Demystifying the Adoption of Behavior-Driven Development in Open Source Projects

Fiorella Zampetti, Andrea Di Sorbo, Corrado Aaron Visaggio, Gerardo Canfora, Massimiliano Di Penta*

Department of Engineering, University of Sannio, Italy

Abstract

Context: Behavior-Driven Development (BDD) features the capability, through appropriate domain-specific languages, of specifying acceptance test cases and making them executable. The availability of frameworks such as Cucumber or RSpec makes the application of BDD possible in practice. However, it is unclear to what extent developers use such frameworks, and whether they use them for actually performing BDD, or, instead, for other purposes such as unit testing.

Objective: In this paper, we conduct an empirical investigation about the use of BDD tools in open source, and how, when a BDD tool is in place, BDD specifications co-evolve with source code.

Method: Our investigation includes three different phases: (i) a large-scale analysis to understand the extent to which BDD frameworks are used in 50,000 popular open-source projects written in five programming languages; (ii) a study on the co-evolution of scenarios, fixtures and production code in a sample of 20 Ruby projects, through the Granger's causality test, and (iii) a survey with 31 developers to understand how they use BDD frameworks.

Results: Results of the study indicate that $\simeq$ 27% of the sampled projects use BDD frameworks, with a prevalence in Ruby projects (68%). In about 37% of the cases, we found a co-evolution between scenarios/fixtures and production code. Specifically, changes to scenarios and fixtures often happen together or

*Corresponding author
Email address: dipenta@unisannio.it (Massimiliano Di Penta)

after changes to source code. Moreover, survey respondents indicate that, while they understand the intended purpose of BDD frameworks, most of them write tests while/after coding rather than strictly applying BDD.

Conclusions: Even if the BDD frameworks usage is widespread among open source projects, in many cases they are used for different purposes such as unit testing activities. This mainly happens because developers felt BDD remains quite effort-prone, and its application goes beyond the simple adoption of a BDD framework.

Keywords: Behavior-Driven Development, Acceptance Testing, Empirical Study, Co-Evolution

---

1. Introduction

Behavior-Driven Development (BDD) entails the creation and usage of executable scenarios, written in natural language during the software analysis phases, to support software acceptance testing [1]. BDD is derived from Test-Driven Development (TDD) and, similarly to it, involves the creation of test cases before coding. As mentioned by North [2], BDD is based on the conjecture that: *"A story's behavior is simply its acceptance criteria —if the system fulfills all the acceptance criteria, it is behaving correctly; if it does not, it is not."* Differently from canonical testing approaches and TDD, BDD focuses more on "the specifications of the behavior of the class" (*i.e.,* specifications of the products' behavior) rather than on the "unit tests of a class". Also, TDD is enacted through test cases, written for example using frameworks like xUnit (*e.g.,* JUnit for Java), aimed at executing source code units. In BDD, test cases are scenarios—hereby also referred to as *specifications*— written using a natural language-based Domain Specific Language (DSL) in a form very similar to user stories. Such test cases describe the system's behavior from the outside. To make them executable, some glue code — hereby referred to as *fixture* — must be written to link scenarios to production code implementing the functionality to be tested.

The introduction of frameworks such as BEHAVE[1], CUCUMBER[2], JASMINE[3], JBEHAVE[4], MOCHA[5], or RSPEC[6] has made the use of BDD possible in practice. However, the level of adoption of BDD is still unclear. Also, it is unclear whether the aforementioned frameworks are actually used for BDD, or whether they are used for different purposes.

Researchers have previously studied the co-evolution of tests and code [3], as well as the extent to which developers are adopting TDD properly [4]. Also, early work indicated that the availability of acceptance tests (written in form of Framework for Integrated Test – Fit tables [5]) helps developers in better understanding requirements [6, 7, 8, 9] and improving the correctness of the developed programs [10]. However, to the best of our knowledge, there are no studies investigating the extent and the way BDD-related technology is being used in practice. Previous research mainly investigated the (i) strengths and limitations of the BDD tool support [1], (ii) the specific maintenance challenges deriving from the BDD adoption [11, 12], and (iii) possible solutions to reduce human efforts in maintaining specification files [13]. In summary, a question left unanswered by previous work is:

> *To what extent are BDD tools used, and, when they are used, do developers leverage them for actually performing BDD?*

The main motivation of this empirical study is to understand whether there is a gap between a technological hype — in this case, the idea, and its implementation in several frameworks, or making scenarios executable — and its adoption. That is, while BDD technology is out of there, and while previous research highlighted the benefits of such a technology, also in the context of acceptance testing [6, 7, 8, 9], the state-of-the-practice may provide a different

[1] `https://github.com/behave/behave`
[2] `https://cucumber.io`
[3] `https://jasmine.github.io/`
[4] `https://jbehave.org`
[5] `https://mochajs.org`
[6] `http://rspec.info`

picture. First, it is unclear the extent to which projects regularly adopt BDD tools and, when they do, whether this adoption is confined to a few developers, or is quite broad across the projects. Second, using BDD frameworks may or may not mean "scenario first" and not even "test first", because in the end such frameworks could be used as regular (unit) testing tools, and do not prevent developers for writing/modifying tests after production code. Last, but not least, developers may perceive benefits in using BDD tools, but may also perceive challenges in properly adopting them. In summary, understanding the *extent* and the *way* BDD tools are adopted and what benefits and challenges such adoption brings, requires a thorough empirical investigation.

To the best of our knowledge, this paper reports the first study aimed at investigating the adoption of BDD in practice. Following the motivations stated above, the study is carried out along three different dimensions, reflecting the three study research questions detailed in Section 3, and summarized in Figure 2. Namely, we study (i) the extent to which BDD tools are used by a large pool of 50,000 open source projects and by their developers, (ii) how developers change BDD scenarios, tests, and production code over time, and (iii) what is the developers' perception about BDD practices and technology. Note that the study is limited to the open-source because of the large availability of projects' data.

Results of the study indicate that BDD tools are used in 27.25% of 50,000 studied open-source projects developed in different programming languages, and that the percentage is high for Ruby projects (68%), and somewhat substantial for Javascript ($\simeq$ 39%) and Python ($\simeq$ 26%), while it is below 3% for the other languages. Except for Ruby, and in particular of developers using the RSpec framework, a very small percentage of developers ($< 7\%$) change BDD-related files. An in-depth analysis of a sample of 20 Ruby projects indicates that scenarios/source code co-evolve for about 40% of the analyzed pairs. In about 37% of the cases, we found a co-evolution between BDD scenarios (and related test fixtures) and production code traced onto them. Finally, 31 respondents of a survey we conducted indicate that, while they understood the intended purpose

of BDD tools, most of them write tests while/after coding rather than strictly applying BDD.

Paper structure. Section 2.1 provides background notions about BDD frameworks, explaining in particular how RSpec works, while Section 2.2 discusses the related literature. Section 3 describes the study definition and planning. Results are reported and discussed in Section 4, while the threats to its validity are discussed in Section 5. Section 6 discusses the study implications for researchers, practitioners, and educators, while Section 7 concludes the paper and outlines directions for future work.

The study dataset is available for replication purposes [14].

2. Background Notions and Related Work

This section first provides some background notions about Behavior-Driven Development (BDD). Then, it overviews related literature.

2.1. An overview of Behavior-Driven Development with RSpec

Similarly to Test-Driven Development (TDD), Behavior-Driven Development (BDD) is a test-first approach. In particular, both techniques leverage the assumption that generating test cases before implementing the software itself could help test engineers and developers in finding inconsistencies in the requirements specification and design documents earlier in the development process [15]. In BDD, the software design flow starts from the design and the implementation of the acceptance tests [16]. Such tests represent the means for communicating between software designers, developers and the other (non-technical) stakeholders. Indeed, for enabling the communication between the different types of actors, BDD augments TDD by forcing the use of natural language for describing automated acceptance tests. Acceptance tests collectively specify the user behaviors that the system has to fulfill [17] through the definition of a collection of *scenarios* and natural language represents the *ubiquitous language* [1] to describe such scenarios. Each scenario is described by defining at least three elements:

(i) a starting position (*e.g.,* navigate to a specific interface), (ii) the execution of a user action (*e.g.,* clicking a button), and (iii) the assertion to verify whether the consequent action has produced the expected effect.

