A Qualitative Analysis of Software Build System Changes and Build Ownership Styles

Mini Shridhar[1,2], Bram Adams[1], Foutse Khomh[2]
[1]*MCIS* – [2]*SWAT, Polytechnique Montréal, Québec, Canada*
{*mini-maria.shridhar, bram.adams, foutse.khomh*}*@polymtl.ca*

ABSTRACT

Context: Recent empirical studies have shown quantitatively how software build systems, which are responsible for converting software artifacts into an installable deliverable for the end user, induce considerable overhead on software developers, taking away their focus from actual development.
Goal: Little, however, is known of what are the typical types of changes that these developers need to make to build systems, the characteristics of these changes and whether developers work on these changes by themselves, or are coordinated by build experts.
Method: This paper qualitatively investigates the build commit history of 18 open-source projects from the Apache and Eclipse eco-systems, over a period of fourteen months, using manual tagging and classification of change types and build system ownership styles.
Results: "Corrective", "Adaptive" and "New Functionality" build changes introduce considerably higher churn and are more invasive, while many changes are identified by accident during regular development. Having dedicated build experts allows software projects to make more invasive "Adaptive" changes.
Conclusions: Build system studies need to take into account the type of build change, since not all build changes are equal.

Categories and Subject Descriptors

D.2.9 [**Management**]: Software configuration management

General Terms

Documentation, Experimentation, Measurement

Keywords

Build System, Qualitative Analysis, Empirical Study, Software Evolution

1. INTRODUCTION

Build systems turn source code into executable programs by orchestrating the execution of compilers, code generators and other compilation tools. For this, the build system takes as input the source code artifacts, the compilation tools and dependencies between the software artifacts, then performs the required actions to produce any required intermediate artifact until the final project deliverable can be generated.

Build systems are the hub of the software development process. Researchers have found that one in every four source code changes, and almost one in every two test case changes, require changes to build files [13]. Developers typically run builds several times a day, to check the impact of their code changes on the software system. Similarly, test engineers also run builds several times a day, to check the impact of the developers' code changes on the test suites. The build system also includes the critical task of packaging the software deliverables, ensuring packaging of components, dependencies, data files and documentation in the right order into the final software product to be delivered to the end-user. All these activities are executed after each code commit by continuous integration systems, driven by the build system. In other words, without a robust build system, many development tasks become tedious, complex and slow, thereby impacting the entire software team.

Despite the critical role that build systems play throughout the software development process, very little is known about their maintenance. Previous empirical research has shown how build system maintenance imposes considerable overhead on the software development process [10, 2, 12, 13]. In an attempt to reduce this maintenance overhead, many software projects like KDE [16] and MySQL [6] even have switched to newer build technologies [21]. These prior studies considered each build change or error as having the same complexity or priority, whereas similar studies on source code changes have shown how the size and impact of changes can differ significantly, and that it is important to consider who is making those changes [18]. However, at present, no concrete advice can be given to practitioners as to what kind of changes to avoid or how to organize their build team [17].

Similar to categorization of source code changes [11, 7], we believe that a qualitative categorization of individual changes to build system code can provide practitioners with *tangible evidence* of potential areas and means through which build code can become non-maintainable and complex. However, since there is no existing categorization of typical build code changes and who makes them (build expert or developer), a first step towards such tangible evidence is to iden-

ESEM '14 September 18–19, 2014, Torino, Italy.

tify the different categories of build changes, analyze the typical invasiveness and size of the changes in each category, and study which changes are more commonly done by developers or by build experts. Therefore, this paper manually investigates each build system-related commit of 13 Eclipse and 5 Apache projects (of varying sizes, histories and build ownership styles) from the 1st of November 2012 until the 7th of January 2014, providing the following main contributions:

- a categorization of the types of changes made to build systems;
- an analysis of the amount of change (churn) and invasiveness introduced by these change categories;
- an evaluation of the characteristics of these change categories, taking into account the build system ownership styles of the studied projects.

The remainder of this paper is organized as follows. Section 2 discusses background and how our work builds on prior work. Section 3 describes research questions that we address, our study setup and approach, while Section 4 discusses the results and findings of our study. Section 5 discusses the threats to validity, while Section 6 concludes the paper.

2. BACKGROUND AND RELATED WORK

A build system typically performs two major tasks. First, it enables configuration of the features that one wants to include in the generated project deliverable, as well as of the specific version of compilers and libraries to use. Second, the configured tools are used to compile and construct the configured features and physically generate the project deliverable. Configuration options and constraints typically are expressed using dedicated tools like autoconf or Kconfig [5], while the commands and dependencies used for construction are specified using well-known tools like GNU Make (C/C++), Apache ANT/Maven (Java) or CMake. A typical large software system like the Linux kernel contains hundreds of configuration and construction scripts, amounting to thousands of lines of build-related code [1]. One mistake can cause a build to break, potentially grinding the development processes to a halt, since not just developers but also testers and release engineers rely on the build system [9].

Given this risk, Kumfert et al. [10] argue that the need to keep build system code (i.e., specifications of configuration and construction) synchronized with source code imposes significant overhead on the development process. They provide initial empirical indications that build system maintenance is a hidden cost, which is usually not accounted for in a project's budget. McIntosh et al. [13] confirm these findings through empirical analysis on ten large, long-lived systems. They observed how one out of four source code changes and two out of five test code changes require changes to the build system, and how, despite their smaller size compared to the source code, build systems change considerably more often.

