What are the Testing Habits of Developers? A Case Study in a Large IT Company

Abstract—**Tests are considered important to ensure the good behavior of applications and improve their quality. But development in companies also involves tight schedules, old habits, less-trained developers, or practical difficulties such as creating a test database. As a result, good testing practices are not always used as often as one might wish. With a major IT company, we are engaged in a project to understand developers testing behavior, and whether it can be improved. Some ideas are to promote testing by reducing test session length, or by running automatically tests behind the scene and send warnings to developers about the failing ones. Reports on developers testing habits in the literature involve highly distributed open-source projects, or are based on a study of students programmers. As such they might not apply to our industrial, closed source, context. In this paper, we take inspiration from experiments of two papers of the literature to enhance our comprehension of the industrial environment. We report the results of a field study on how often the developers use tests in their daily practice, whether they make use of tests selection and why they do. Results are reinforced by interviews with developers involved in the study. The main findings are that test practice is in better shape than we expected; developers select tests "ruthlessly" (instead of launching an entire test suite); although they are not accurate in their selection, and; contrary to expectation, test selection is not influenced by the size of the test suite nor the duration of the tests.**

Index Terms—**Regression Test Selection, IT company, Case study, Interviews.**

I. Introduction

With the increase in complexity of software applications, the need to test every piece of code becomes compulsory. For a long time, tests were done by running the application manually. This practice is not dead, but with the improvement of testing tools, frameworks, or processes, there is a push to automate testing and make it more systematic.

However, developers may be slow to adopt new practices, either by lack of training, lack of understanding of the advantages of unit testing, or sheer habit. And even when they do implement automated tests, their use might still be hampered by other factors: In a major IT company, we found projects with an environment so complex and with so many tests that hours are necessary to run them all. One of our hypotheses is that this factor hampers the proper use of automated testing in the company: developers are not encouraged to regularly test every modification they make if this implies getting the answer hours later.

Yoo and Harman [16] expose several solutions to this problem: First, the test suite minimization identifies tests covering the same piece of software and remove them. The test suite is reduced, but not drastically because, in practice, only a few tests are redundant in term of coverage. Once the suite is reduced, all tests are run. Second, the test suite prioritization first runs the tests that could be impacted by the developer modifications and second all the other tests. Such a solution is valuable but, one still need to wait for the whole test suite execution to know that there is no error. Third, the test selection — or RTS, Regression Test Selection — [5, 8, 16] selects only the tests that could be impacted by the modifications. In this paper, we focus only on the latter solution.

But current development environments (*e.g.,* Eclipse, Intellij), configuration management systems (*e.g.,* Maven), or version control tools (*e.g.,* Git) do not offer integrated test selection mechanisms other than manually selecting and running one test, one class, or one package; or by entering in a complex manual configuration.

Based on our hypothesis that lengthy test sessions are a factor slowing the adoption of good testing practices, the major IT company engaged in a partnership with us to (i) characterize the testing behavior (or habits) of its developers; and (ii) look for solutions on how to improve it.

We found two publications in the literature that matched these goals. First, Beller et al. [4] study the usage of the IDEs by the developers to understand "When, How, and Why Developers Test". This research would cover well our need to characterize the testing behavior of developers. However, most of the projects in the study are open-source projects and as a consequence the conclusions may not apply to the company (open-source projects have been known to behave differently from closed source ones [17]). Second, Gligoric et al. [6] is a "Comparison of Manual and Automated Test Selection". As such, it seems to match well our second goal of verifying whether test selection can improve testing practice. However, most of the developers in the study were students and again, the conclusions may not apply to our context.

We wanted to make a similar case study in the environment of the company. Therefore, we took inspiration of both case studies[4, 6] to establish a baseline to compare to. A secondary goal of the case study (out of the scope of this paper) is also to collect base data to be able to detect possible impact of future (automated test selection) actions.

The main findings of our study are:

- We found that test practice are better than what we expected, even when one compares to open-source development that may have an image of being more test oriented (or driven);

- Developers perform test selection to a high-level;
- However, they are not accurate in their selections, launching unneeded tests and ignoring tests that would exercise their new code;
- Finally and contrary to one of our initial hypotheses, they do not perform test selection because of long delays in test executions or because a project has too many tests, but because these tests are the ones they believe to be relevant.

In Section II, we give background information on our problem and describe the case studies proposed by Gligoric et al. and Beller et al. that inspired our study. Then, in Section III, we define the research questions and describe our experiment. Section IV analyses and compares the results of the case study. Section V presents the threats to validity of our case study. Finally, we present the related works and conclude in Section VI and VII respectively.

II. Problem Description

In large industrial projects, executing all tests after each change can turn into a costly operation requiring several hours.

The company developers run automatic tests through their IDE and we would like to support them in their task.

A. The Company Problem

The company is the European leader in payment and transactional-services industry. It is present in 17 countries across the globe with approximately 7,500 employees.

Tests are crucial to the company for different reasons. First, the company provides payment and transactional-services that are critical. Errors, bugs or denials of service are not allowed. Second, the company provides solutions from design to deployment and maintenance. Maintainers can use information from tests to help them evolve programs [1].

The company is always looking for ways to improve its development practice. We set up a poll for the company employees to identify the potential area of actions. It concluded that testing was a main subject of improvement.

By investigating this topic, we found that, for some projects, running all tests takes hours. This seemed a problem for developers checking the validity of their latest development. It was hypothesized that introducing automated Regression Test Selection (RTS) techniques could improve this situation. Such techniques try to minimize the size of the test set to re-run after a code modification to only those that could actually be impacted by this modification. The reduction in size can be very significant (a few percent of the entire test suite [3]) and thus can provide a tremendous feedback speed-up to the developers.