Pragmatically speaking, BDD puts together the specification of an outcome relevant for an end-user (*e.g.,* role-feature matrix) and the examples and/or scenarios expressed with a single notation *given-when-then* implemented in DSLs (for example, CUCUMBER[7] uses the Gherkin DSL). That is, *given* some initial contexts, *when* an event occurs, *then* it ensures some outcomes. The latter (i) makes the behavior descriptions readable and easily accessible to domain experts, testers and developers, (ii) allows developers to associate the scenarios to the production code, and (iii) makes them executable (*i.e.,* provided that the acceptance criteria are executable). This is achieved through some glue code (*fixtures*), typically written using the same notation used for xUnit test cases, that links scenarios to production code under test. As a consequence of the test-first principle, when efficiently maintained, BDD tests also represent an accurate and up-to-date documentation source for the entire software system [18].

As described in the introduction, various BDD frameworks exist. Since for addressing one of our research questions we inspected projects (written in Ruby language) that use RSPEC for test automation, in the following it is convenient to specifically focus on RSPEC, to briefly describe how it works, though its metaphor is relatively similar to the other frameworks, with some differences (*e.g.,*CUCUMBER keeps scenarios separated from fixtures).

RSPEC allows developers to encapsulate the behavior under test via the `describe` block, to contextualize a set of tests by means of the `context` block, while tests are written using the `it` block. Moreover, RSPEC provides a way to set up test state using the `before` and `after` directives that will apply to all the tests included in the `describe` block.

Figure 1 shows a very simple RSPEC example aimed at testing the functionality of the *Bowling* class containing the method *hit*. The `describe` keyword

[7] `https://cucumber.io`

```
describe Bowling do
        context "with no strikes or spares" do
                it "sums the pin count for each roll'" do
                        bowling = Bowling.new
                        20.times { bowling.hit(4) }
                        expect(bowling.score).to eq 80
                end
        end
end

class Bowling
        attr_reader :score

        def initialize
                @score = 0
        end

        def hit(pin_count)
                @score += pin_count
        end
end
```

Figure 1: RSpec example.

takes the class name and requires a block argument containing the individual tests (*i.e.,* examples). The block argument is designated by the Ruby `do/end` keywords. The `context` provides a common setup to multiple tests, *e.g.,* those of the *Bowling* class in our case. After that, it is required to specify the `it` keyword aimed at defining a specific example that takes a string argument highlighting the expected behavior. The block argument belonging to the `it` clause contains a concrete specification of the test. Specifically, after having instantiated an object belonging to the *Bowling* class (*bowling*), the *hit* method is invoked 20 times. Finally, the `expect` statement defines the "expectation" checking whether a specific expected condition is satisfied. In our case, the code expects that the *score* attribute is equal to 80.

2.2. Studies on Behavior-Driven Development

While the characteristics of BDD, the related frameworks, and the strengths and limitations of the existing tool support have been first investigated by Solis

and Wang [1], to the best of our knowledge, no prior work studied how developers of open-source projects use BDD frameworks and the extent to which changes on specifications and the related production code might influence each other.

Recent research efforts have been devoted to the investigation of the specific challenges deriving from the BDD adoption in software projects. For instance, Binamungu *et al.* [11], by surveying 75 practitioners, investigated the challenges that may arise in maintaining the BDD specifications. They found that the management of BDD specifications over the long term can be challenging, as different team members could update such specifications multiple times and this could result in redundant specifications that are more costly to maintain. For this reason, in a different work, Binamungu *et al.* [19] presented an automated approach for detecting duplicate examples in BDD specifications. Rahman and Gao [12], instead, highlighted some of the maintenance issues that teams might face when adapting BDD tests to changing environments and proposed a reusable automated acceptance testing architecture to address reusability, and maintainability concerns.

Finally, Yang *et al.* [13] proposed a text-based approach to recommend when specification files should be modified, as a consequence of changes in the production code, for reducing the developers' effort in maintaining up-to-date specifications. Differently from Yang *et al.*, we rely on structural dependencies between spec files and production code, as explained in Section 3.1.

2.3. Studies on Acceptance Testing

Research on acceptance testing is relevant to our work because BDD tools can be used to automate acceptance testing, using test cases derived from use case scenarios or user stories.

Several researchers have studied the use of executable acceptance tests in software projects and in particular the use of Fit [5]. Melnik *et al.* [7] investigated, through controlled experiments, the use of Fit user acceptance tests for specifying functional requirements, highlighting that the use of Fit tables and the possibility to execute them improve the comprehension of requirements. In a follow-up

study, Melnik *et al.* [6] investigated whether acceptance tests can be authored effectively by customers of agile projects. Their results did not support the hypothesis that the quality of executable acceptance tests is positively correlated with the quality of production code. At the same time, results indicated how acceptance tests help customers to specify functional requirements.

From a different perspective, Ricca *et al.* [9] assessed the impact of Fit tables on the clarity of requirements, but also, in a different work, Ricca *et al.* [10] highlighted how the use of Fit tables during software development helps to improve the correctness of the produced code, without significantly affecting the productivity. Results consistent to those of Ricca *et al.* were also obtained by Park and Maurer [8] in a case study with custom-built executable acceptance test cases. Specifically, their study showed how acceptance test cases are useful to communicate requirements and to improve developers' productivity.

Using automated acceptance testing can be effort-prone for testers, as also pointed out by respondents to our survey. A study by Monden *et al.* [20] investigated, through a simulation model, the cost-effectiveness of testing (and in particular acceptance testing) activities, indicating that the testing effort can be reduced of 25% if accurate defect prediction is being used.

Our study substantially diverges from previous studies on acceptance testing. While we do not specifically look for benefits of executable acceptance testing, our study shows that since the early adoption of executable acceptance testing (described in the aforementioned related work), the available technology has gained maturity and adoption. At the same time, it is not only used to realize the BDD paradigm, but also as a support for traditional unit testing. Our result is consistent with results of previous studies showing how acceptance tests can help improving productivity [8] and even code correctness [10].

2.4. Studies on the Co-evolution of testing and production code

Zaidman *et al.* [3] studied the co-evolution of production and test code, by mining software repositories to better understand the testing processes that are followed in practice. In a subsequent study, Zaidman *et al.* [21] experimented

the use of three views, namely *Change History View*, *Growth History View* and *Test Coverage Evolution View* in two open-source and one industrial software project, to study how test and production code co-evolve over time. Their results indicated that, through the analysis of these complementary views, it is possible to recognize different co-evolution scenarios (*i.e.,* synchronous and phased).

Lubsen *et al.* [22] presented an approach based on association rule mining to determine whether production and test code co-evolve synchronously, and evaluated the proposed approach on an open-source and an industrial project, showing its effectiveness in recognizing the testing approach followed by software developers. Association rule mining has been also exploited by Marsavina *et al.* [23], who investigated co-evolution patterns of production and test code, and identified six patterns reflecting the presence (or the lack of) co-evolution.

Beller *et al.* [4] monitored the development activity of 416 software engineers over five months, observing that the majority of developers do not practice testing actively. Moreover, they evaluated the co-evolution between test and production code to demonstrate that TDD is not widely practiced, and developers who claim to apply it in many cases do not properly follow TDD principles.

Canfora *et al.* [24] performed a TDD experiment with professionals, finding that while TDD improves the quality of tests, it significantly slows down the whole testing process.

Our study on BDD adoption is inspired by previous research. While we share with Beller *et al.* [4] the somewhat negative results about the proper application of BDD (TDD in their case), our study is focused on the co-evolution between BDD specifications and production code. Also, and differently from previous work, we conduct an analysis in the large for assessing the extent to which BDD frameworks are used in open-source projects and, from a methodological perspective, employ the Granger causality test [25] to assess the test-production code pairs that statistically exhibit co-evolution patterns.