Another line of research has been measuring the evolution of build systems across successive releases. McIntosh et al. [12] show that the complexity of ANT build systems grows over time, and co-evolves with source code complexity. This corroborates earlier findings by Adams et al. [1] on the GNU Make-based Linux build system. Apart from quantitative measures, the latter study and earlier ones on (amongst others) a closed source system and Quake 3 used the MAKAO tool to visualize, query and compare the execution of Make-based build systems [2, 1]. Tu et al. [23] also used a dynamic analysis-based approach to understand the complexity of build systems.

What can build system developers do to reduce this complexity? Smith's practitioner-oriented book [20] provides a list of recommendations to reduce complexity, focusing especially on simplifying the build system code and keeping build dependencies synchronized with source code dependencies. Tamrawi et al. [22] introduce SYMake, which statically analyzes Makefiles to enable build system refactorings like renaming of variables or to detect build system smells. MAKAO [2] instead uses dynamic analysis to identify smells and propose refactorings. Some build system architectures like recursive make also have been found to cause errors and inconsistent build results [14]. Finally, Neitsch et al. [15] and Seo et al. [19] qualitatively studied build system execution problems, proposing catalogues of (anti-)patterns and possible solutions.

Our work builds on this prior work by performing a first qualitative study of the kinds of changes practitioners make to a build system. Whereas existing studies treat each change as equal, we measure the size and invasiveness of each category of changes to understand which changes are larger (and possibly riskier) in nature than others. Furthermore, building on the initial work of McIntosh et al. [13] and Phillips et al. [17], we analyze which changes are more commonly performed by build experts (concentrated build ownership [13]) and which ones by developers (dispersed build ownership [13]). This helps practitioners to understand what kinds of changes (not) to expect when choosing one kind of authorship or the other.

Our categorization of changes is based on that of Lientz et al. for source code maintenance [11]. They proposed four main categories of changes, i.e., corrective (fixing failures), adaptive (making the system work in newer environments), perfective (improving inefficiencies) and preventive (improving future maintainability) changes. Later on, Hindle et al. [8] and Amor et al. [3] extended this classification with more detailed subcategories. However, since those subcategories are specific to source code changes, we started from Lientz et al.'s original classification, extending it where needed.

3. CASE STUDY SETUP

This paper addresses the following three research questions:

RQ1 What are the typical types of build changes that are performed in the studied software projects?

RQ2 How invasive and large are these build changes?

RQ3 How does the build ownership style of the project affect the characteristics of these change types?

Since there is no existing benchmark for defining build change categories, we manually analyzed the commit messages and changed build code of build system-related changes in the version control system. The commit message provides a developer's intention for a change, which, similar to Lientz et al. [11], is the level of granularity our change categories

Table 1: Subject Systems and their characteristics (1st of November 2012 to the 7th of January 2014).

	System	total # commits	#build commits	#build files	ownership style
Apache	Hadoop-Common	2,666	136	57	Collective
	Maven-Plugins	1,035	520	1,766	Dedicated
	ANT	676	24	239	Strong
	Maven-Core	394	129	347	Strong
	Maven-Release	524	33	59	Strong
Eclipse	Equinox-Framework	406	45	113	Collective
	Equinox-P2	365	58	158	Collective
	LinuxTools	1397	162	355	Collective
	PDE	57	23	74	Collective
	BPEL	19	5	76	Dedicated
	Mylyn	64	45	30	Dedicated
	Platform	99	25	59	Dedicated
	Platform-Releng	255	124	76	Dedicated
	AJDT	113	43	58	Strong
	CDT	1,085	50	334	Strong
	JDT-Core	246	41	33	Strong
	Orion-Client	2,403	68	32	Strong
	Orion-Server	963	431	60	Strong

aim at. In other words, we are not interested in syntactical changes like "ANT target is added" or "Maven library dependency is removed", since those are technology-dependent and are just a mechanical means towards a specific intention. It is that intention that we want to classify with our categories. Furthermore, in the few cases where a build-related commit message mentioned a specific bug report that was fixed, we also analyzed the report to further improve our understanding of the change's intention. Given the low number of commits for which this was possible, we additionally manually inspected the actual changes done to the build files and any coupled source files (inside a build commit) to cross-validate the changes' intention.

Below, we explain the data sources that we used, the extraction process of our data, the change categories we based our work on and the characteristics that we measured in the catalogued changes.

3.1 Data Source Selection

For our qualitative study, we selected *thirteen* Eclipse projects and *five* Apache projects. We chose these projects based on the variation in build ownership style, number of (build) commits and number of build files, as is shown in Table 1. A build commit is a commit in which at least one build file has been changed (possibly together with other build or even source code files). Furthermore, having projects from two ecosystems to a large extent reduces bias related to different development guidelines or philosophies, since Apache projects and (especially) Eclipse projects share common guidelines amongst each other. Finally, members of both ecosystems helped us identify the projects with the clearest specified build ownership, as discussed further below.

We mainly analyzed projects with Maven- and ANT-based build systems, since those technologies are the two most popular build languages for Java systems and belong to the top build technologies overall. Many Eclipse projects recently migrated from an ANT-based build system to a Maven-based build system called "Tycho", which is basically a set of Maven plugins and extensions for building Eclipse plugins and OSGi bundles. OSGi bundles are high-level Java components that use their own metadata for expressing dependencies or source folder locations, which overlaps with data found in a regular Maven file.