To convince upper management of possibly imposing a change in work practices of thousands of developers, we need hard data on the gains that such a change can bring. We therefore looked in the literature to find more information on this. Two papers were found that shed some light on the issue. However, we have reasons to question their relevance to our context (see Sections II-C and II-D). We therefore decided to take inspiration from these case studies to better understand the testing behavior of the developers in the company and to make our own hypothesis (see Section III).

B. Regression Test Selection

Before presenting the case studies of the literature, we introduce here the concept of Regression Test Selection (RTS).

More formally, following Rothermel and Harrold [13], the test selection problem is defined:

Problem: Given program *P*, its modified version *P'*, and test set *T* used previously on *P*. Find a way, making use of *T*, to gain sufficient confidence in the correctness of *P'*.

A solution to this problem is to only select the tests in *T* that exercise the modified code in *P'*. The outcome of the other tests should not have changed since they are not impacted [16].

Some test case selection approaches are based on the notion of dependency graph. The general idea is that tests can be said to depend on the source code that they exercise. After a piece of code is changed, a test case selection technique just needs to navigate the dependency graph and go back from the changed piece of code to the tests that depend on it. A change can be any modification of the source code, even it has no impact on the application behavior. Figure 1 illustrates this principle for two methods and two tests: testMethod1 depends on method1 and method2 (testMethod1 calls method1 and method2), testMethod2 depends only on method2. If method1 is changed, only testMethod1 needs to be re-run as the outcome of testMethod2 cannot have changed. This is, of course, a simplified example. In real cases, establishing dependencies from tests to code is made difficult by many adverse factors [3].

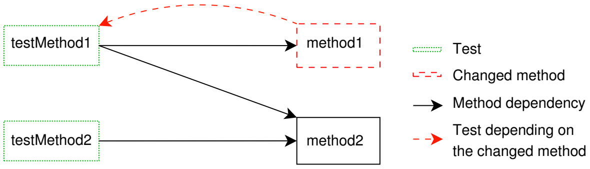


Fig. 1. Test Selection Simple Case

We now proceed to describe the two previously published case studies that we take inspiration from in our study.

C. Paper 1: "When, How and Why Developers (Do Not) Test in Their IDEs"

Beller et al. [4] report on a large scale, field study, monitoring 416 software engineers. Their findings of interest to us are:

- A majority of the developers rarely test in their IDE (note that they could run tests outside of the IDE). The authors explain this fact by the difficulty to perform adequate and fine RTS in the existing IDE. This would indicate that

testing is not a common practice. This is a preconception that we had in the company;
- Statistically, quick tests do not lead to more test executions. Developers selected test cases whatever the duration. This would not fit our hypothesis that developers avoid test executions because of their duration.
- Some failing tests are fixed later: 50% of the test repairs happen within 10 minutes whereas 75% within 25 minutes. This indicates a good test practice: tests results are considered. On the other hand, 25% of the failing tests take a long time to be corrected. If we get similar results, this is one of the points we strive to improve.

Anyone could take part in this case study, but it seems that most of the projects were open source. As such, this case study may not apply to our closed-source context (see a counter example of predicting closed source properties from open-source ones in Zimmermann et al. [17]).

D. Paper 2: An Empirical Evaluation and Comparison of Manual and Automated Test Selection

The second paper of interest is the one of Gligoric et al. [6]. The authors assessed how developers manually select tests and compare this manual selection to an automatic one. They conclude that there is a need for better automated test selection techniques that integrate well with developer IDEs.

For their study, Gligoric et al. use a group of 14 developers composed of five professional developers and nine students. They asked them to install a plugin in their IDE which records code changes and test executions. It is possible that students are better trained on regression testing techniques that developers of the company. Consequently, because of the high rate of students, the conclusions from this research might also not fit well our context. We expect that less testing is made in the company.

This paper focuses more on RTS than the previous one and the main findings of interest to us are:

- Test selection is frequently done (59% of the test executions), and most of the time, the ratio of tests selected is less than 20%. It is important to see that developers routinely perform test selection. We wonder if it applies in our context;
- There is a low correlation between the amount of code changed immediately before a test session and the number of manually selected tests in that session. This finding is in opposition with Beller observations. We need our own position on this issue;
- Manual selection results in more tests executed than automated selection in 73% of the cases and results in less tests executed in 27% of the cases. This shows that manual RTS miss to run some potential failing tests. This could be improved by an automated tool. We need to see whether this is the case in the company too.

III. Experimental Setup

This section presents the research questions that we set to ourselves and the methodology to answer them.

A. Research questions

The two experiments we just described seem to answer well our needs: the first one characterizes how developers use tests in their daily work, and the second one characterizes how developers use RTS which can provide faster feedback after modifying a piece of code. However, we highlighted that both experiments also had characteristics that did not fit well our context and that might render their conclusions useless to us. We only kept the questions that are of interest for us and where data where available.

We set the following research questions for our case study:

RQ1: How and Why Developers Run Tests?
This research question mostly take inspiration from both papers. It is decomposed as follows:

RQ1.1 Do developers test their code changes?
RQ1.2 How long does a test run take?
RQ1.3 Do quick tests lead to more test executions?
RQ1.4 Do developers practice test selection?
RQ1.5 What are common scenarios for manual RTS?

RQ2: How Do Developers React to Tests Run?
This research question stems from paper 1:

RQ2.1 How frequently tests pass and fail?
RQ2.2 How long does it take to fix a failing test?

RQ3: How and Why Developers Perform Test Selection?
This last research question includes most of the questions from paper 2:

RQ3.1 Does manual test selection depend on size of test suites?
RQ3.2 Does manual test selection depend on size of code changes?
RQ3.3 How does manual test selection compare with automated one, in terms of precision and safety?