3. Study Design

The *goal* of this study is to investigate the adoption of BDD-related technology in open-source projects. The *perspective* is of researchers interested in understanding whether and the extent to which BDD is adopted in open-source software development, and whether BDD-related technology is used for different purposes. The study *context* consists of open-source projects hosted on GitHub (*i.e.,* 50,000 for the first study and a sample of 20 Ruby projects for the second study). Moreover (in RQ_3), to empirically validate the achieved results, we conduct a survey — collecting a total of 31 responses — in which we ask developers the kinds of BDD frameworks they use, and the main purpose for using them, *i.e.,* performing BDD, or rather using the frameworks for TDD or generally for other testing purposes.

Following the general goal of this study, and also what was discussed in the introduction, to understand BDD practices, we first need to investigate the precondition to any possible BDD practice, *i.e.,* whether BDD tools are used by projects, or not. Also, while BDD tools are being used by projects, this may or may not reflect a broad adoption of BDD practices by all developers (*i.e.,* only some of them use BDD tools). Therefore, our first research question is:

> **RQ_1** *To what extent do open source projects use BDD-related frameworks?*

Of course, using a BDD framework does not necessarily mean that developers of such projects were strictly following BDD. It is important to understand whether BDD specifications are created before, together or after their related production code, and how they co-evolve. To this aim, our second research question is stated as follows:

> **RQ_2** *How do* RSpec *specifications co-evolve with source code?*

The first two research questions provide a view of BDD practice adoption as it is reflected from the projects' repositories. However, it does not tell much about

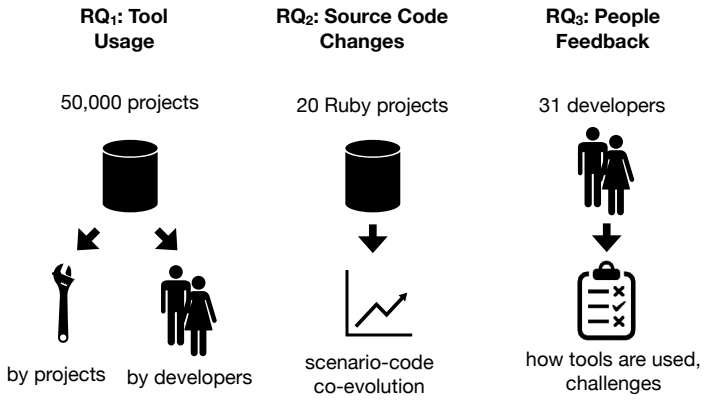

Figure 2: Study Overview.

developers' perception of BDD usefulness, about the way they are using BDD tools, challenges they encounter, and limitations of the used tools as perceived by them. For these reasons, we would like to hear directly from developers about their direct experience in using BDD tools. To this aim, we ask our third and last research question:

RQ$_3$ *How do developers use BDD-related frameworks?*

Figure 2 provides an overview of how the three research questions, together, contribute to the understanding of the BDD practice, analyzing the tool usage (RQ$_1$), changes to BDD-related resources and production code over time (RQ$_2$), and developers' perception (RQ$_3$).

3.1. Context Selection and Analysis Methodology

In the following, for each research question, we describe the context selection and the analysis methodology. Note that the context slightly varies because, for instance, while in **RQ**$_1$ we aim at obtaining a broad overview of the adoption of BDD tools, in **RQ**$_2$ we study the phenomenon with a deeper degree of detail (on a fairly-limited number and kinds of projects we could analyze), and, finally, in **RQ**$_3$ we are interested in getting feedback from developers.

To address **RQ**$_1$, we require a large set of projects, written in different programming languages, that may or may not use BDD tools. While performing

Table 1: BDD tools considered.

Name	Website	Supported languages
Behave	`https://github.com/behave/behave`	Python
Cucumber	`https://cucumber.io`	All programming languages
Jasmine	`https://jasmine.github.io/`	Javascript
JBehave	`https://jbehave.org`	Java
RSpec	`http://rspec.info`	Ruby
Mocha	`https://mochajs.org`	Javascript

such an analysis also for closed source is desirable, doing it on a large scale, and on a diverse set of projects (in terms of programming language, size, domain) is only feasible for open-source software projects. For this reason, we decided to conduct the study on open-source projects hosted on GitHub.

We analyze data from the top 50,000 projects — ranked in terms of the number of stars — hosted on GitHub for the five most popular programming languages on GitHub, *i.e.,* Java, Javascript, PHP, Python, and Ruby. As a further means to avoid toy projects, we only consider projects having at least one fork. Also, we exclude forked projects (to avoid duplicate data), and inactive projects (no changes after December 31st 2017). The selected projects have a median (i) size of 4,6 KLOC, (ii) number of stars of 677 and (iii) number of forks of 256.5. We considered 10,000 projects for each language, because a similar sample allows us to obtain high representativeness (*i.e.,* a margin of error less than 1% with a 95% confidence level) even for very large populations. Then, we identify, for each programming language, what are the BDD frameworks typically being used. This analysis has been performed by means of web searches. In the end, we identified six different tools described in Table 1, *i.e.,*Behave, Cucumber, Jasmine, JBehave, Mocha, and RSpec.

Clearly, it is possible that during our investigation, we could have missed some relevant BDD-related tools, as we based our analysis on those mentioned in previous research [1] and white papers about BDD. However, the selected tools are in-line with those used by our survey respondents (see RQ_3).

For each analyzed project, we look at imports/external dependencies into source code or configuration files, to determine the adoption (or not) of one (or more) of the identified BDD tools. In particular, as BDD frameworks need to be imported as libraries into software projects to use their testing automation features, we implemented a script able to (i) inspect the external dependencies (that could be defined in configuration files, *e.g.,* `pom.xml`) and the import statements directly occurring into source code files (*e.g.,* the statement `from behave import` occurs in Python programs using BEHAVE), and (ii) automatically check the BDD tools used.

We report, for each programming language, the number and percentage of projects using at least one BDD-related tool, as well as the number of projects using the considered tools (or even more than one tool).

After that, we identify the developers changing BDD-related files. To this aim, we clone the repositories of all projects using at least one BDD framework, and analyze their change history, including all branches while excluding merge commits (to avoid analyzing the same change twice). Then, we look at (i) commit authors and (ii) file paths being modified in each commit. From the latter, we use regular expressions to determine whether a commit (and consequently its author) has changed BDD-related files. For instance, for CUCUMBER and Java projects we have looked at the main programming practices used to write testing features and we have defined different regular expressions aimed at verifying whether or not a file modified is a BDD-related file. Specifically, we check (i) whether the file has extension `.feature`, or (ii) the file is in a `test` folder and its name ends with `Steps.java`, or (iii) the file is a Java file inside a `steps` or `step_definition` folder. Regarding RSPEC, instead, we check (i) whether the file is a Ruby file inside a `spec` or `test` folder, and (ii) whether its name contains the `spec` keywords. Since an author could use different emails (or IDs) within a project, we use a simple heuristic to unify them: if two entries have the same git ID, but different email, or the same email but different IDs, they belong to the same identity. As a result, we report, for each programming language and for each BDD tool, the average percentage of developers that change BDD-related

files.

To address $\mathbf{RQ}_2$, we look at how specifications and production code co-evolve to determine the extent to which tools are actually used to perform BDD. Since this is a deeper analysis (which is also technology- and language-dependent) we decided to focus on a sample of Ruby projects only, based on the results of $\mathbf{RQ}_1$ (see Section 4.1), which show that BDD is more adopted in Ruby than in other languages, and that RSpec is by large the most adopted BDD tool for Ruby.