We studied subprojects of the following Eclipse and Apache projects: **Eclipse AJDT** provides tool support for aspect oriented development in Java. **Eclipse BPEL** manages WS-BPEL 2.0 processes of web services. **Eclipse CDT** provides a fully functional C and C++ Integrated Development Environment. **Eclipse Equinox Framework** is an implementation of the OSGi R4 core framework specification. **Eclipse Equinox p2** is a sub-project of Equinox that focuses on provisioning technology for OSGi-based applications. **Eclipse JDT Core** is the Java development infrastructure of the Java IDE. **Eclipse LinuxTools** extends the CDT project even further with C and C++ IDE functionality. **Eclipse Mylyn** is the task and application lifecycle management framework for Eclipse (ALM). **Eclipse Orion** is a browser-based open tool integration platform. **Eclipse PDE** provides tools to manage Eclipse plug-ins and their deployment. **Eclipse Platform** defines the common infrastructure below all Eclipse plugins and RCP applications. **Eclipse Platform RelEng** provides release engineering services for the Eclipse Project team. **Apache ANT** and **Apache Maven** are the two most popular build system technologies for Java systems. **Apache Hadoop-Common** is the set of common utilities that supports other projects based on the Apache Hadoop distributed computing framework.

3.2 Data Extraction

Once data sources were chosen, we needed to identify the build files of each project, then examine the metadata (commit messages, author and committer names) of all commits that touch build files. To get more context about build system changes, we also analyzed all bug reports involving build files as well as the build file changes themselves.

We first extract all commits from the version control systems of the studied projects over a period of fourteen months, from the 1st of November 2012 to the 7th of January 2014. We then used the same semi-automated method as McIntosh et al. [13] to classify build files for each project. More specifically, using regular expressions based on known file names and extensions, we first filtered out test and source code files as well as typical build file names like "build.xml" and "pom.xml". We did not take into consideration OSGi manifest files as build files, since they do not contribute towards actual compilation commands. Afterwards, we double-checked the automated classification results to deal with ambiguous file names, and manually classified the remaining files with less common file extensions.

We then only kept the commits that touch at least one build file, and called those commits "build commits". For each such commit, we automatically extracted the commit log message, commit author and committer, and whether the commit involved only build code changes, source code changes or both. Furthermore, during our manual analysis of the build commit log messages, we also looked for bug identifiers, and extracted the corresponding build-related bug reports from the bug repository of the system.

3.3 Identification of Change Categories

Table 2: Broad classification of build change categories.

Category	Maintenance Class
Adaptive	Change in environment
Corrective	Fix to build code
Perfective	Improvements to build behaviour
Preventive	Improving future maintainability
New Functionality	Addition of new features
Reflective	Side-effect change

To identify the different categories of build changes, we performed an exhaustive, manual analysis of each project's set of build files, build commits, commit comments and bug information in Bugzilla. Similar to other qualitative studies [4], we used a card sort-based approach, which is a lightweight form of grounded theory to derive taxonomies from textual data. Basically, the information of each textual document (commit) is put on a (virtual) card, then cards are analyzed by the first author and clustered together if their content has the same intent. Finally, the identified clusters are validated with the other authors. It is worth mentioning that the first author has 12 years of experience as release and build engineer working at multinationals like Broadcom, Cisco, Nokia, and Motorola, and hence was the ideal candidate for doing the initial categorization.

We made a small adaptation, in that we started with four empty clusters for the four change categories identified by Lientz et al. [11], i.e., "corrective", "adaptive", "perfective" and "preventive" changes (first four entries in Table 2). Often, *keywords* like "fix", "update", "Adding", "Refactoring", "CleanUp", "Adapting", "Correcting", "Maintenance" and "Porting" were helpful in determining the intent (and hence category) of a build commit. If we found a build change that did not fit in any of the pre-defined categories, we added a new category. Frequently recurring terms in the change logs and other text helped establishing a name for such new categories. If we found a build commit that could be in more than one category, we added it into the more "dominant" category. Eventually, two new clusters were identified, i.e., "reflective" and "new functionality".

The resulting set of categories looks like this:

Adaptive changes that adapt to a new (build/deployment) environment or to new functionality in the source files. These can include, but are not limited to, changes in the method of packaging a build, porting builds to a new platform, or including new source code files in the build.

Corrective changes that fix any kind of defect in the build code.

Perfective changes that improve existing design or functionality of builds and build system. Examples include changes to enhance build performance and build efficiency.

Preventive changes made exclusively to refactor build code, with the aim of improving readability, cleaning up the build code and removing existing build code smells. In other words, preventive changes aim to improve future maintainability of the build system.

New Functionality changes are made to meet new build functionality requirements, such as addition of a new target to the build, without corresponding source code changes (otherwise the change would be adaptive). An example would be adding version numbering of deliverables or adding functionality that not just builds but also packages and deploys a project deliverable.

Reflective changes performed on build code to reflect a change (i.e., restructuring or refactoring) that happened on the source code. Contrary to the adaptive change type, where a build needs to react to *new functionality* in the source code, reflective change encompasses changes where an *architectural restructuring, design refactoring, or bug fix* in the source code propagates to the corresponding build code or at least induces the developer to look at the corresponding build code and change it.

3.4 Analysis of Change Categories

In order to quantitatively analyze the identified categories of build commits, we computed the following metrics:

Build Commit Density is the percentage of all build commits of a project that belongs to a specific category. This allows to identify the most frequent change category of a project.

Build Churn is the average amount of build churn per file changed by the commits of a category. Churn is the sum of the number of added and removed lines of code in a build commit. We normalize this sum by the number of files of a project in order to compare the resulting number across projects.

Invasiveness is the median number of unique build files modified by the build commits in a category. The higher, the more invasive (and hence risky) build commits are.

Apart from the churn and invasiveness of build commits, we also want to learn who makes build changes. McIntosh et al. identified two major build ownership styles, i.e., concentrated (dedicated build expert) and dispersed (no expert) [13]. Through contacts with Eclipse developers, we obtained information about the build ownership styles of projects related to the Eclipse Platform project. Further analysis showed that these build experts could in fact be identified by considering the top contributors to the build system files (in the version control system) and in the build system-related mailing list topics. Hence, for the Apache projects, we used these approximations to determine the build ownership of the extracted build changes.