B. Experimental protocol

Participation to the case study was voluntary. We advertised it on the internal mailing of the major IT company and we set up a lottery to attract more volunteers. We also did as much advertising as possible through our network of relations. Participants had to download and install one of the plugins we developed (see after). These plugins made data collection completely transparent for the participants which was a strong requirement for them.

For data collection, we needed information on the test runs and from the source code (to compute code changes). One difficulty is that the development environments of the company are heterogeneous. Developers can code in the IDE of their liking (usually Eclipse or IntelliJ), and use different frameworks to run their tests (usually JUnit or Maven). The versions of all these tools are also not always the same.

Developers were *very* concerned that the participation to the case study should not add any burden or delay to their normal work. This, combined with the heterogeneous aspect of the developers environments, limited the data we could collect. It made it very difficult to log data with the same level of detail as the two case studies (down to keyboard and mouse events for Beller et al. [4]). This in turn impacted how well we can answer some questions (see Section III-C). We elected to collect test information through plugins that were developed for Eclipse and IntelliJ, each one logging the same data:

Developer id: A unique id given to the developer;

Project name: Refering to the Eclipse or IntelliJ project;

Repository URLs: The names of the source code repositories related to the project, one project can be stored on several repositories;

Repository version: The source code version in the repository, *i.e.,* commit id of the last checkout/update/pull request;

Test session start: Timestamp (date and time) of the launch of the test runner;

Test session end: Timestamp at the end of the last test execution;

Tests executed: The list of each test executed in the session with the following details:

- **Fully qualified method name:** The name of the test method with its class and package;
- **Test duration:** The duration of the method execution;
- **Test status:** The result of the test: PASS, FAIL (wrong assert), ERROR (unexpected exception), or SKIPPED (*e.g.* annotated with @Ignore in JUnit);

The plugins record the tests sessions (if they are launched from within the IDEs) and send the data to a server. The plugins look for either JUnit runs or Maven runs. Tests run out of IDEs are not logged. This can be a concern, primarily for Maven, as it is rarer to run JUnit stand-alone.

C. Filtering and Massaging Data

As usual for *in vivo* case studies, filtering and massaging data to get meaningful answers, was a major task. Because of the way we collected data — this in turn dictated by a strong requirement from the company and the developers —, some information were not readily available. We discuss here some of the hypotheses we had to make.

Test session. Gligoric et al. [6] define a test session as a run of at least one test between two sets of code changes. Beller et al. [4] split developers work in Eclipse sessions: from the opening of the IDE until its closing. They see Eclipse sessions as natural dividers between work tasks and between work days. Next, they subdivide sessions in intervals. A JUnitExecution interval is created at the invocation of the JUnit runner (or Maven test build) and ended when another interval starts (*e.g.*, typing interval).

For this paper, we defined a test session as one execution of the JUnit test runner or of a Maven test job. They can be composed of one or several tests. In Figure 2, test sessions are represented by $T1, \ldots T7$.

However, because we are considering test selection and the tools used are not well suited for it, we have to introduce another concept (Agglomerated Session, see next). Consider a developer wanting to run two specific tests from two different classes. With JUnit, the options are either to run all the tests of the project or run independently the two tests. Developers often choose the second option. This means we will have two test sessions. However as far as test selection is concerned, we would like to consider that there was only one "session" including the two tests.

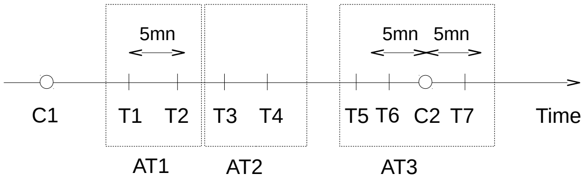


Fig. 2. A test/code session with three agglomerated test sessions (AT1, AT2, AT3) themselves comprising several test sessions (T1, ..., T7). C1 and C2 are commits, C1 being the direct ancestor of C2. All events after C1 occur on the same project by the same developer.

Agglomerated test session. An agglomerated test session is a set of successive tests sessions in the history of a project. Beller et al. also consider test session (called JUnitExecution) and agglomerated test session (called Test Session). Because we do not log every interaction in the IDEs (keyboard or mouse events), we need an heuristic to bound the agglomerated sessions. Two successive test sessions on the same project from the same developer id will be agglomerated as long as:

- The two test sessions occur within a fixed time frame (we chose 5 minutes). Beller et al. [4] have a similar heuristic with their "reading interval" backed by an *inactivity timeout* of 16 seconds;
  In Figure 2, $AT2$ and $AT3$ are two separate agglomerated test sessions because the time frame between test sessions $T4$ and $T5$ is greater than 5 minutes.
- No single test occurs twice in an agglomerated session. The idea here is that if a developer runs a test, then change the code (which we cannot see), then rerun the test to check if it works, we do not want to group both execution of the test as a single group of tests. Gligoric et al. [6] had the same issue but they can verify whether the developer coded between two executions of the same test or not. This issue is discussed in Session V.
  Figure 2, $AT1$ and $AT2$ are separate agglomerated sessions because $T3$ repeats a test also included in $T1$ or $T2$.

Code tested. Another important issue was to determine what code was being tested. Because we only monitor test sessions and commits (represented as circle in Figure 2), it is difficult to know exactly what was the source code tested. For the research questions that require this information (mainly those relating to test selection), we had to use another heuristic and a subset of all the test sessions.

To check the accuracy of test selection in comparison with the code changed, we must (i) associate the tests with the code tested, and (ii) identify what changes were made to the code.

Test/Code session. This is a test session that we could associate with a commit, and thus with the source code that was tested. For this, we group together test sessions and commits that occur on the same project, by the same developer id, and within a time frame of five minutes (similar to the agglomerated test session). This is the case for $AT3$ in Figure 2. The five minutes threshold was chosen after manually looking at a number of test sessions and commits that were close. One main difference between test/code session and agglomerated test session is that the former *must* have a commit associated.