We therefore randomly select a sample of 20 Ruby projects (detailed in Table 2), using RSpec and Travis-CI as Continuous Integration (CI) platform. For each project, we report the number of commits, commit authors and specification files. The reason behind the inclusion of projects using Travis-CI is twofold. On the one side, we wanted to guarantee that developers use BDD-frameworks to fail the build process in the presence of test failures. On the other side, as reported by North [2], BDD has grown to encompass the wider picture of agile analysis and automated acceptance testing. However, it is important to note that the use of CI does not necessarily imply a co-evolution of BDD scenarios and test fixtures. Indeed, a BDD scenario could be added before it is linked to production code through a fixture. At the same time, a developer could add a new feature in the production code without having created any tests for it. Finally, the only case for which we expect that test and production code always co-change is when a feature changes its behavior or a bug is fixed.

The selected projects (see Table 2) cover different application domains, from frameworks for building and releasing (*e.g., capistrano* and *fastlane*) to support frameworks for testing activities (*e.g., capybara* and *vcr*) to utilities frameworks like *httparty*, a HTTP library for Ruby, or *spree*, an e-commerce solution for Ruby on Rails. Moreover, the involved projects are different in terms of the number of specification files varying from 7 for *middleman* to 1,230 for *ruby*.

Due to the specificity of the analysis, we intendedly focus on a fairly limited set of projects, also for showing detailed results obtained in each studied project. Also, as Table 2 shows, the analysis involves nearly 200k commits, over 3,600 spec files and, as reported in Table 7 and Table 8, 8,780 spec-code pairs. Indeed, for

Table 2: Characteristics of projects selected to address $\mathbf{RQ}_2$.

Repository	**Commits**	**Contrib.**	**Spec Files**	**Files (*.rb)**
Homebrew/brew	17,513	634	318	1397
capistrano/capistrano	1,624	217	30	86
teamcapybara/capybara	3,420	248	41	186
carrierwaveuploader/carrierwave	2,214	309	29	84
freeCodeCamp/devdocs	2,748	82	28	546
diaspora/diaspora	19,737	341	228	753
discourse/discourse	30,808	664	382	2661
thoughtbot/factory_bot	990	175	42	124
fastlane/fastlane	14,794	905	316	901
guard/guard	2,396	151	52	119
jnunemaker/httparty	1,153	169	14	62
huginn/huginn	3,072	168	120	419
middleman/middleman	3,294	181	7	221
thoughtbot/paperclip	1,966	376	48	143
rspec/rspec-rails	2,650	263	34	89
ruby/ruby	53,662	51	1,230	7,389
sinatra/sinatra	4,032	362	31	139
spree/spree	18,932	763	317	1,220
hashicorp/vagrant	11,746	866	344	1,216
vcr/vcr	1,997	107	27	89
Min	**990**	**51**	**7**	**62**
Median	**3,183**	**256**	**45**	**204**
Avg	**9,937**	**352**	**182**	**892**
Max	**53,662**	**905**	**1,230**	**7,389**
Total	**198,748**	**7,032**	**3,638**	**17,844**

each project, we first clone the repository and extract the change history log for each branch, excluding merge commits. After that, we build a traceability matrix looking at the dependencies between test fixture (*i.e.,* spec files) and production code. To this aim, we exploit the (i) Ruby library structure[8], and (ii) the RSpec naming conventions[9] to statically extract traceability links between "spec" files and production code. It is worth to point out that a manual inspection of the selected projects allowed us to verify that each of such projects complies with the aforementioned conventions. Specifically, for each project, we automatically inspect all the spec (or test) files to collect:

- The relative path of the spec (or test) file: the module under test will reside in a file having the same sub-path structure of the spec (or test) file and the same name without the *_spec* (or *_test*) modifier; if such a file exists in the project, a traceability link between the spec file and such a file is added;
- The file names (and the relative paths) included in the require or load clauses (*e.g., require "path/filename.rb"*): if a similar file (along with the specific path) exists in the project, a traceability link between the spec (or test) file and such file is added;
- The defined namespaces: for a given namespace defined in the spec (or test) file, *e.g., A::B::Class*, we check if a file with the `.rb` extension and a similar sub-path structure, *e.g., a/b/class.rb*, exists in the project (also considering snake case or camel case variants) and, in this case, a traceability link between the spec (or test) file and such file is added.

Then, based on the availability of traceability matrices and the change history, we identify, for each pair RSpec-production code files:

- Which one has been added into the system first, or whether they have been added together. In particular, for each pair P_i we analyze the commit

[8] `https://ryanlue.com/posts/2017-03-03-how-to-structure-a-library`
[9] `https://relishapp.com/rspec/rspec-rails/v/3-0/docs/directory-structure`

history to establish whether the specification file, S_i, has been added to the code base in the same commit (*i.e., simultaneously*) or in a prior (*i.e., before*) or subsequent (*i.e., after*) commit with respect to the commit in which the production code file, C_i, has been added.

- When such files change, we look at the type of change made on the test file distinguishing between (i) changes to specifications only (*i.e.,* only a "describe" or "it" clauses has been modified), (ii) changes to fixture code only (*i.e.,* modifying only the code leaving unchanged the behavior expressed in the specification), and (iii) changes occurred to both specifications and fixtures.

Based on the change history of the sampled projects, we report the number and percentage of commit authors that modify BDD-related files (*i.e.,* following the same approach used for RQ_1), and we verify as well how those values vary while looking at those commits that modify both specifications and fixtures.

To (statistically) analyze the co-evolution of each files' pair, we use the Granger's causality test [25], *i.e.,* a technique for determining whether a time series is useful in forecasting another one. The basic idea is that a cause cannot come after the effect implying that if a variable x affects a variable z, the former variable should help to improve the predictions of the latter.

Let $fs_i(t)$ be the time series of the cumulative changes occurred to the specification file fs_i at commit t. In other words, $fs_i(t) = 0$ if the file does not exist at t, $fs_i(t) = 1$ if t adds it, and then increases of 1 for each change. Similarly, let $fp_j(t)$ be the time series of a production code file traced onto $fs_i(t)$. The simplest bivariate Granger test between two time series, $fs_i(t)$ and $fp_j(t)$, uses the autoregressive specification [26], which consists of the following univariate (restricted model):

$$fp_j(t) = c_1 + \gamma_1 fp_i(t-1) + \gamma_2 fp_i(t-2) + \cdots + \gamma_p fp_i(t-p) + e(t) \quad (1)$$

and bivariate (unrestricted model):

$$fp_j(t) = c_1 + \gamma_1 fp_i(t-1) + \gamma_2 fp_i(t-2) + \dots$$
$$+ \gamma_p fp_i(t-p) + \beta_1 fp_i(t-1) + \beta_2 fp_i(t-2) + \dots + \beta_p fp_i(t-p) + e(t) \quad (2)$$

solved by estimating the ordinary least squares. In the equations, p is the lag length (*i.e.,* the delay, in terms of number of commits, between spec and production code update, or vice versa), which can be estimated with various criteria [26], and $e(t)$ is an independent random variable. To test whether $fs_j(t)$ Granger-causes $fp_j(t)$, the null hypothesis to reject (H_0: changes to fs_i do not Granger-cause changes to fp_j) is defined as:

$$H_0 : \beta_1 = \beta_2 = \dots = \beta_p = 0$$

In our work, we use the Akaike Information Criterion (AIC) [27] to estimate the lag p for each pair of time series (details about the obtained lags are in our replication package [14]). The AIC rewards models that achieve high goodness of fits, but penalizes the increase in the number of parameters, to avoid over-fitting.

In summary, for each pair (fs_i, fp_j) for which there exists a traceability link, we test the Granger's hypothesis. Consider that in presence of transitive dependencies (*i.e.,* multiple production files covered by the same specification file) we look at each pair, instead of considering the cumulative changes among them. Also, we test the Granger's causality in both directions. Since we have to perform multiple Granger's tests, we adjust the obtained p-values using the Benjamini-Hochberg procedure [28], which ranks p-values, keeping the largest p-value unaltered, multiplying the second largest by the number of p-values divided by the rank (*i.e.,* two), and treating the remaining ones similarly to the second.

