currently under development so it is rather early to measure the productivity due to TRICA. However, we discuss a few of cases how we have taken advantages of TRICA. On

[4] http://www.cmake.org
[5] http://www.doxygen.org/

average, it took for us 18 hours and 20 minutes from detecting a fault and fixing it in the integration. With the exception of the maximum and the minimum values, it took 15 hours and 50 minutes to fix a fault in average. It means that if someone commits developed code with a fault into the repository and goes back home, the error is expected to be fixed just several hours after the person starts work on the next day. Detected faults may be distributed throughout entire teams but they are typically fixed less than a day.

In addition to the aforementioned case for bug fix, there is another example that TRICA contributed to save the project. During the development, there was a critical change to the software architecture. It was unavoidable to modify already developed code. At that time, we reverted to the latest stable snapshot stored in the Subversion repository. What most developers needed was to keep developing. Only a few developers were required to work on the changes. We cannot imagine what would have happened if all the project members had to find the location that had to be corrected, modify the code, and integrate it again.

In order to alleviate anxieties and maximize advantages by using open source tools, we had two strategies: right choice and active customization. While most open source tools are extensible and interoperable, they need to provide version compatibility. When candidate components for infrastructures are decided, it is very important to verify the applicability of each component. For example, our development environment is based on Linux and Linux has complicated dependencies among tools. Without careful consideration, it is likely to require a lot of work to make them support each other. In addition, the work might turn to be useless without active customizations.

Fig. 3 shows how the requirements are connected to our development process and software engineering tools in TRICA. Automatic build and CI are closely related with SCM; SCM is primarily practiced during the implementation, testing, and maintenance phase. Progress and issue sharing are required for effective communication and project management. All of these relations among tools, process, and project members are to be supported within TRICA.

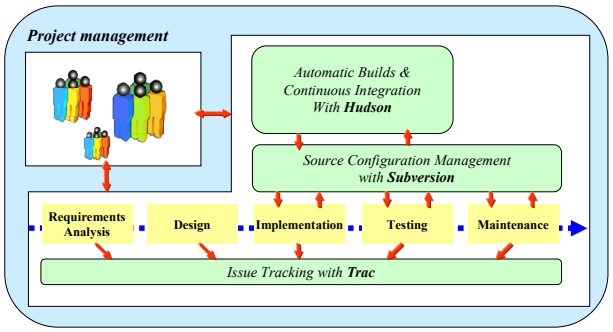

Figure 3. Relations between the requirements and development activities in TRICA.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we showed that TRICA, an open source-based framework consists of Subversion, Trac, and Hudson, can contribute successful software development by decentralizing risks throughout the software development life cycle. TRICA satisfies the requirements on software development infrastructure in our project as following:

- Regular integration helps maintain the integrity of most recent source code and avoid potential risks from big-bang integration.
- It enforces integration process by executing a series of build process automatically and repeatedly.
- It combines integration results with SCM and issue tracking tools. The combined information tightly supports developers to resolve various problems.
- It maintains the agreement between the design documents and the implemented code.
- Based on TRICA, project members can monitor and share issues effectively. This helps us reduce collaboration problems between different organizations.

We firmly believe that TRICA will contribute the success of our project.

TRICA as well as our project is in the course of development, especially the test framework as a part of CI. Test framework for integration test is essential for the success of the project. The test framework will practically link the integration test process into TRICA. Another thread of future work is to enhance TRICA upon the requests from project members such as automatic generation of issues when critical events occurred during the development.

REFERENCES

[1] T. Bruckhaus, N.H. Madhavii, I. Janssen, and J. Henshaw, "The Impact of Tools on Software Productivity", IEEE Software, vol. 13, no.5, Sep. 1996, pp. 29-38, doi:10.1109/ 52.536456.

[2] A. Stellman and J. Greene, Applied Software Project Management, O'Relly Media, California, USA, 2005.

[3] M.J. Harrold, and M.L. Soffa, "Selecting and Using Data for Integration Testing", IEEE Software, vol. 8, no. 2, Mar. 1991, pp. 58-65, doi: 10.1109/ 52.73750

[4] W.T. Tsai, X. Bai, R. Paul, W. Shao, and V. Agarwal, "End-to-end Integration Testing Design", proc. IEEE Intl. Computer Software and Applications Conference (COMPSAC 01), Oct. 2001, pp. 166-171, doi: 10.1109/CMPSAC.2001.960613.

[5] Comparison of Revision Control Software, Online: http://en.wikipedia.org/wiki/Comparison_of_revision_control_software , Retrieved August 26, 2009.

[6] Forrester Research. "The Forrester Wave: Software Change and Configuration Management, Q2 2007", Online: http://www.collab.net/forrester_wave_report/index.html .

[7] Comparison of Issue Tracking Systems. Online: http://en.wikipedia.org/wiki/Comparison_of_issue_tracking_systems, Retrieved August 26, 2009.

[8] M. Flower and M. Foemmel, "Continuous Integration", Online: http://martinfowler.com/articles/continuousIntegration.html.

[9] P.M. Duvall, Continuous Integration: Improving Soft-ware Quality and Reducing Risk, Addpison-Wesley, Boston, USA, 2007.

[10] C. U. Ciborra, "The Platform Organization: Recombining Strategies, Structures, and Surprises", Organization Science, vol. 7, no. 2, Mar-Apr. 1996, pp. 103-118.

[11] Christof Ebert, "Open Source Software in Industry," IEEE Software, vol. 25, no. 3, pp. 52-53, May/June 2008, doi:10.1109/MS.2008.67