Additionally there may be other test sessions ($AT1$ and $AT2$ in Figure 2) between a test/code session ($AT3$) and its ancestor commits ($C1$). Test sessions carry a commit identifier (see Section III-B), if this is the same as the ancestor commit ($C1$) then they will be added to the test/code session. The scenario envisioned here is that the developer does a checkout/update/pull request, then changes the code and tests it, then make further changes, tests it more, and finally commits it. In this case, we group the test sessions in the test/code session and we assume they all test the code that was committed. This last step is independent of any five minutes interval. This (partial) dismiss of the five minutes threshold is the other differences between agglomerated sessions and test/code sessions. In the end, everything after $C1$ in Figure 2 is considered one single test/code session.

Code change. We use the test/code sessions to compute code changes. We compare the code in the commits of a test/code session ($C2$, final code) to the code in the direct ancestor of these commits ($C1$, original code).

Amount of code changes. Some research questions require to evaluate the amount of code changed. This will be estimated as the textual (line based) diff between two versions of code. [6] used the number of AST node differences between two versions of the code, but because of the size of the projects and the number of projects, it was intractable for us to use the same solution.

D. Automatic Test Selection

To answer research question RQ3.3, we must compare manual test selection done by the developers with what they should have selected given the changes to the code. For this, we must compute code changes (see above) and what tests exercise these parts. As described in [3], there are several approaches to compute test selection automatically. The dynamic approach executes the tests and record their trace to see what methods each test calls. But given the difficulty to set up the development environment for all the projects (compilation of the source code, set up of the database...), it was not feasible to use this approach. We used the static approach that parses the code and deduces the methods likely called by a test through an analysis of the static call graph. If one of these methods is affected by a code change, the associated test should be selected and re-run to check the validity of the change.

E. Interviews with the Participants

To extract more insight from the participants to the case study, we conducted an interview at the end of the gathering of the results[15]. The participation to the interviews is on voluntary basis. Its goal is not only to get more context of the environment of the developers but also to have some explanation on the qualitative results extracted from the recording of the tests executions.

On face to face, or through video conferences, we conducted 20-30 minutes discussion. After a brief description of themselves (their experience in the company) and a quick description of their project, we asked them to describe their behavior about testing in the context of the application they are working on: how they create the tests, how they manually select the tests, launch then, what actions they take after a failing test. The result of the interviews are integrated in each research question to explain the quantitative results found by monitoring the developers.

IV. Results and Discussion

We now present our results and compare them to the two previous case studies. These results were obtained between April 20th, 2016 and March 8th, 2017. Tables I and II present some descriptive statistics on the case studies.

TABLE I
Descriptive statistics on the three case studies

	Beller et al. [4]	Gligoric et al. [6]	us
# Developers	48	14	32
# Projects	73	17	64
# Test sessions	3 424	5 757	14 686
# Agglomerated sessions	-	-	13 611
# Test executions	10 840	264 562	153 763
Tests / Session	3.2	45.9	10.5
Sessions / Developer	71.3	411.2	458.9
Study Duration (months)	4	3	10

We have 32 participants in 64 different projects which sets us in between both others case studies, closer to the first paper [4]. The second paper [6] had 14 developers as participants.

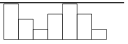
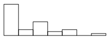
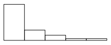
Among the 32 participants, 11 accepted to be interviewed. These participants are not only developers (7 people) but also technical leaders (3), or architects (1).

We have more test sessions (14 686) than the first paper (3 424) with less participants (see also session/developer). We also have an order of magnitude more single test executions (153 763 for us; 10 840 for the first paper). This can already be seen as a good indication for test practice in the company. But comparisons are made difficult by the fact that we do not know which developments principles are used in the open-source projects.

We are also intermediary in the number of single test executions per test session (10.5) and above all for the number of sessions per developer (458.9). Again we have more test sessions per developer than the open and closed source projects which is a good sign for the company.

As an additional indication, we give in Table II statistics for developers of our case study: number of days of collecting data, number of test sessions, and number of sessions per day

TABLE II
DESCRIPTIVE STATISTICS PER DEVELOPER

	Min	Q_1	Median	Q_3	Max	Histogram
Calendar Days	1	60	193	241	326	
Activity Days	1	6	23	54	140	
Sessions	3	32	192	706	2343	
Sess./Activity Day	1	3	8	18	110	

where tests have been made (activity days), all developers combined. We note that developers participated to the case study for more than 5 months on average and did 61.7 testing sessions per activity day. From this last number, it seems that testing is well implanted in company's developers daily practice. However, by running the interviews, it seems that in a majority of projects of the company, developers do not run tests automatically during the development but only manually or at the integration testing step.

A. *RQ1: How and why developers run tests?*

RQ1.1 Do developers test their code changes?

For this question, we evaluate whether there is a correlation between the number of tests run and the number of changes to source code. We used Spearman correlation as our data do not follow a normal distribution. The correlation is weak $\rho = 0.20$ thus confirming that more code change tend to lead to more tests, but to a very slight degree.

Beller et al. [4] differentiate the number of changes to test code or number of changes to production code. They correlated them both to the number of tests run. They have a good correlation ($\rho = 0.66$) with test code changes, and a weak one ($\rho = 0.38$) for production code. Our correlation goes in the same direction than their results but is weaker despite the fact that our changes concatenates test and production code changes.

From the interviews, it seems that developers would like to run tests after they made changes in their application. However, some pieces of software like human machine interfaces, insertion in databases, or complex systems requires specific testing frameworks. Theses frameworks are difficult to put in place due to a lack of training and time to understand it. Moreover, developers confess that they often do not run tests after minor changes. The main reason given is lack of time.