Granger's test has been performed using the *lmtest* R package [29], while the AIC procedure used to estimate the lag belongs to the *vars* R package [30, 31].

Once having the time series, for each type of change we tested the Granger's hypothesis with the time series belonging to the production code. In other

words, we test (in both directions) the causality of changes (i) between specification+fixture changes and production code changes, and (ii) between fixture-only changes and production code changes. Note that we did not consider the co-evolution of specification-only changes with source code, as this happened in a very limited number of cases (a median of zero times, mean=0.3, and max<10 per specification file).

Finally, we use statistical procedures to determine (i) whether the co-evolution between tests and production code occurs more when the whole specification (description plus fixture) is changed or when only the fixture is changed, and (ii) whether changes to production code follow changes to test (*i.e.,* BDD- or TDD-like), or, instead, the other way around happens, or either they co-change. More specifically, we compare the number of co-evolution instances using the paired Wilcoxon rank-sign test [32] assuming a significance level $\alpha = 5\%$, and using the Cliff's d effect size [33].

To address $\mathbf{RQ}_3$, we conducted a survey questionnaire. Previous work [34, 35, 36, 37, 38, 39] provided relevant guidelines on how to design and conduct survey studies in software engineering. Also, there exist broader guidelines for survey research in general [40].

Following these guidelines, we (i) designed *specific and measurable* goals, (ii) targeted subjects *able to answer* the questions posed (*i.e.,* subjects having testing experience), (iii) grouped questions into topics, and (iv) when possible, used *standardized response formats*. However, we are unable to assess whether the set of respondents to our survey is a *representative subset* of the target population, as we do not have prior knowledge of such a population and collected feedback from developers enrolled in online forums.

The questionnaire consists of two sections. The first section asks demographics' questions, *i.e.,* the respondent's maximum education degree, her experience in programming, testing, and the programming languages being used.

The second section features four questions. The first one asks which BDD frameworks are being used (if any). The second question asks whether they write BDD specifications to model the system behavior before starting to code,

whether they are used in a test-first manner (like TDD), along with coding, or after. The third question asks the main purpose of using BDD frameworks. Finally, the last question is about further comments the respondent may have.

We created the questionnaire using a Google Form, and distributed it through different channels, namely Twitter, Linkedin, and Reddit channels about testing and Ruby, namely *BDD*, *TDD*, *softwaretesting*, *test*, and *Ruby*. Moreover, as the scraping of personal information for spamming purposes explicitly violates the GitHub's terms of service[10], we have also looked for the development mailing lists/forums related to the Ruby projects sampled for answering our $\mathbf{RQ}_2$ and posted our questionnaire onto the mailing lists/forums related to 14 of these projects, intending to also receive feedback from developers of such projects. For the remaining 6 projects, we were not able to find any mentions of similar development communication channels. 31 replies from respondents with (self-rated) experience in testing activities from moderate to high were collected and analyzed. It is worth to point out that all the respondents to our survey own some college degree (Bachelor's, Master's, PhD or others) and the majority of them (*i.e.,* 16) are professional developers having more than 5 years of programming experience.

4. Study Results

In this section we report and discuss the main results obtained during our exploratory investigation.

4.1. To what extent do open source projects use BDD-related frameworks?

Table 3 reports the number and percentage of projects using BDD frameworks. Note that the "Overall" column is different from the sum of the other columns since it indicates the number of projects using at least one BDD framework, indeed there are projects using more than one BDD framework.

[10] `https://help.github.com/en/articles/github-terms-of-service#5-scraping`

Table 3: Usage of BDD tools in 50,000 projects (10,000 for each programming language) in GitHub.

Language	Behave	Cucumber	Jasmine	Jbehave	Mocha	Rspec	**Overall**
Java	✗	78 0.78%	✗	57 0.57%	✗	✗	125 1.25%
Javascript	✗	39 0.39%	1,206 12.06%	✗	2,915 29.15%	✗	3,902 39.02%
Python	2,522 25.22%	108 1.08%	✗	✗	✗	✗	2,578 25.78%
PHP	✗	226 2.26%	✗	✗	✗	✗	226 2.26%
Ruby	✗	706 7.06%	✗	✗	✗	6,735 67.35%	6,794 67.94%
Overall	2,522 5.04%	1,157 2.31%	1,206 2.41%	57 0.11%	2,915 5.83%	6,735 13.47%	13,625 27.25%

As the table shows, the percentage is low for Java and PHP, 1.25% and 2.26% respectively. Instead, the percentage grows substantially for Python projects (25.78%, mostly using Behave), Javascript projects (≃39%, mostly using Mocha) and, above all, for Ruby (67.94%). In this case, not only we can notice a large majority of projects using RSpec (67.35%), but, also, we notice that 7.06% of the analyzed projects also use Cucumber (the percentage of projects using Cucumber is 2.26% or below for the other programming languages).

As mentioned in Section 2.1, RSpec is a framework conceived for BDD, but it can also be used as a generic xUnit framework for Ruby, although other specific tools also exist for that purpose, see for example Minitest[11]. A more detailed analysis of how RSpec is being used in the open-source community is performed in $\mathbf{RQ}_2$.

To deepen our analysis about BDD tool usage, we investigate how many developers (commit authors) change BDD-related files. Table 4 reports, for

[11] `https://github.com/seattlerb/minitest`

Table 4: Percentage of commit authors that changed BDD-related files.

Language	Behave	Cucumber	Jasmine	Jbehave	Mocha	Rspec
Java	✗	2.62%	✗	1.76%	✗	✗
Javascript	✗	0.32%	5.4%	✗	2.77%	✗
Python	6.64%	4.91%	✗	✗	✗	✗
PHP	✗	4.55%	✗	✗	✗	✗
Ruby	✗	6.09%	✗	✗	✗	20.58%

each BDD-tool and for each language, the percentage of developers that modify BDD-related files. Unsurprisingly, except for Rspec, the percentage is always lower than 7% meaning that, only a limited number of developers change BDD-related files. The latter suggests that when a team introduces specific frameworks for following BDD practices only a few developers have the responsibility to create and modify them. For Ruby projects using Rspec, the percentage is substantially greater than for other languages and tools, and it reaches 20.58%. We conjecture that this happens because, very often, Ruby developers tend to use a specific BDD framework not only to follow BDD practices but rather for other purposes, *e.g.,* to perform unit testing.

> While the studied BDD frameworks are rarely used in Java and PHP projects, there is a substantial proportion of Python ($\simeq$ 26%) and Javascript projects ($\simeq$ 39%) using BDD frameworks, and above all a large majority of Ruby projects ($\simeq$ 68%) using RSpec, but also Cucumber. Looking at developers, less than 7% of the total number of developers change BDD-related files, except for Ruby projects using RSpec, where this percentage is of 20.58%.

4.2. How do RSpec *specifications co-evolve with source code?*

Table 5 reports, for each analyzed project, the number (and percentage) of times in which (i) a specification file has been added to the code base before the related production code (referred as **SAB** in the table), (ii) a specification file and its related production code file have been added to the code base simultaneously

Table 5: Specification files additions: Spec Added Before (SAB), Spec Added Simultaneously (SAS), Spec Added After (SAA).