Based on this analysis and Martin Fowler's blog on "Code Ownership Styles"[1], we refined McIntosch et al.'s two ownership styles into 3 distinct ownership styles:

Dedicated Ownership where a software team has a dedicated build team or build expert who manages and "owns" the build system. These build experts are the only ones making changes to the build system (this corresponds to McIntosh et al.'s concentrated ownership).

Strong Ownership where one or more developers predominantly make changes to build code and others seek

[1]http://martinfowler.com/bliki/CodeOwnership.html

Table 3: Build change categories with prototypical example.

Adaptive	*'JVM used to run maven must now be Java 1.6 or newer - can still compile and run tests with Old JVM via toolchains'*
Corrective	*'Our plugin artifacts don't seem to have the correct groupId in any of the streams. The general naming convention is the groupID should be the first 3 segments of the plugin'*
Perfective	*'I would fully +1 any breaking change if it means moving more towards best practices'*
Preventive	*'I think we should try another effort to replace this hard-coded value in each pom with a variable and make the pom a "real" file... And while you are in that neighborhood, please consider removing all those obsolete modules'*
New Funct.	*'Add functionality for Hudson builds'*
Reflective	*'This requires a change in each pom.xml file in your repository: for code bundles and tests the change is the same'*

their approval to make changes to build code, but no one fully "owns" the build system by himself.

Collective Ownership where any developer can make changes to the build system, and no one "owns" the build system.

Once we established the build ownership styles in the studied projects, we studied the effect of these different build system ownership styles on the change categories, by comparing the churn and invasiveness metrics amongst authorship styles.

4. CASE STUDY RESULTS

For each research question, we first provide a motivation, followed by the approach and a discussion of the results.

RQ1. What are the typical types of build changes that are performed in the studied software projects?

Motivation. Build system maintenance has a different impact on different kinds of build system changes. For example, small "Adaptive" updates of the copyright year intuitively seem less harmful or difficult to manage than large "Preventive" refactorings or additions of Ant or Maven files. Understanding the major change categories in a project would show us the focus of build system maintenance and the associated effort and risk.

Approach. We used the build commit density to understand the popularity of the six change categories of Table 2 in each project, and also provide examples of discussions in bug reports on each category (Table 3). Finally, Table 4 shows the most common types of changes that we identified in each change category during our qualitative analysis.

Findings.

"Corrective" and "Adaptive" changes are the most popular, while "Preventive" changes are relatively rare. The bold build commit density values in Table 5 indeed show that in both the Eclipse and Apache projects, changes from the "Corrective" category occur the most, followed by "Adaptive" changes and (to a lesser degree) "New Functionality" changes. Even when "Corrective" changes are not the most popular category, it does not lag behind much, except for the "Adaptive" changes of the Eclipse BPEL, Mylyn and (especially) Eclipse Orion-Server projects, with differences of up to 85%. Conversely, "Preventive", "Reflective" and (to some degree) "Perfective" changes are far less common. Indeed, a Kruskal-Wallis non-parametric omnibus test, followed by post-hoc tests (with α value of 0.05) showed how "Perfective" and "Preventive" changes have statistically significantly lower commit density values than the other change categories.

The prototypical examples in Table 3 and the common types of changes inside each category in Table 4 provide more insight into the kinds of build changes encountered. Now, we discuss some of the qualitative findings found for each studied eco-system:

1. Apache ANT has many build-only commits involving changes to build files like build.xml and pom.xml. Similarly, in *Apache Maven-Core* and *Apache-Maven-Release*, we find that most build commits predominantly involve build files only, except in the "Corrective" category (Apache-Maven-Release), where commits change other file types as well in considerable numbers, along with build files.

On the other hand, changes to ivy.xml and other Apache Ivy files (support library for Ant build systems to resolve dependencies on 3rd party Java libraries, like Maven does), in any change category almost always has accompanying changes to other types of files like source files, test case files and so on. This was found to be especially true for commits in the "Adaptive", "New Functionality" and "Perfective" categories, but not necessarily "Reflective". In other words, these library dependencies rather are updated to adapt to a new environment, add new build/code features and optimize the build, rather than in response to restructuring or refactoring of the source code.

In *Apache-Hadoop-Common*, all change categories, except for "Perfective", had build commits that changed other types of files too, in addition to build files. We attribute the differences with Apache ANT, Apache-Maven and sub-projects to the fact that the latter projects are themselves build tools and hence have a large number of build files and many commits changing build files. This is also explains the relatively high percentage if "New Functionality" build changes compared to the other projects. Apache-Hadoop-Common, on the other hand, has fewer build files and changes to build files, in comparison.

2. Apache Maven-Plugins exhibits a large percentage of changes in the "New Functionality", "Adaptive" and "Corrective" categories. Interestingly, those changes seem to belong especially to release preparation commits, i.e., commits preparing the next release. As such, these commits are reverted (i.e., undone) quite often, to include more changes, until developers are satisfied that they have all the required changes into the release and are ready for a fresh development iteration. This seems to suggest (1) that some kind of build tests would be interesting to know for sure that a build change is complete, and/or (2) that build changes might have (possibly hidden) dependencies that makes it difficult to get things right the first time. Change impact analysis techniques could possibly play a role here.

3. Eclipse AJDT does not change many build files, in nei-

Table 4: Most common types of build changes in each change category.