RQ1.2: How long does a test run take?

We observe that 50% (median) of our test sessions finish in less than 3 sec. and over 75% (third quartile) of tests sessions finish within 16 seconds (results are similar for agglomerated sessions with respectively 3 and 18 sec.). Moreover, 9.4% of the test sessions take longer than 1 minute and 5.0% take longer than two minutes (respectively 10% and 5.6% for the agglomerated sessions). Detailed results can be found in Table III. In general, tests sessions are short.

We measured a maximum duration of the test sessions of 4 h 23. In this session, only one test was launched. Other executions of this test take few seconds to run. But, the results being anonymous, it is not possible to ask the developer for more information on this long duration.

Beller et al. report that 50% of their test sessions finish in less than 0.5 seconds and over 75% of the sessions finish within 5 seconds. For their test sessions, 7.4% take longer than one minute and 4.8% take more than two minutes. They conclude that most of the test sessions are short.

Results are comparable, orders of magnitude are the same excepted the duration of the test sessions. On this point, one could hypothesize that tests are broader in scope in the company. We confirmed through the interviews that among the tests, there are integration tests that require setting up a database and have a higher number of method tested than unit tests. These tests are longer to run than unitary tests.

RQ1.3: Do quick tests lead to more test executions?

To answer this question, we evaluate the correlation between test execution length and the number of times tests are executed. The expectation is that short tests will be executed more often, thus the correlation value is expected to be negative. However, our Spearman correlation was $\rho = 0.20$. Beller et al. also get a positive correlation of $\rho = 0.26$. Both lead to the conclusion that there is no relevant correlation.

These answers are contrary to expectation, faster tests are not executed more often (corollary: longer tests are not executed less often). This was one of our hypotheses to try to improve test practice in the company and it does not hold.

Interviewed people seems to launch the tests that cover the part of the application they changed without distinction of the duration of the test. Exceptions are made when the tests are too long (some minutes). In this case, the tests are not relaunched after each change but either after a set of changes or delegated to the continuous integration job.

RQ1.4: Do developers practice test selection?

We report 58% of the agglomerated sessions with only one test, 24.5% with more than 5 tests, and 4.0% with more than 50 tests. We can reach the conclusion that developers of the company practice test selection.

Beller et al. report that 87% of test sessions include only one test case, 6.2% include more than 5 tests, and 2.9% more than 50 tests. From this, they concluded that their developers did practice test selection. Gligoric et al. report 3 594 test sessions (62.4%) with only one test.

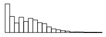
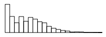
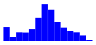
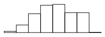
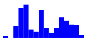
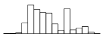
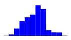
It seems company's developers and those in the second paper case study [6] select less "aggressively", *i.e.,* with less test sessions consisting of only one test.

These results are misleading on projects with very few tests: if a project had only 50 tests, one could not be expected to run more than this number and still not do test selection. The three case studies do have projects with less than 50 tests.

For us, in 50% of the test sessions, 4% of the available tests of the project are selected, in 75%, and 17.8% are selected (See Table III).

TABLE III

Comparison of our results with those of Beller et al. [4]. (When computing number of tests per session, we give results for test sessions and agglomerated sessions to match Beller et al.'s case study). Histograms are in log scale

			min	Q_1	median	Q_3	max	*unit*	histogram
RQ1.2	Test session duration	us (test sess.)	0	1.0	3.0	16.0	15 820.0	*second*	
		us (agglom.)	0	1.0	3.0	18.0	15 820.0	*second*	
		Beller	0	0.03	0.5	3.4	73.8	*second*	
RQ1.4	Percentage of executed tests	us	0	1.2	4	17.8	100	%	
		Beller	0	1	1	12.5	100	%	
RQ2.2	Time to fix failing test	us	0	3.1	4 981.0	1 042.0	359 600	*minute*	
		Beller	0	1.7	65.1	25.0	4 881	*minute*	

Beller et al. further note that in 50% of the test sessions, only 1% of the available tests of the project are selected, and in 75% of the cases, 12.5% are selected.

For us, almost all the tests (> 95% of all the tests) are selected in 1.8% of the test sessions, and, for Beller et al., all tests are launched in 3.7% of the cases

So developers of the company select more tests than those of the first paper when they select, but they execute all the tests available much less often, they almost always do test selection.

We report that test selection occurs in 81.4% of the studied test sessions, between Gligoric et al. (59.19%) and the other paper (about 96.3%[1]). Finally we report an average selection ratio (number of executed tests divided by number of available tests) of 8.8%. For Gligoric et al. this ratio is almost the same with 9.0%. So again, it seems that our company developers tend to select more tests when they select.

Thanks to the interviews, we identified several profiles of testers: Some developers run all the tests of the module or subproject where the modification has been made. It is the preferred solution if the tests are fast, else, developers select more rigorously the tests. Other developers run tests based on naming conventions: the test classes have the same name than the application one. A small group of developers uses the call graph available in the IDE to retrieve the tests to relaunch. It is an advanced approach to select the tests. The last group gather the majority of the developers, according to the interviews: they select tests according to their feelings and experience. The testers feel they know what tests are potentially affected by the latest changes.

RQ1.5: What are common scenarios for manual RTS?

By analyzing the data, we identified two common scenarios for test selection. They are also described as in Gligoric et al. [6]:

- "After one or more tests fail, developers usually start making code changes to fix those failing tests and keep re-running only those failing tests until they pass. After all the failing tests pass, the developers then run most of or all the available tests to check for regressions."
- "[Developers] fix tests one after another, re-running only a single failing test until it passes."

