Build System Maintenance[*]

Shane McIntosh
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
mcintosh@cs.queensu.ca

ABSTRACT

The build system, i.e., the infrastructure that converts source code into deliverables, plays a critical role in the development of a software project. For example, developers rely upon the build system to test and run their source code changes. Without a working build system, development progress grinds to a halt, as the source code is rendered useless. Based on experiences reported by developers, we conjecture that build maintenance for large software systems is considerable, yet this maintenance is not well understood. A firm understanding of build maintenance is essential for project managers to allocate personnel and resources to build maintenance tasks effectively, and reduce the build maintenance overhead on regular development tasks, such as fixing defects and adding new features. In our work, we empirically study build maintenance in one proprietary and nine open source projects of different sizes and domain. Our case studies thus far show that: (1) similar to Lehman's first law of software evolution, build system specifications tend to grow unless effort is invested into restructuring them, (2) the build system accounts for up to 31% of the code files in a project, and (3) up to 27% of development tasks that change the source code also require build maintenance. Currently, we are working on identifying concrete measures that projects can take to reduce the build maintenance overhead.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming environments—*Programmer workbench*; D.2.9 [**Software Engineering**]: Management—*Productivity, Programming teams, Software configuration management*

General Terms

Management, Measurement

Keywords

Empirical software engineering, build systems, mining software repositories

[*]This work is a joint effort with Dr. Bram Adams and Dr. Ahmed E. Hassan of Queen's University.

1. INTRODUCTION

Software build systems are responsible for automatically transforming the source code of a software project into a collection of deliverables, such as executables and development libraries. A build process may involve hundreds of command invocations that must be executed in a specific order to produce a set of deliverables swiftly and correctly.

Most of the stakeholders in the software development process directly or indirectly interact with the build system on a daily basis. Developers constantly interact with the build system to produce testable artifacts after completing a source code change. Software testers rely on the build system to be able to test the developer changes for regressions in the software behaviour. Project managers use the build system to generate releases of the software system for delivery to customers.

We conjecture that, similar to source code, the build system itself requires substantial maintenance, based on the following three supporting examples: (1) Through a developer survey, Kumfert *et al.* estimate that developers spend 12% of their time maintaining the build system rather than fixing defects and adding features [5]; (2) Adams *et al.* find that the Linux build engineers spent numerous releases evolving the core build machinery of the Linux kernel to simplify the integration of new code by contributors [1]; and (3) The KDE 3 project's build system was such a burden to maintain that it limited the productivity of KDE developers, and even warranted migration to newer build technologies. The migration required a substantial investment of developer time and effort, as the existing build infrastructure had to be reimplemented [9].

Despite its critical role and non-trivial maintenance, the build system is often disregarded by researchers [10] and project managers [5]. Without a strong understanding of how build systems evolve, resources cannot be properly allocated, and software releases may be delivered late and overbudget. For example, Firefox 3.0 was delayed by a build defect that prevented users from accessing the address and search bars in a networked environment [11]. The root cause was uncovered four months after the defect was opened and involved the linking of an incorrect version of the SQLite library with the Firefox product during the build process.

Our main research goal is to identify concrete measures that projects can take to reduce the overhead of build maintenance (Section 4). To achieve our goal, we first need to perform empirical studies to grasp the scale and characteristics of build maintenance (Sections 2 and 3).

Table 1: Listing of the studied projects. Asterisks (*) denote previously used build technologies.

	ArgoUML	Hibernate	Eclipse	Jazz	GCC	Git	Linux	Mozilla	PLplot	PostgreSQL
Timespan	'98-'09	'01-'07	'01-'10	'07-'08	'88-'05	'05-'09	'05-'10	'98-'10	'92-'09	'96-'09
Program lang.	Java	Java	Java	Java	C	C	C	C	C	C
Build techs.	Make* ANT	ANT* Maven	PDE	PDE	Autotools	Make Autoconf	Make KConfig	Make Autoconf	Make* Autotools* CMake	Autotools
# Build Files (BF)	614	211	483	5,967	1,719	43	3,726	10,709	652	771
# Prod Files	7,116	9,272	2,391	45,275	14,181	743	42,912	43,952	659	2,683
# Test Files	891	7,426	1,211	14,738	21,109	824	340	30,835	791	1,377
Total (TS)	8,007	16,698	3,602	60,013	35,290	1,567	43,252	74,787	1,450	4,060
$\frac{\text{BF}}{\text{BF+TS}}$	7%	1%	12%	10%	5%	3%	8%	12%	31%	16%

2. UNIQUENESS OF APPROACH

We first mine the Version Control Systems (VCS) of ten projects to classify all files that existed in an analyzed timespan as either a build, test, or source code file (Table 1). We then analyze this data at two levels of granularity: (1) per revision, and (2) per group of revisions related to a common work item (as recorded in an Issue Tracking System).