Adaptive	Corrective	Perfective	Preventive	New Functionality	Reflective
Update Plugin Versions	Fix for compile errors	Shorten Build Times	Remove unused/redundant dependencies	Add new Build Profile	Removal obsolete source bundles from compiling
Addition/Removal of Plugin Dependencies	Fix for wrong paths	Improve Build Performance	Remove dead build code	Add new Targets	Removal unused jars (components) from compiling
Addition of new bundles to compile	Add missing includes	Flag "circular dependencies" as errors	Remove hard-coded values	Add new Goals/-Tasks	Removal of unused configurations
Addition of properties/qualifiers	Fix for Group/Artifact IDs	Restrict warnings thrown in build log	Remove duplicated build code	Add new Module Build	Removal of unused include build dependencies
Removal of inner jars signing	Fix for version mismatch	Make build output less noisy	Improve Existing Build System Design		
Change publish/archive methods	Fix for Copyrights				
Reorder update sites	Fix for typos				

Table 5: Values of build commit density, with the highest value for each project bolded.

		Adaptive	Corrective	Perfective	Preventive	New Functionality	Reflective	Ownership Style
Apache	Hadoop-Common	26.47	**47.06**	4.41	1.47	18.38	2.21	Collective
Apache	Maven-Plugins	29.81	26.35	1.92	10	**31.92**	0	Dedicated
Apache	ANT	12.5	**45.83**	8.33	8.33	25	0	Strong
Apache	Maven-Core	**31.01**	26.36	11.63	2.33	26.36	2.33	Strong
Apache	Maven-Release	**39.39**	36.36	0	0	24.24	0	Strong
Eclipse	Equinox-Framework	26.67	**57.78**	0	0	13.33	2.22	Collective
Eclipse	Equinox-P2	34.48	**48.28**	1.72	1.72	8.62	5.17	Collective
Eclipse	LinuxTools	23.46	**29.63**	15.43	6.79	13.58	11.11	Collective
Eclipse	PDE	**47.83**	**47.83**	0	4.35	0	0	Collective
Eclipse	BPEL	**60**	0	0	0	40	0	Dedicated
Eclipse	Mylyn	**51.11**	17.78	0	6.67	22.22	2.22	Dedicated
Eclipse	Platform	32	**56**	8	4	0	0	Dedicated
Eclipse	Platform-Releng	24.19	**52.42**	9.68	5.65	2.42	5.65	Dedicated
Eclipse	AJDT	13.95	39.53	0	2.33	**44.19**	0	Strong
Eclipse	CDT	30	**38**	0	0	32	0	Strong
Eclipse	JDT-Core	24.39	**73.17**	0	0	2.44	0	Strong
Eclipse	Orion-Client	7.35	**44.12**	7.35	1.47	30.88	8.82	Strong
Eclipse	Orion-Server	**89.33**	4.64	0.46	0	4.18	1.39	Strong

ther of the categories. During the studied period, there was substantial source code churn, but there were not as many changes to build files as expected based on the other studied projects. This can be attributed to the fact that most of the new Eclipse features added to AJDT did not require more than a new profile to the OSGi manifest files to include the feature in the build system. Similarly, in *Eclipse BPEL*, we noted that changes to build files are few and, if any, consist more of version number updates and such. The same holds for *Eclipse-Equinox-Framework* and *Eclipse-Linuxtools*. The latter is a special case, since the number of changes in the build commits is actually high, yet it only represents a tiny portion of the overall changes, since this project integrates, at set times, 3rd party code from various open source debuggers and other tools into its code base. Those integrated changes outnumber the build changes. *Eclipse-PDE* also has a small proportion of build changes, except for the "Corrective" and "Adaptive" categories of changes, whose build commits only change build files.

4. Eclipse CDT has a comparable number of changes to build files across the three major change categories. It is interesting to note that in this project, most build commits fall in the "Corrective" category, since many build changes consist of very simple fixes for typos and relative paths.

5. The other projects showed less clear trends.

"Corrective" changes to build files, especially those that fix compilation errors, are done in 2-3 commits, until the issue is fully fixed. This again hints at the need for build tests or means to identify dependencies between different build errors with a common cause. In most projects, build changes (in general) appear together in batches of commits, changing both source and build files.

Developers frequently happen upon maintenance related issues in build code, by accident, when doing other changes. We repeatedly spotted this phenomenon in the analyzed commit messages and the bug discussions.

Some examples of how these maintenance issues are discovered include sudden, perceivable large build times, or a bug introduced into build code a long time ago, which is stumbled upon while doing some other type of build change. When such symptoms are discovered and fixed ("Corrective" change), they sometimes lead to on-the-spot "Preventive" and "Perfective" types of changes. Developers somehow prefer to immediately deal with build maintenance issues rather than reporting them somewhere (e.g., in a bug report). This confirms the fact that we hardly found any bug report dedicated to build system issues for the studied systems.

Build changes can cause source code changes, and vice versa. We found evidence of some build changes that cause source-code changes. Examples of these include cases where developers find that an unused *jar* is still being compiled ("perfective" change). They then trace to the corresponding source code files, then remove the dead code from both the build and source code files. We also found evidence of the inverse case, where developers change build code as a side-effect of a change to source files. Such "reflective" changes were not that common though (as shown in Table 5). A typical example of such a change is when developers refactor source code and find that they have to correspondingly change or correct the build code.

"Corrective" and "Adaptive" changes are the most popular, while "Preventive" changes are rare. Many build issues are identified accidentally while making other changes, with "Corrective" changes being performed in batch until an error is fixed. This could hint at quality assurance issues with the build system.

RQ2. How invasive and large are these build changes?

Motivation. Similar to the motivation of RQ1, the impact of having a small change to a build system is different from that of a large change. However, the kind of change also plays a role, since a small "Corrective" change could be more risky than a large "Reflective" change. Frequency of change is another important factor, with many small build changes potentially more harmful than few large changes. Therefore, here we explore the invasiveness and size (churn) of each change category.