We found both these patterns in the case study of Gligoric et al. and in our company. In the interviews, developers said that they launch tests one-by-one to avoid side effects between the tests. They also run semi-automatic tests one-by-one: they run the tests injecting data in database automatically and check manually the result. But both scenarios are equally frequent and depend of the context the developer is into. Group launching is made when the tests are jointly failing and cover the same feature(s) of the application. One-by-one launch is frequently used when only one feature needs to be checked and there is only one test associated to it.

B. RQ2: How do developers react to test runs?

RQ2.1: How frequently tests pass and fail?

In our case, on 153 763 tests executions, the ratio of failing tests is 13% (20 272), and the ratio of passing tests is 83% (127 704). We can also report 4% (5 787) of skipped tests. In Beller et al. [4], on 10 840 tests executions, 65% (7 047) fail and 35% pass successfully.

We found a much lower ratio of failing tests in our case study. By interviewing developers, we can propose some explanations:

- The tests are launched and followed up. This shows a good practice about testing in the company that they maybe do not have in open-source projects.
- The tests are passing because they miss assertions to check the behavior of the application. But in reality, the test should fail.
- The tests are not really tests but are launching scripts to insert fields to set up the database.

However, these last two explanations do not justify such a large difference in the ratio of failing tests.

RQ2.2: How long does it take to fix a failing test?

[1] Our statistics from their numbers

In our case study, for the failing tests that get fixed, 50% are resolved in approximatively 20 minutes and 75% within approximately 17 hours 20 minutes. The maximum duration that we observed to fix a test is 249 days, 17 hours and 20 minutes, almost the duration of the entire case study (322 days).

In Beller et al. [4], for 70% of the tests (2 051), the authors observed at least one successful execution and 30% have no successful execution. Therefore a significant part of the tests are never fixed. For the 2 051 failing tests that are fixed at some point, 50% are executed again with success within 10 minutes and 75% within 25 minutes.

Results for this question can also be found in Table III.

Our longer delays could be caused by the fact that the tests in the company are broader in scope. They mainly implies complex environment with database or external applications. As already discussed in RQ1.2, broader tests would make it more difficult to pinpoint the error when they fail.

C. *RQ3: How and why developers perform test selection?*

RQ3.1: Does manual test selection depend on size of test suites?

On all projects we studied, all developers performed test selection. On the other hand, we have an average 254.1 tests per project with a minimum of 1 and a maximum of 2 216. We can conclude that developers performed manual test selection regardless of the size of their test suites. In Gligoric et al. [6], almost all developers performed manual test selection, and they also had a wide range of test suite sizes. They further report an average of 174.3 tests per project; the minimum was 6 tests, and the maximum was 1 663 tests. The authors finally add that "considering that these projects are of small to medium size, and because they exhibit manual [test selection], [they] expected that developers of larger projects would perform even more manual [test selection]."

Through our interviews, it appears that if the test take too much time (one minute), the test selection is more aggressive. The projects are frequently split into modules and each module contains its own tests. So, if a change occurs in one of the module, the developer relaunch all the tests of the module. If the developer knows which test is related to the part it changed, it will select only few tests to be relaunched.

RQ3.2: Does manual test selection depend on size of code changes?

We consider the relationship between the size of recent code changes and the number of tests that developers select in each test session. This correlation is $\rho = 0.11$ which is low and can be considered as not relevant correlation (although it was considered statistically significant with p-value < 0.01 meaning that the correlation is not null).

For Gligoric et al. [6], this correlation is $\rho = 0.28$. We found an even lower correlation than them.

The conclusion from this research question is that one would expect developers to run more tests after large code changes or to perform more test selection when there are more test in a project. The findings go against both assumptions. We may relate this to RQ1.3 where we noted that faster tests did not lead to more execution and to RQ3.1 where developers avoid to select tests that takes more than one minute to run. There seems to be convergence of evidences that, contrary to our hypotheses, test selection and execution are not significantly influenced by the duration of the tests or their number.

Interviews lead to the fact that developers are potentially running more tests if the change they made is in several modules: they run the tests of all modules impacted. But, most of the time, changes are located in only one module.

RQ3.3: How does manual test selection compare with automated one, in terms of precision and safety?

We present the comparison of manual versus automated test selection in a dot plot (Figure 3, right). Numbers of expected test selected (automatic test selection) are represented on Y axis, and number of actually selected tests (manual test selection), on X axis. The desired behavior would be to have all points on the diagonal $x = y$. Points above the diagonal indicate that the manual selection missed some tests (low recall, assuming the tests selected are all correct). Points below the diagonal indicate that the manual selection chose undesirable tests (low precision, assuming no needed tests were missed). Our study reports a correlation of $\rho = 0.16$ between the two metrics which confirms the visual impression of no relevant correlation.

Gligoric et al. [6] whose data are on left side of the Figure 3, got similar results with a correlation of $\rho = 0.18$.

Our case study can further report a precision of 37.43% (ratio of selected tests that are correct) and a recall of 28.77% (ratio of required test that were selected) which should be considered low results. The conclusion is that manual test selection is not accurate which was expected.

From the interviews, developers said that they do not always carefully select tests: sometimes they are launching more tests than required (to test the whole changes they made), or not enough tests (to test only the algorithm they just implemented). When the modification in the source code is minor or deals with a graphical part of the application, developers do not always run the associated tests.

V. Validity Discussion

This section discusses the validity of our case study using validation scheme defined by Runeson and Höst [14]. The construct validity, the internal validity, and the external validity are presented.

A. *Construct Validity*

Construct validity indicates whether the studied measures really represent what is investigated according to the research questions. The purpose of this study is to evaluate the behavior of developers of the company about testing.