Project name	**SAB (#)**	**SAB (%)**	**SAS (#)**	**SAS (%)**	**SAA (#)**	**SAA (%)**
brew	324	39.08	140	16.89	365	44.03
capistrano	2	3.64	22	40.00	31	56.36
capybara	10	18.18	9	16.67	35	64.81
carrierwave	6	10.53	10	17.54	41	71.93
devdocs	0	0	53	96.36	2	3.64
diaspora	65	19.58	148	44.58	119	35.84
discourse	101	18.57	236	43.38	207	38.05
factory_bot	13	20.31	47	73.44	4	6.25
fastlane	76	11.57	263	40.03	318	48.42
guard	55	24.12	48	21.05	125	54.82
httparty	11	37.93	10	34.48	8	27.59
huginn	5	3.52	92	64.79	45	31.69
middleman	0	0	5	45.45	6	54.55
paperclip	0	0	2	3.17	61	96.83
rspec-rails	6	14.63	7	17.07	28	68.29
ruby	588	14.04	220	5.25	3,381	80.71
spree	125	15.57	250	31.13	428	53.30
vagrant	33	6.16	121	22.57	382	71.27
vcr	0	0	0	0	54	100.00
Min	**0**	**0**	**0**	**0**	**2**	**3.64**
Median	**10.50**	**12.81**	**47.50**	**32.81**	**49.50**	**54.69**
Avg	**71**	**12.87**	**84.85**	**33.58**	**283.15**	**53.53**
Max	**588**	**39.08**	**263**	**96.36**	**3,381**	**100**

(**SAS**), and (iii) a specification file has been added to the code base after its related production code (**SAA**).

Although using BDD frameworks (*i.e.,*RSpec), it seems that developers often avoid putting into practice the test-first principle of BDD. Indeed, for all the analyzed projects, the majority of the specification files are introduced into the code base with or after the related production code. Moreover, 5 out of 20 projects (*carrierwave, middleman, paperclip, sinatra*, and *vcr*) have no cases in which the specification files have been introduced before the related production code. Looking deeper into these cases (*i.e.,* specifications introduced before the production code), we found that this frequently occurs when (i) the production

code files are renamed[12][13], and (ii) the dependency between the specification file and the production code file is established when new specifications or test cases are introduced into the specification file, as a result of improvements in the production code[14][15].

It is important to note that by "addition before" or "addition after" we mean at any time in the observed project duration. Additions of either specification or production code files may occur at different development stages. In particular, a file added years after a spec file may have been linked to the specification long after its creation. Similarly, specification files could have been introduced long after the related production code, as RSpec has been added during the project, and, consequently, not used for BDD purposes on the already existing production code. In summary, the addition of specification and production code is not necessarily aligned. Instead, as we will observe in the rest of the section, it is more relevant to analyze how they co-evolve once having been added.

Based on the change analysis performed, and based on the results obtained in $\mathbf{RQ}_1$, we now refine (on the 20 Ruby projects) the analysis concerning the percentage of developers that used BDD tools. Table 6 reports, for each project, (i) the number of developers (unique commit authors in the observed history), (ii) the number and percentage of those modifying RSpec files (third column, as done in $\mathbf{RQ}_1$), and, (iii) as a fine-grained analysis, the number and percentage of those who, in doing so, modified RSpec scenarios (features) at least once (fourth column).

Concerning the percentage of developers that change RSpec files, it varies in the range [1.6%-48.4%]. While there are a few projects in which RSpec files are modified by a limited number of developers (*i.e.,* only 5 projects have a percentage $< 15\%$), in all the other cases that percentage increases (*i.e.,* 15 projects have a percentage $> 15\%$). The fourth column in Table 6 reports the number and

[12] `https://github.com/jnunemaker/httparty/commit/4c6514f`
[13] `https://github.com/teamcapybara/capybara/commit/6b099a4`
[14] `https://github.com/jnunemaker/httparty/commit/d0d88fe`
[15] `https://github.com/teamcapybara/capybara/commit/db1bbd2`

Table 6: RQ_2: Percentage of commit authors that change the BDD-related files.

Project Name	**# Dev.**	**Changing** RSpec **files**	**(%)**	**Changing scenarios**	**(%)**
brew	828	51	(6.2%)	48	(5.8%)
capistrano	426	106	(24.9%)	52	(12.2%)
capybara	338	124	(36.7%)	62	(18.3%)
carrierwave	410	125	(30.5%)	119	(29.0%)
devdocs	128	2	(1.6%)	2	(1.6%)
diaspora	502	118	(23.5%)	118	(23.5%)
discourse	923	147	(15.9%)	147	(15.9%)
factory_bot	247	69	(27.9%)	25	(10.1%)
fastlane	1162	308	(26.5%)	194	(16.7%)
guard	186	55	(29.6%)	13	(7.0%)
httparty	215	104	(48.4%)	90	(41.9%)
huginn	213	47	(22.1%)	47	(22.1%)
middleman	221	6	(2.7%)	6	(2.7%)
paperclip	453	188	(41.5%)	60	(13.2%)
rspec-rails	322	57	(17.7%)	54	(16.8%)
ruby	276	17	(6.2%)	10	(3.6%)
sinatra	451	112	(24.8%)	32	(7.1%)
spree	1065	245	(23.0%)	227	(21.3%)
vagrant	1113	139	(12.5%)	139	(12.5%)
vcr	140	49	(28.6%)	10	(7.1%)

percentage of developers that changed scenarios at least once while modifying RSpec tests. As the reader can notice, in 7 projects (*capistrano*, *capybara*, *factory_bot*, *guard*, *paperclip*, *sinatra*, and *vcr*) the percentage substantially drops (of 50% or more, compared to the third column). In other words, only a minority of developers who changed test cases also modify the scenarios. Therefore, the rest of the developers changing tests are using RSpec as a unit testing tool. In the following, by looking at the co-evolution it is possible to gain more insights into how developers use BDD-related frameworks.

Tables 7 and 8 summarize the results of the Granger's Causality test related to the co-evolution of specification files and their related production code. As explained in Section 3, we split the analysis between changes that involved the whole specification (Table 7) and changes to fixture only (Table 8). The tables

Table 7: Granger Causality Test: summary of results (changes to both scenarios and test code).

Proj.	**Pairs**	**Exact Match**	**Code→ Spec**	**Spec→ Code**	**Both**	**Overall**	**(%)**
brew	829	90	110	80	190	380	45.84
capistrano	55	4	12	11	16	39	70.91
capybara	54	0	12	9	4	25	46.30
carrierwave	57	2	8	13	6	27	47.37
devdocs	55	7	7	7	11	25	45.45
diaspora	332	8	99	48	96	243	73.19
discourse	544	17	123	81	129	333	61.21
factory_bot	64	6	21	11	8	40	62.50
fastlane	657	3	124	65	52	241	36.68
guard	228	0	20	41	1	62	27.19
httparty	29	1	5	3	5	13	44.83
huginn	142	8	32	12	67	111	78.17
middleman	11	2	3	1	3	7	63.64
paperclip	63	1	7	11	3	21	33.33
rspec-rails	41	2	9	6	9	24	58.54
ruby	4,189	69	620	122	202	944	22.54
sinatra	37	2	9	6	5	20	54.05
spree	803	13	102	91	157	350	43.59
vagrant	536	23	132	53	183	368	68.66
vcr	54	0	2	1	0	3	5.56
Min	**11**	**0**	**2**	**1**	**0**	**3**	**5.56**
Median	**63.50**	**3.50**	**16**	**11.50**	**10**	**39.50**	**46.84**
Avg	**439**	**12.90**	**72.85**	**33.60**	**57.35**	**163.80**	**49.48**
Max	**4,189**	**90**	**620**	**122**	**202**	**944**	**78.17**
Total	**8,780**	**258**	**1,457**	**672**	**1,147**	**3,276**	**37.31**

report, for each project (i) the number of overall pairs analyzed (**Pairs** column), (ii) the amount of pairs exhibiting a perfect co-evolution (**Exact Match** column), (iii) the number of statistically significant pairs for which changes on the specification file occur after the changes in the production code (**Code→Spec** column), (iv) the amount of statistically significant pairs for which changes on production

Table 8: Granger Causality Test: summary of results (changes to test code, scenarios unchanged).