Using the classified revision and work item data, we measure how strong the relationships between the build, test, and source code components are using metrics such as logical coupling [4] (Table 2). A Src $\Rightarrow$ Bld coupling of 0.03 indicates that the 3% of source code changes require accompanying build maintenance.

At each granularity, an author can be labeled as a build, test, or source code developer. We assume that developers who produce at least one source code revision are source code developers, since source code development is the main focus of a development team. We only label authors as build developers if their personal Src $\Rightarrow$ Bld coupling is greater than or equal to the project Src $\Rightarrow$ Bld coupling. Similarly, we only label authors as test developers if their personal Src $\Rightarrow$ Test coupling is greater than or equal to the project Src $\Rightarrow$ Test coupling. We measure the logical coupling between author labels to study the distribution of build maintenance work across developers.

3. EMPIRICAL STUDY

We include a synopsis of our results. We present the related work as it motivates five of our research questions. Full details are provided in our publications [6, 7].

RQ1) How large is a typical build system?
Motivation –Robles *et al.* find that the KDE build system is made up of some 39,337 files (9% of all files) [10]. To gain a perspective on the amount of development activity associated with build systems, we want to know if this percentage is consistently across other software projects.

Results – As shown in Table 1, the build system accounts for up to 31% of all project files, with a median of 9%, which complements the earlier results for KDE [10].

RQ2) How much does the build churn?
Motivation – Prior studies have found that frequently changing source code, i.e., code with high churn, has a higher defect density [8]. We want to measure the churn rate in the build system to gain insight into how susceptible the build system is to defects. We compare the churn rate to that of the source code.

Table 2: Logical coupling of source and build code.

		Eclipse	Jazz	Mozilla
Revisions	Src $\Rightarrow$ Build	0.03	0.04	0.08
	Test $\Rightarrow$ Build	0.03	0.07	0.16
Work Items	Src $\Rightarrow$ Build	0.16	0.04	0.27
	Test $\Rightarrow$ Build	0.20	0.08	0.44

Results – Since the build system is a much smaller component than the source code (RQ1), we normalize each component by the component size to ensure that the comparison is fair. When the normalized churn rate of the build system is compared to that of the source code, we find that the two are very similar, at most differing by 7%. Furthermore, build system changes induce more relative churn than source code changes do.

RQ3) How do Java build systems evolve?
Motivation – Initial findings suggest that build maintenance in C projects, e.g., Linux [1] and Amd [13], is difficult since they use arcane technology and must manually track dependencies among artifacts. The Java compiler provides features like automatic dependency resolution [3] that should reduce build maintenance. We want to find out whether Java and C build systems evolve differently.

Results – Similar to C build systems [1, 13], we find that the build systems for Java projects tend to grow in size and complexity from release to release unless explicit effort is invested to restructure them [6]. This indicates that Java build systems require continual maintenance, which project managers should explicitly account for, regardless of the build technology used.

RQ4) How much developer overhead is created by the build?
Motivation – Based on a developer survey, Kumfert *et al.* estimate that developers spend 12% of their time on build maintenance [5]. We want to verify this result empirically.

Results – As shown in Table 2, source and test code revisions rarely contain build changes, however source- and test-changing work items in Mozilla often require build changes (27% and 44% respectively).

RQ5) How is the build maintenance process managed?
Motivation – Since build systems have high churn (RQ2), members of the development team must be making changes to the build. Yet, it is not clear what proportion of the development team is responsible for these changes. We want

to analyze the different ways in which projects allocate personnel to the build system.

Results – We observe two build ownership styles in the analyzed systems: (1) Concentrated: The Linux and Git projects have 25% and 22% coupling between source code and build authors, i.e., given that an author produces source code changes, it is relatively rare that the author also produce a considerable number of build code changes. In both Linux and Git, only 5% of all developers are responsible for writing 80% of the build changes. (2) Dispersed: The Jazz project has a high coupling between source code and build authors (79%) and requires 34% of the developers to account for 80% of the build changes.