We detected that developers may use "false" tests as a standalone application to run a server and make manual tests. We manually removed these by looking at all test sessions longer than 10 minutes and having only one or two tests. It could be the case that such tests are still present in our data for

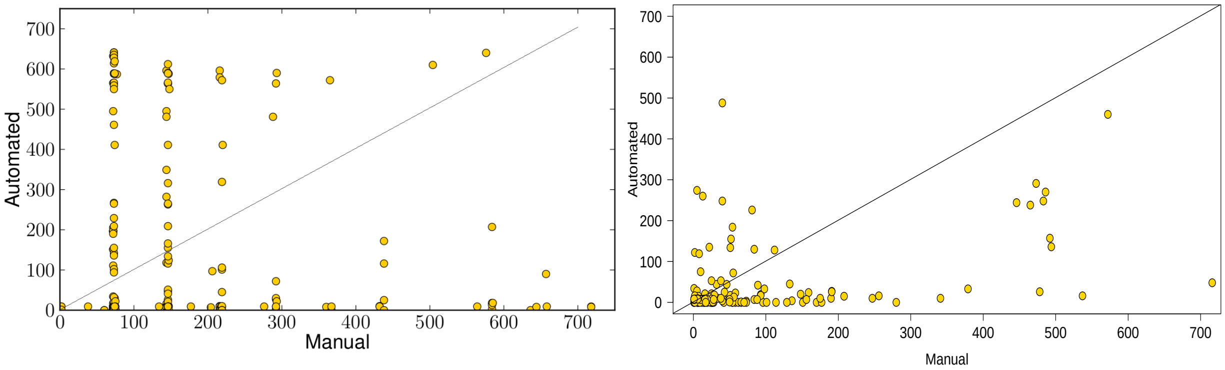


Fig. 3. Relation between the number of automatic and manual test selection (left [6], right our case study)

example if they were launched as part of a session with three tests but removing them all would require manually analyzing every test (little less than 7 000).

The plugin records only the execution of JUnit and Maven test sessions if they are launched from within the IDEs. If other tests runners are launched from outside the IDE (less probable for JUnit than for Maven), we would have no trace of that. This is a common issue that Beller et al. [4], Gligoric et al. [6] also had. However, the question was asked in the interviews. It appears that most of the automated tests are launched by the developers in their IDEs. Only a couple of developers are running tests through a maven command line. But it is mostly to deploy the application or because the tests are too long.

The presence of continuous integration is another bias that can persist in the case study. Despite the fact that developers should launch the tests locally to avoid committing potential bugs and propagate them to their colleagues, they tend to delegate this validation to the continuous integration. Consequently, less tests are made locally and potential bugs are dispatched to the others developers of the team.

Associating code tests and test sessions is still a real issue. The ideal solution would be to record all the source code for each test session. But this would mean a much more intrusive plugin that we are warry to install on the developers computers for now. One action that we may take would be to check whether our current (imperfect) solution makes a noticeable difference in the result compared to a “perfect” one. This can actually only impact the results on the computed precision of the test selection.

B. *Internal Validity*

Internal validity indicates whether no other variables except the studied one impacted the result.

The developers know that they are under study, Hawthrone effect[2] may have taken place. However the case study lasted 10 months and this effect usually levels off after some time.

The sample may be biased towards developers who are actively interested in testing because participation was voluntary. In this sense, our results could be an overestimation of the real testing practices.

To encourage participation of less testing aware participants, we organized a lottery (Beller et al. did the same), but it only brought four additional participants.

C. *External Validity*

External validity indicates whether it is possible to generalize the findings of the study.

Participants to the case study all originated from the same IT company. This would point toward a real threat to generalization. However, the fact that we make our case study close to the Beller et al. [4], Gligoric et al. [6] ones, in different conditions, and mostly confirmed their results, seems to be a good enough guarantee.

All projects studied are in Java (as 80% of the projects developed in the company). This can be an issue to generalize on other programming languages where testing is more difficult to do.

Testing practices of 32 employees during more than 10 months were studied, it is a case study of the general state of testing in a major IT company and can be difficult to generalize to other company.

VI. Related Works

The case studies of Beller et al. [4], Gligoric et al. [6] are already described in Section II-C and Section II-D, we took inspiration of these papers for this study.

Kasurinen et al. [10] interviewed 55 industrials from 31 companies and studied 12 software systems in development. Their survey revealed that organizations use automated testing only in 26% of their test cases. It is considerably less than the authors expected based on the study of the literature. The results indicate that adopting test automation in software organization is a demanding effort. The lack of a common strategy for applying automation was also evident in many organizations they interviewed. These observations also indicate communication gaps between stakeholders of the overall testing strategy, especially developers and testers. The major company we work with wants to improve these flaws: Tools

[2] Tendency of people to work harder and perform better when they participate in an experiment.

and trainings are given to make developers more efficient. Through our interviews, we saw that a majority of the projects use manual tests, but there is a will of the developers to have more tests in their application. But, for this training, developers have to spend time they do not have due to.

Pham et al. [12] conducted a study with 97 computer science students and made interviews to explore their attitudes regarding testing in a collaborative software project. Students tend to push test automation to the end of the project and consequently avoid to have a test suite during the development. The authors explain that it is mainly because of lack of time that they do not become productive with testing. We felt the same behavior with the employees of the company. Due to the tight schedules, testing is often left in background. However, a majority of the interviewees thinks that the tests are essential in the development of their applications.

Jones [9] implemented for Ada software an approach to select a safe set of tests covering some code change. They use a combination of source code analysis and test execution coverage to find the minimal, safe, set of tests potentially affected by a change. The experimental results on two software systems show a 97% reduction in test-case execution and an 84% reduction in retesting time. Their solution is more advanced than the traditional static analysis we used here. Adding test execution coverage will provide more safeness to our static analysis solution (does not miss a test) and minimality (can select more tests than required).