Proj.	**Pairs**	**Exact Match**	**Code→ Spec**	**Spec→ Code**	**Both**	**Overall**	**(%)**
brew	829	1	71	40	23	134	16.16
capistrano	55	0	10	8	9	27	49.09
capybara	54	0	16	3	4	23	42.59
carrierwave	57	0	25	5	3	33	57.89
devdocs	55	0	1	4	0	5	9.09
diaspora	332	0	93	33	70	196	59.04
discourse	544	0	97	71	63	231	42.46
factory_bot	64	0	7	2	0	9	14.06
fastlane	657	1	59	93	29	181	27.55
guard	228	0	1	2	0	3	1.32
httparty	29	0	4	1	1	6	20.69
huginn	142	0	31	24	17	72	50.70
middleman	11	0	1	0	0	1	9.09
paperclip	63	0	10	5	2	17	26.98
rspec-rails	41	1	5	3	1	9	21.95
ruby	4,189	179	479	7	185	671	16.02
sinatra	37	0	3	3	7	13	35.14
spree	803	61	117	83	165	365	45.45
vagrant	536	0	73	60	74	207	38.62
vcr	54	0	3	2	0	5	9.26
Min	**11**	**0**	**1**	**0**	**0**	**1**	**1.32**
Median	**63.50**	**0**	**13**	**5**	**5.5**	**25**	**27.27**
Avg	**439**	**12.15**	**55.30**	**22.45**	**32.65**	**110.40**	**29.66**
Max	**4,189**	**179**	**479**	**93**	**185**	**671**	**59.04**
Total	**8,780**	**243**	**1,106**	**449**	**653**	**2,208**	**25.15**

code occur after changes in the related specification file (**Spec→Code** column), (v) the number of pairs for which changes in specification and production code files influence each other (*i.e.,* the null hypothesis of the Granger's Causality test is rejected in both directions, **Both** column), and (vi) the number (and percentage) of overall pairs exhibiting any statistically significant causality relationship

(*i.e.,* Exact Match+Code→Spec+Spec→Code+Both, **Overall** column).

Looking at the quantitative results achieved for changes affecting both specifications and fixtures (Table 7), we found that, for 9 out of 20 projects, more than 54% of the considered specification-production code pairs exhibit causality relationships. However, a comparison of the number of co-evolution pairs (*i.e.,* the **Overall** column in Table 7 and Table 8) reveals that there is significantly more co-evolution when both specifications and fixtures are changed, than for changes affecting fixtures only (p-value=0.0009, $d = 0.26$ - small). The latter implies that even when using a BDD framework, the feature addition and/or modification results in modifying both production and test code.

For what concerns the co-evolution of specifications and fixtures together with production code (Table 7), we found that the number of co-evolving pairs where specifications/fixtures are changed together (**Exact Match** and **Both** columns) or after the production code (**Code→Spec** column) is significantly greater than the number of co-evolving pairs where specifications/fixtures change before production code (**Spec→Code** column). The paired Wilcoxon signed-rank test returned a p-value=0.0005, and the Cliff's $d = 0.4$ – medium. Similar results were obtained when analyzing the co-evolution of changes to fixtures only with production code (p-value=0.0005, d=0.33 - small).

Although some differences can be observed across projects, the median lag between changes of specification and source code (in a direction or in the other) is of only 3 commits. This clearly contrasts with distances observed in the additions discussed above. That is, while specifications and related source code files can be added at a distance of years (for the reasons explained above), once they are in, they exhibit a close co-change, either in a direction (specification before code) or in the other (code before specification).

A manual inspection of pairs exhibiting statistically significant co-evolution values indicates that very often the additions and/or modifications of test cases are clearly driven by changes applied to production code, hence violating the BDD principles. As an example, we report a case occurring in the *middleman* project, in which a new production file is added to address a pull request aimed at

implementing new piece of functionality. Before merging the pull request, there is a comment asking for properly testing the implemented functionality[16]. Once created, the production and test files are changed together as a consequence of feature improvements, changes, or refactoring operations. Similar results have been identified for the *httparty* project where the modifications to the production code file `lib/httparty/request.rb`[17][18] are very often followed by changes (in subsequent commits) introducing new testing scenarios within the spec file `spec/httparty/request_spec.rb`[19][20].

In other cases, it happens that after some changes occurred on both test and production files, developers start modifying only the fixtures by applying refactoring operations or test code improvements. For instance, in the *huginn* project, a commit *"to increase test stability"*[21] can be noticed. Finally, by looking at cases in which developers change the fixtures only, we found that this is done in order to make tests passing on Travis-CI by (i) accounting for implementation details not contemplated in the test case yet[22], or (ii) by simply skipping the test case, *e.g.,* when it failed for unknown reasons[23].

Even if it seems that developers do not follow the BDD practice, we found a few cases in which the specifications included in the test files have been used to guide functional implementation. For instance, for the *middleman* project we report a commit[24] in which the addition of a set of example groups is aimed at testing a specific feature previously implemented. After that, the developers realize that a conditional branch included in the test specifications was not implemented at all in the production file, and, thus, the production code file is

[16] `https://github.com/middleman/middleman/commit/9ba1dc04`
[17] `https://github.com/jnunemaker/httparty/commit/50c46e2`
[18] `https://github.com/jnunemaker/httparty/commit/0f606b2`
[19] `https://github.com/jnunemaker/httparty/commit/604bfa6`
[20] `https://github.com/jnunemaker/httparty/commit/dc14b30`
[21] `https://github.com/huginn/huginn/commit/771afe9`
[22] `https://github.com/teamcapybara/capybara/commit/573668c`
[23] `https://github.com/teamcapybara/capybara/commit/864fb62`
[24] `https://github.com/middleman/middleman/commit/49a7435`

subsequently modified to implement the missing functionality[25].

In general, it may be possible that developers modify only the specifications without altering the fixtures. However, looking at those cases we found many situations in which changes were aimed at (i) addressing compatibility issues[26], (ii) updating to a new Ruby version syntax[27], or (iii) fixing typos or spelling issues in the description of the behavior[28].

> BDD scenarios are almost always introduced in the code along with the related production code, or (even a long time) after the code has been added. BDD scenarios co-evolve with production code in up to 37% of the cases, but once again it is more likely that they change together (or after) production code than before (the median lag is of 3 commits). In essence, only in a minority of cases, the co-evolution analysis indicates symptoms of a likely application of BDD principles.

4.3. How do developers use BDD-related frameworks?

We received in total 35 responses, of which four were discarded as the respondents declared to have a low experience on testing. We, therefore, analyzed a total of 31 responses. Out of the 31 respondents, 19 own a Bachelor Degree (or some college degree), 9 a Master's degree, and 3 a Ph.D. 14 of them have more than 10 years of experience in programming, 8 between 5 and 10, and 9 less than 5. Most of them (21) are professional developers, but we got 2 responses from academics, 2 (Ph.D.) students, 2 industrial researchers, 2 IT managers, and 2 QA consultants. Their level of experience in testing activity is (self)-rated from moderate (14 of them) to high (17 of them).

The adoption of programming languages, with the exception of PHP, reflects quite well the choices of our study. 17 respondents indicated to develop in

[25] `https://github.com/middleman/middleman/commit/6ab6669`
[26] `https://github.com/vcr/vcr/commit/00f4309`
[27] `https://github.com/capistrano/capistrano/commit/2c14957`
[28] `https://github.com/middleman/middleman/commit/b42e4ed`

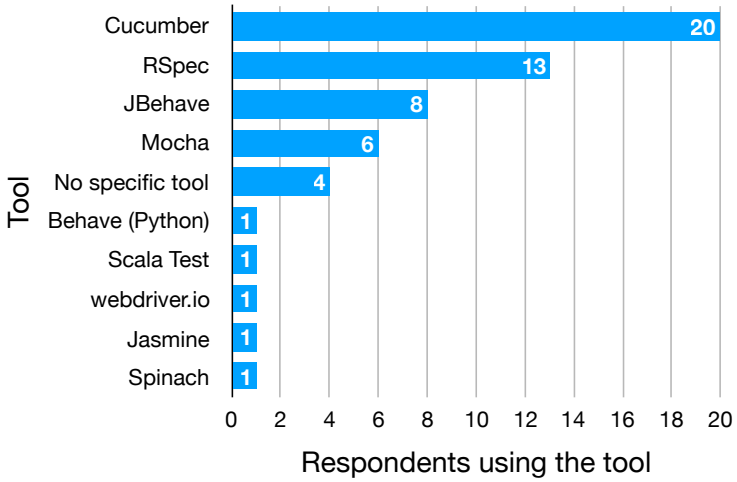

Figure 3: BDD frameworks used by respondents.