4. REDUCING BUILD MAINTENANCE OVERHEAD

The groundwork that we have established in Sections 2 and 3 enables us to identify concrete measures that projects can take to reduce the build maintenance overhead, such as: (1) build resolution prediction, (2) build maintenance recommendation, and (3) build ownership assessments. We briefly discuss each measure below.

Build resolution prediction – Build-related defects such as the Firefox 3.0 one [11] are difficult to resolve. In addition, we have found that 27% of all source code-related work items require accompanying build changes in the Mozilla project (RQ4). To better understand whether build maintenance slows down development, we are performing an empirical study of the impact that build maintenance has on the resolution time of developer work items, i.e., the elapsed time between developer investigation and resolution delivery.

Build maintenance recommendation – Developers often struggle with code that they are not familiar with [2]. As the same holds for build code. Projects such as Linux [1] and Perl [12] have dedicated build teams. However, since build and source code tend to *co-evolve* [1, 6], i.e., changes to the source code often require changes to the build system, and vice versa, a novice developer may easily introduce a source code change, unaware that build maintenance is required. If the build system is not changed when a change is required, the source code may not compile or may produce incorrect deliverables. Hence, we are currently working on a recommendation system to assist developers with identifying code changes that require build maintenance.

Build ownership assessment – We find that the studied projects adopt either a concentrated or dispersed build ownership style (RQ5). However, we did not have data to establish whether either style performed better or worse than the other. Hence, we are in the process of qualitatively and quantitatively studying the impact that the different build ownership styles have on development progress in other C and Java systems. This will allow us to identify the build ownership style of those projects, and to recommend best practices for build maintenance.

5. CONCLUSIONS

According to our findings, project managers can anticipate that the build system requires considerable maintenance (RQ1-RQ4). We find that 4-16% of work items in the studied Java projects require build maintenance, and 27% for the studied C project. Development teams adopt dispersed or concentrated styles for coping with the build maintenance process (RQ5). We are currently leveraging our empirical findings to produce tools that practitioners can use to reduce the build maintenance overhead.

REFERENCES

[1] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter. The evolution of the linux build system. *Electronic Communications of the ECEASST*, 8, 2008.

[2] G. Antoniol and Y. G. Guéhéneuc. Feature Identification: A Novel Approach and a Case Study. In *Proc. of the 21st Int'l Conf. on Software Maintenance (ICSM)*, pages 357–366. IEEE Computer Society.

[3] M. Dmitriev. Language-Specific Make Technology for the Java Programming Language. In *Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. ACM, 2002.

[4] H. Gall, K. Hajek, and M. Jazayeri. Detection of Logical Coupling Based on Product Release History. In *Proc. of the 14th Int'l Conf. on Software Maintenance (ICSM)*, pages 190–198, Washington, DC, USA, 1998. IEEE Computer Society.

[5] G. Kumfert and T. Epperly. Software in the DOE: The Hidden Overhead of "The Build". Technical Report UCRL-ID-147343, Lawrence Livermore National Laboratory, CA, USA, 2002.

[6] S. McIntosh, B. Adams, and A. E. Hassan. The Evolution of ANT Build Systems. In *Proc. of the 7th working conf. on Mining Software Repositories (MSR)*, pages 42–51. IEEE Computer Society, 2010.

[7] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*. ACM, 2011.

[8] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proc. of the 27th Int'l Conf. on Software Engineering (ICSE)*, pages 284–292, New York, NY, USA, 2005. ACM.

[9] A. Neundorf. Why the KDE project switched to CMake – and how (continued). `http://lwn.net/Articles/188693/`, 2010. last viewed: 06-Mar-2010.

[10] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo. Beyond Source Code: The Importance of Other Artifacts in Software Development (A Case Study). *Journal of Systems and Software (JSS)*, 79(9):1233–1248, 2006.

[11] T. Steiner. mozStorage chokes on databases over AFP. `https://bugzilla.mozilla.org/show_bug.cgi?id=417037`. Last viewed: 18-Aug-2010.

[12] Q. Tu and M. Godfrey. The build-time software architecture view. In *Proc. of Int'l Conf. on Software Maintenance (ICSM)*, pages 398–407. IEEE Computer Society, 2002.

[13] E. Zadok. Overhauling Amd for the '00s: A Case Study of GNU Autotools. In *Proc. of the FREENIX Track on the USENIX Technical Conf.*, pages 287–297, Berkeley (CA, USA), 2002. USENIX Association.