Legunsen et al. [11] evaluate static RTS against dynamic one. Static RTS could be more beneficial than dynamic RTS for systems with long-running tests, non-determinism, or real-time constraints. It shows promising results when used at the right granularity. For our case study, we would not like to spend time to run all the tests. This action is indeed required to create gold standard for the dynamic approach. In Gligoric et al. [7], the authors present Ekstazi, a lightweight RTS technique. This tool tracks dynamic dependencies of tests on files. Evaluated on 32 open-source projects, Ekstazi allows to reduce the testing time of 32% and 54% for longer running test suites. It is now integrated in some popular open-source projects like Apache Camel. The approach the authors use requires to run the tests. In our context, we were not able to use these approach due to difficulties to set up the execution environments for all the projects of the case study.

Amann et al. [2] studied the general usage of the Visual Studio IDE. They tracked the interactions with the tool of 84 professional C# developers in an industrial environment, combining 6 300 hours of work time study. They found that unit-testing tools are rarely used. They mention a tool (NCrunch) that automatically runs tests on identified code changes and displays the results. They estimated that 9 developers (11%) used this tool over a total of 21 developer days (2%[3]) whereas testing tools are "used on little more than a fourth of all developer days." They deduce that the number of developers actively using NCrunch is small. NCrunch (only available for VisualStudio) matches the solution we wish to implement in the company in that it runs tests in the background to pro-actively give feedback to the developers. However, they describe their test selection mechanism as still experimental and very rudimentary[4]. We have no information to explain its low adoption in the experiment, but foremost, we have no information on its consequences on the testing habits of the developers.

VII. Conclusion

Before trying to improve testing practices of the major IT company developers, we need to understand them. This paper presents an empirical study inspired from two previous case studies [4, 6], in an industrial, closed source context. The Beller et al., Gligoric et al. case studies were conducted with students (mainly) in one case [6] or on open-source projects in the other case [4]. Thus it was not clear whether their conclusions applied to industrial closed source environments. We could confirm many findings of both papers, thus giving more weight to their conclusions.

Our main findings are:

- Test practice in the company is good compared to open-source developers practice [4]. This was not entirely expected but could be biased by the voluntary participation in the case study;
- Developers do perform test selection, and to a high-level. For example more than half of the test sessions ran only one test. This is coherent with previous findings;
- Manual test selections is not accurate, many impacted tests are not launched after a change (recall=37%) and others are whereas they did not need to (precision=29%). We noted a tendency to err on the side of minimality rather than safeness. This is in contrast with the reports from Gligoric et al. [6] (73% of the sessions executed more tests than an automated RTS would have);
- Contrary to intuition, amount of test execution or test selection do not depend on the size of the test suite nor on the duration of the tests: Longer tests are not executed less often, and the number of tests run is not impacted by the total number of tests available in a project. Interviews confirmed that tests are actually selected on their ability to confirm the quality of a code change. This is good news at it reinforces the need to provide adequate test selection mechanisms to help software developers getting faster and better feedback.

As future work, we plan to introduce in the company a solution that will test developers code changes in the background and give them rapid feedback on the passing/failing tests for these changes. The results collected here will be used as a ground basis to evaluate the impact of this future solution on test practices in the company.

Acknowledgments

Details omitted for double-blind reviewing.

[3] Our statistics from their numbers.

[4] From consulting the web page http://www.ncrunch.net/documentation/concepts_engine-modes, 08/23/2016

References

[1] Hiralal Agrawal, James L. Alberi, Joseph Robert Horgan, J. Jenny Li, Saul London, W. Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, 1998. doi: 10.1109/2.689678. URL http://dx.doi.org/10.1109/2.689678.

[2] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A Study of Visual Studio Usage in Practice. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 124–134. IEEE, 2016.

[3] Anonymous. Details omitted for double-blind reviewing. 2016.

[4] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, How, and Why Developers (Do Not) Test in Their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786843. URL http://doi.acm.org/10.1145/2786805.2786843.

[5] Emelie Engström, Per Runeson, and Mats Skoglund. A Systematic Review on Regression Test Selection Techniques. *Information and Software Technology*, 52(1):14–30, 2010.

[6] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 361–372. ACM, 2014.

[7] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 211–222, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3620-8. doi: 10.1145/2771783.2771784. URL http://doi.acm.org/10.1145/2771783.2771784.

[8] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001. ISSN 1049-331X. doi: 10.1145/367008.367020. URL http://doi.acm.org/10.1145/367008.367020.

[9] Andrew V. Jones. Addressing the Regression Test Problem with Change Impact Analysis for Ada. In Marko Bertogna, Luis Miguel Pinho, and Eduardo Quiñones, editors, *Reliable Software Technologies – Ada-Europe 2016*, number 9695 in Lecture Notes in Computer Science, pages 61–77. Springer International Publishing, June 2016. ISBN 978-3-319-39082-6 978-3-319-39083-3. doi: 10.1007/978-3-319-39083-3_5.

[10] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Software test automation in practice: empirical observations. *Advances in Software Engineering*, 2010, 2010.

[11] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 583–594. ACM, 2016.

[12] Raphael Pham, Stephan Kiesling, Olga Liskin, Leif Singer, and Kurt Schneider. Enablers, inhibitors, and perceptions of testing in novice software teams. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 30–40. ACM, 2014.

[13] Gregg Rothermel and Mary Jean Harrold. A Safe, Efficient Algorithm for Regression Test Selection. In *Proceedings of the International Conference on Software Maintenance (ICSM '93)*, pages 358–367. IEEE, September 1993.

[14] Per Runeson and Martin Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, 14(2):131–164, 2009.

[15] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.

[16] S. Yoo and M. Harman. Regression Testing Minimization, Selection and Prioritization: A Survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012. ISSN 1099-1689. doi: 10.1002/stvr.430. URL http://dx.doi.org/10.1002/stvr.430.

[17] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100. ACM, 2009.