Approach. We study the build churn, invasiveness and commit density metrics of Section 3.4 in each build change category. The commit density numbers were shown in Table 5, whereas churn and invasiveness are shown in Table 6 and Table 7. Our intuition here is that a change category is more "risky" if it involves frequent large changes (high commit density and build churn) or if it makes large changes across many build files (high build churn and invasiveness). We also hypothesize that a lower invasiveness factor can offset a high churn factor and vise versa, in terms of the riskiness of the change category.

Results. The highest churn and most invasive changes belong to the "Adaptive", "New Functionality" and "Corrective" change categories.

Those build commits affect many build files at once, and make mostly changes to build files. For example, the "Adaptive category" of changes has a high invasiveness factor, since this change category usually involves changes to many different build files. This is contrary to the "Corrective" category of changes, where, although the churn factor is high and the invasiveness factor is moderate, the same sets of build files are changed repeatedly (i.e., the fixed errors seem to be focused in a small set of files). If we also consider the popularity of change categories (RQ1), we observe that the three most popular categories are also those that contain the largest and most invasive changes, even though, for half of the projects, the most popular category is not necessarily identical to the category with the most invasive or largest changes.

The popularity of large "Adaptive" changes in certain projects like Eclipse Orion can be attributed to the process of minification of the Orion project, during the period of study. Minification, especially in javascript, is the process of removing all unnecessary characters from source code to make downloading to browsers faster. The minification exercise in Eclipse Orion alos includes minification of Orion builds and Orion Continuous Builds in Hudson, which accounts for a steep churn in the "Adaptive" category of build changes for this project. However, the invasiveness of these minification changes are low to moderate, indicating that builds were constrained to only a limited number of build files (typically 1 for Orion).

The invasiveness factor for the "Perfective", "Preventive" and "Reflective" categories is low, yet the highest for 5 projects. This is confirmed by a Kruskal-Wallis non-parametric omnibus test, followed by post-hoc tests (with α value of 0.05), which showed how "Perfective" and "Preventive" changes have statistically significantly lower build churn values than the other change categories. In most projects, the "Preventive" category of changes only touches build files, without involving co-changing files of other types. Hence, the churn induced by this change category in most cases purely boils down to build file churn. In Eclipse Equinox and Mylyn, we found that the resulting low churn for "Preventive" changes was due to the fact that developers here often discover latent maintenance issues and quickly fix these. Such issues are mostly discovered while doing other changes to either build or source code, and hence the "Preventive" maintenance fix is included in the same commit as the originally intended "Corrective" fix. As such, our categorization does not count this as "Preventive", leading to low churn values. Although the Eclipse Platform's "Preventive" and "Perfective" build churn is not that much higher than that for Equinox or Mylyn, we did find traces there of more dedicated (i.e., non-accidental) maintenance changes and improvements for build files.

The "Reflective" change category induces the least churn and is the least invasive, among all the categories. The number of changes falling in this category are also quite low. A similar Kruskal-Wallis non-parametric omnibus test as used for commit density and build churn showed how "Reflective" changes have statistically significantly lower invasiveness values than the other change categories.

Finally, Eclipse Platform Releng couples a high churn for new functionality with low invasiveness. This is due to the fact that when new builds are added, typically the changes for these are large but do not span across too many build files.

Table 6: Values of build churn, with the highest value for each project bolded.

		Adaptive	Corrective	Perfective	Preventive	New Functionality	Reflective	Ownership Style
Apache	Hadoop-Common	**28.63**	19.60	22.77	0.39	2.18	1.37	Collective
Apache	Maven-Plugins	1.44	**5.86**	0.07	4.85	1.41	0	Dedicated
Apache	ANT	1.19	1.25	0.02	0.45	**2.65**	0	Strong
Apache	Maven-Core	0.95	1.57	0.63	0.06	**2.51**	0.07	Strong
Apache	Maven-Release	2.69	**377.24**	0	0	3.34	0	Strong
Eclipse	Equinox-Framework	4.39	**26.02**	0	0	2.01	0.02	Collective
Eclipse	Equinox-P2	**7.40**	4.28	0.01	0.03	1.46	0.51	Collective
Eclipse	LinuxTools	**4.67**	2.37	1.47	1.41	4.30	2.32	Collective
Eclipse	PDE	**21.81**	1.61	0	0.08	0	0	Collective
Eclipse	BPEL	5.54	0	0	0	**12.47**	0	Dedicated
Eclipse	Mylyn	**26.73**	14.4	0	1.37	20.37	0.13	Dedicated
Eclipse	Platform	5.97	**6.66**	0.17	0.85	0	0	Dedicated
Eclipse	Platform-Releng	12.32	**21.54**	4.34	2.89	1.03	1.62	Dedicated
Eclipse	AJDT	0.60	9.41	0	0.76	**29.97**	0	Strong
Eclipse	CDT	1.07	0.87	0	0	**1.80**	0	Strong
Eclipse	JDT-Core	9.09	**21.03**	0	0	0.45	0	Strong
Eclipse	Orion-Client	2.88	35	7.22	0.81	**95.28**	3	Strong
Eclipse	Orion-Server	**37.8**	3.4	0.7	0	33.1	4.8	Strong

Table 7: Values for invasiveness, with the highest value for each project bolded.

		Adaptive	Corrective	Perfective	Preventive	New Functionality	Reflective	Ownership Style
Apache	Hadoop-Common	5	**6**	4	1	5	2	Collective
Apache	Maven-Plugins	3	4	2	3	**6**	0	Dedicated
Apache	ANT	**25**	6	1	3	**25**	0	Strong
Apache	Maven-Core	1	2	2	1	**13**	1	Strong
Apache	Maven-Release	1	**4**	0	0	3	0	Strong
Eclipse	Equinox-Framework	3	4	0	0	**24**	1	Collective
Eclipse	Equinox-P2	**3**	2	1	1	1	**3**	Collective
Eclipse	LinuxTools	3	**4**	2	3	3	1	Collective
Eclipse	PDE	1	1	0	**2**	0	0	Collective
Eclipse	BPEL	**30**	0	0	0	4	0	Dedicated
Eclipse	Mylyn	**7**	3	0	4	2	1	Dedicated
Eclipse	Platform	**6**	2	1	2	0	0	Dedicated
Eclipse	Platform-Releng	2	2	**3**	2	1	1	Dedicated
Eclipse	AJDT	1	1	0	**3**	1	0	Strong
Eclipse	CDT	1	1	0	0	**3**	0	Strong
Eclipse	JDT-Core	**7**	3	0	0	2	0	Strong
Eclipse	Orion-Client	1	2	2	1	**3**	1	Strong
Eclipse	Orion-Server	1	1	**2**	0	**2**	**2**	Strong

"Adaptive", "New Functionality" and "Corrective" changes induce more churn and are more invasive. The other categories are less invasive, partly because they are more focused, and partly because they are performed together with "Corrective" or other changes.

RQ3. How does the build ownership style of the project affect the characteristics of these change types?

Motivation. Finally, we want to study the impact of a particular kind of build system ownership on the kind of build system changes performed. That is, we want to know for example if "Corrective" changes happen frequently in collectively owned systems (which might be risky since there is less quality control), or if they only enter into the version control system via a dedicated owner.

Approach. We study the effect of the build system ownership style on the characteristics of the build changes for each category. Build changes are characterised using the metrics from Section 3.4, i.e., commit density, churn and invasiveness. More specifically, to analyze possible links between change categories and ownership, we discretize the values of each of the three metrics across the 18 projects into low/medium/high values using equal frequency binning (3 bins). For example (if no overlapping metric values), the six projects with the lowest density value for "Adaptive" changes across all 18 projects are mapped to "low", the six projects with the highest density are mapped to "high", and the remaining six to "medium". We do this for each change category, and for each of the three metrics. For each metric and (change category,ownership style) pair, we then perform Pearson chi-squared tests (with α value of 0.05) with as null hypotheses that the distribution of the (discretized) metric for the change category is independent of build ownership. Furthermore, Table 8 provides some qualitative examples of decisions and discussions from bug reports for each owner-

Table 8: Build Ownership Styles with Example Discussions from Bug Reports.

Dedicated	*In any case, seems the fewer relative paths the better ... more modular, for future I suppose if we wanted to spend time on re-working how Ant launches / composes its classpaths / handles class loading we could probably do away with the support jars completely, but to be honest, no one has time to invest in that amount of work*
Strong	*sdk adds source bundles which are not used by the tests. Removing this should have some minimal effect to reduce our already way too long build time but we should still have a recommended structure, so we're consistent*
Collective	*'Or ... there are still 25 bundles without quite the right relativePath or groupId, seems to come from a range of projects If we are going to make breaking changes like this then we might as well stop providing the bundle altogether Lets not "improve" the bad practice'*

ship style.

Findings.

Projects having strong and dedicated ownership styles tend to make more invasive "Adaptive" changes to adapt the builds to a new environment or source code feature. Only the Pearson chi-squared tests for invasiveness of "Adaptive" changes was able to reject its null hypothesis (p-value of 0.01042). This shows that the "Adaptive" changes are not independent of build ownership, i.e., projects with strong and dedicated ownership styles tend to be more thorough in their "Adaptive" changes.

We can attribute this to two reasons. In projects with collective ownership, there is a distributed responsibility among several developers to make the adaptive changes, and hence these occur in a less concerted way, appearing dispersed. Second, we often find in collective ownership projects that developers need time to find all areas of the build code that need to be modified for an "Adaptive" change. As such, those changes are spread across multiple commits, each one being less invasive. This appears to be due to unfamiliarity with build code in such projects. We note this particularly from commit messages and bug report exchanges for the Eclipse Platform project, where the dedicated build expert coordinates such large "Adaptive" changes across all other (collectively owned) projects interacting with Eclipse Platform.

However, we did not find a link between "Adaptive" changes and build ownership styles for the commit density and build churn metrics. This shows that although projects with collective build ownership tend to make smaller "Adaptive" changes over dispersed commits, the number and churn of such "Adaptive" change commits does not differ significantly compared to those of projects with strong and dedicated build ownership styles.

The invasiveness of "Adaptive" changes is affected by the build ownership style. However, we do not find a significant link between the other change categories and the build ownership styles.

5. THREATS TO VALIDITY

The study is based on a pre-defined categorization of build changes [11], extended with two new categories identified during our study using a card sort approach [4]. There can also be categories of build changes that have not been catalogued here, simply because we did not encounter these in the studied projects. In future work, we plan to improve this catalogue, by performing a study of different build system flavours, of different sizes, including for example make-based systems.

We refined McIntosh et al.'s [13] original categorization of build system ownership styles according to Martin Fowler's definition of source code ownership styles and based on our discussions with Eclipse developers. However, it is possible that certain software systems exhibit other types of ownership styles, even though we cover the extreme cases. Also, other factors like the software development methodology, the programming language used, the software architecture (blackboard vs. layered) and the experience level of the developers in the software system can affect the evolution of build changes. Again, we intend to study these aspects by mining projects with different characteristics.

Another factor that might impact our analysis of build changes is the maturity of the subject systems. For example, young systems might be more prone to "New Functionality" changes (since they are starting from scratch) and "Adaptive" changes (since their build technology or architecture might not yet be set in stone). Mature systems on the other hand might see more "Corrective" changes. However, we could not find a direct link between maturity and build system changes. For example, in the Apache ecosystem, the Apache-Hadoop-Common project is one of the youngest projects (although it as well is at least 8 years old), yet it has less "New Functionality" changes than Apache Maven, which is an older project. More work is needed to clear up the impact of maturity.

For some Eclipse projects, the period of data that we studied also coincided with a migration from Ant-based builds to Tycho-based builds. Hence, our data during this period for these projects, which were still actively migrating to Tycho, could be biased. However, only few Eclipse projects suffered from this problem, while the Apache projects did not have this problem at all.

To the best of our abilities, we tried to select projects with a considerable number of build files and build code churn. Furthermore, we opted for two ecosystems in order to make our results comparable within the ecosystems. However, our results may be different when we study commercial software systems or systems with much higher build churn. We plan to perform practitioner interviews to deal with this.

6. CONCLUSION

In this paper, our goal was to "unveil" what is going on during build system maintenance, by mining the types of changes being made to build files, their size, churn and invasiveness, instead of just counting the number of changes. We also studied these change types in light of the build ownership styles of the studied projects.

For this, we studied thirteen Eclipse projects and five

Apache projects with varied characteristics and ownership styles. By grouping changes into six major categories of changes ("Adaptive", "Corrective", "Perfective", "Preventive", "New Functionality" and "Reflective"), we showed that the "Corrective", "Adaptive" and (to some extent) "New Functionality" changes are the most common, and induce the largest churn and invasiveness in the build system. Still, most of these changes are found and made as part of other build or code changes, and we noticed various occasions where the completeness of a change was not guaranteed and led to reverting of changes or many additional (batched) changes. Measures should be investigated for assuring the quality of build system changes (e.g., change impact analysis or build tests).

Finally, we also discovered that for the "Adaptive" change category the build ownership style plays a role in that more daring, invasive build changes are being attempted for dedicated/strong ownership compared to the weaker collective ownership. Their larger build system know-how enables projects with dedicated and strong ownership style to adapt their build system quicker to new environments or source code features.

We believe that our findings form a first step for practitioners who perform build maintenance activities to understand and name the different kinds of build changes, and how they correlate to organizational characteristics of the build system like build ownership. The next step will be to link these changes to a measure of build system maintainability to identify which changes are more risky or error-prone than others.

Acknowledgements

We are grateful to Mrs. Kim Moir for her guidance in identifying the build ownership styles in Eclipse projects.

7. REFERENCES

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

[2] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter. Design recovery and maintenance of build systems. In *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 114–123, 2007.

[3] J. J. Amor, G. Robles, J. M. Gonzalez-barahona, and A. Navarro. Discriminating development activities in versioning systems: A case study. In *Proc. of the 2nd intl. workshop on Predictor Models in Software Engineering (PROMISE)*, 2006.

[4] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proc. of the 2013 Intl. Conf. on Software Engineering (ICSE)*, pages 712–721, 2013.

[5] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proc. of the IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, pages 73–82, 2010.

[6] L. Grimmer. Building mysql server with cmake on linux/unix.

[7] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt. Automatic classication of large changes into maintenance categories. In *Proc. of the 17th IEEE Intl. Conf. on Program Comprehension (ICPC)*, pages 30–39, 2009.

[8] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proc. of the Intl. Working Conf. on Mining Software Repositories (MSR)*, pages 99–108, 2008.

[9] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In *Proc. of the Intl. Symp. on Empirical Software Engineering and Measurement (ESEM)*, pages 384–387, 2011.

[10] T. G. W. Kumfert and G. K. Epperly. Software in the doe: The hidden overhead of "the build". Technical report, Lawrence Livermore National Laboratory, 2002.

[11] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: a Study of the Maintenance of Computer Application Software*. Addison-Wesley, August 1980.

[12] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of ant build systems. In *Proc. of the 7th IEEE Working Conf. on Mining Software Repositories (MSR)*, pages 42–51, 2010.

[13] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *Proc. of the 33rd Intl. Conf. on Software Engineering (ICSE)*, pages 141–150, 2011.

[14] P. Miller. Recursive make considered harmful. *AUUGN Journal of AUUG Inc*, 19(1):14–25, 1998.

[15] A. Neitsch, K. Wong, and M. W. Godfrey. Build system issues in multilanguage software. In *Proc. of the 28th IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 140–149, 2012.

[16] A. Neundorf. Why the kde project switched to cmake–and how (continued), 2010.

[17] S. Phillips, T. Zimmermann, and C. Bird. Understanding and improving software build teams. In *Proc. of 36th Intl. Conf. on Software Engineering (ICSE)*, pages 735–744, 2014.

[18] F. Rahman and P. Devanbu. Ownership, experience and defects: A fine-grained study of authorship. In *Proc. of the 33rd Intl. Conf. on Software Engineering (ICSE)*, pages 491–500, 2011.

[19] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. Programmers' build errors: A case study (at google). In *Proc. of 36th Intl. Conf. on Software Engineering (ICSE)*, pages 724–734, 2014.

[20] P. Smith. *Software Build Systems: Principles and Experience*. Addison-Wesley Professional, 2011.

[21] R. Suvorov, B. Adams, M. Nagappan, A. Hassan, and Y. Zou. An empirical study of build system migrations in practice: Case studies on kde and the linux kernel. In *Proc. of the 28th IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 160–169, 2012.

[22] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Build code analysis with symbolic evaluation. In *Proc. of the 34th Intl. Conf. on Software Engineering (ICSE)*, pages 650–660, 2012.

[23] Q. Tu and M. W. Godfrey. The build-time software architecture view. In *Proc. of the IEEE Intl. Conf. on Software Maintenance (ICSM)*, pages 398–407, 2001.