Javascript, 15 in Java, 15 in Ruby, 11 in Python, 2 in PHP, and one for each of the following languages: Kotlin, Scala, Objective-C, and Swift. Note that values do not sum up to 31 as respondents could indicate multiple languages.

As shown in Figure 3, in terms of frameworks being used, unsurprisingly, most of the respondents (20) indicated CUCUMBER, but this is also because CUCUMBER was mentioned as a framework being used by programmers working in various languages (all five we considered in our study). Nearly all Ruby programmers use RSPEC (13 out of 15), while Mocha is less frequent for Javascript programmers (only 6 out of 17), and JASMINE used by only one respondent. 8 out of 15 Java programmers indicated to use Jasmine. Finally, some developers mentioned specific frameworks we did not consider in $\mathbf{RQ}_1$, *i.e.,*SCALA TEST (as we did not study Scala), WEBDRIVER.IO (which however is a Web testing tool using the other frameworks we considered) and SPINACH. It is important to notice that, with the exception of the last three frameworks, all others have been considered in our $\mathbf{RQ}_1$ investigation. Finally, our pool of respondents also included four respondents indicating that they did not use specific frameworks for BDD, a practice used in the past for automated acceptance test [6].

By looking on the testing practices, our study participants in principle agreed

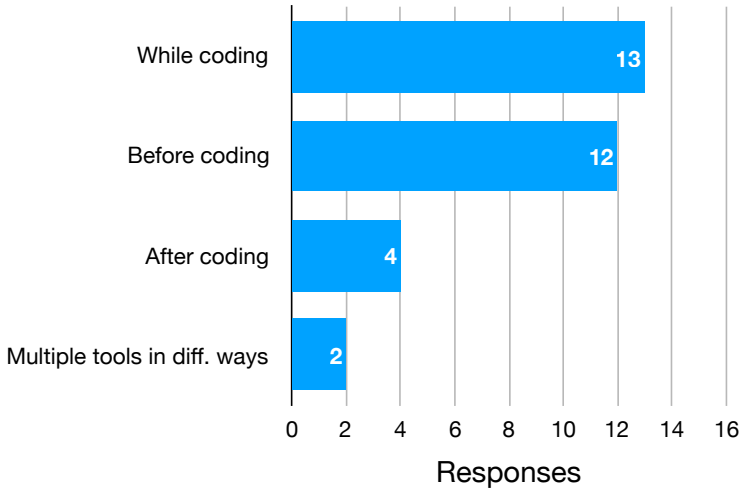

Figure 4: When test cases are written.

on the intended purpose of the BDD frameworks, *i.e.,* writing scenarios to model the system behavior before coding, and therefore performing BDD (17 responses) or, at least, performing TDD (5). However, there are also respondents using the frameworks for other testing purposes (6), or for multiple purposes (2).

However, when we asked when developers write test cases (Figure 4), only 12 of them indicated that they write test cases before coding. 13 respondents do it while coding and 4 after coding.

In the open comments[29], on the one side, some respondents highlight the difficulties in applying BDD in practice: *"it's just an overly complicated way to write unit tests"*. Also, and despite results of previous literature [6, 8, 9, 10], developers are skeptical about the advantages of BDD *"I have found that behavior and test-driven development results in less than optimal code quality and slower overall development with the benefit of a marginal improvement in bug reports."* On the other side, they point out how BDD DSLs enforce cleaner and consistent write-up of scenarios *"significance of using a DSL and the defined glossary is to*

[29]Due to the limited number of open comments, we simply report and discuss them, as coding would not make sense.

develop a shared understanding of the problem." Also, they highlight that writing specifications in a development team has, sometimes, different purpose than using them for testing: it can be misleading to talk about *"BDD testing tools, as writing and automating executable specifications is quite different to testing".* Moreover, one developer mentions how BDD is being used during evolution activities, including refactoring to automatically check regressions and/or that pre- and post-conditions still hold after refactoring. A different developer, instead, highlights that *"I more often write tests first when refactoring code or when working in scenarios where the programming interface is already well-defined. I more often write tests afterwards when what I need to write is less well-defined, requiring more exploratory coding."* In the former case, she is applying TDD instead of BDD, while in the latter case, despite the BDD approach may help in defining/clarifying the system behavior under specific conditions, the surveyed developer declares that she prefers starting with coding. Finally, there is a respondent highlighting that the scenarios are mainly written by the product owner instead of the developers *"The product owner (non-developer) writes the scenarios."*

> While respondents understood the purpose of the BDD frameworks they use, in practice many of them write tests while (or after) coding, hence they do not perform BDD and not even TDD. Respondents felt that while BDD frameworks enforce cleaner and more consistent specifications, the effort in adopting them might not be paid back by an improved code quality.

5. Threats To Validity

Threats to *construct validity* concern the relationship between theory and experimentation. In $\mathbf{RQ}_1$, while we used several heuristics to select projects that help us to avoid inactive projects, duplicates (forks), and projects without source code, it is still possible that our sample contains non-development projects. Considering that we also selected projects based on their number of commits, and

considering the large sample, we do not expect the percentage of such projects to be relevant so to introduce substantial noise in our results.

The heuristics used in $\mathbf{RQ}_1$ to detect the tool usages might be imprecise or erroneous. We mitigated this threat through manual analysis of a few projects for each programming language and BDD framework. Similarly, the fine-grained analysis of changes in $\mathbf{RQ}_2$ is based on `git diff` and regular expressions, and despite some tests of our scripts, imprecisions are still possible. Moreover, traceability links between specification and production files have been heuristically recovered by exploiting directory structures and file naming conventions. However, some links could have been missed or wrongly identified. To alleviate this concern we manually checked that all the selected projects comply with the aforementioned conventions. As for $\mathbf{RQ}_3$, the major threat is represented by the limited understanding of BDD by respondents. This is partially mitigated by the relatively high experience and expertise they declared, in particular about testing.

In $\mathbf{RQ}_2$, the observation of whether BDD is applied in projects is limited to the analysis of Granger causality. Further investigation might be required to verify whether the process is properly followed. In $\mathbf{RQ}_3$, we based our conclusions from respondents' feedback, which might deviate from their actual practice.

Threats to *internal validity* concern factors internal to our study that could influence our conclusions. Recent research [41] studied the perils of mining GitHub repositories for software engineering research, observing that the majority of projects are personal and inactive. To avoid considering similar projects (*i.e.,* personal and inactive) in $\mathbf{RQ}_1$, we selected the top 10,000 projects (in terms of stars) for each programming language, excluding forked projects, projects having no changes in the last year (*i.e.,* 2018), and projects having a single contributor.

As we explained in Section 3.1, we used a simple heuristic to unify identities from git repositories. Although more complex and reliable heuristics exist for this purpose [42, 43], but they are needed when merging identities from different, heterogeneous repositories.

While through Granger's test we establish a causal relationship on a statistical

basis, we cannot claim the presence of a cause-effect relationship between changes to tests and production code, or vice versa. We mitigate this threat with a qualitative analysis detailed in $\mathbf{RQ}_2$. In addition, similarly to previous work [3], to study the co-evolution of specification and production code, we rely on the data that is stored in the version control system. However, especially for additions/modifications of specifications and related production files that occur in the same commit, we can not establish which elements are added/modified first. This issue is partially mitigated by the empirical assessment carried out in our $\mathbf{RQ}_3$, in which the majority of survey respondents stated that they usually perform modifications to specifications while or after coding.

$\mathbf{RQ}_3$ is affected by the self-selection of its participants, which could have influenced the kinds of provided answers. Although we do not claim full representativeness of the universe of developers using BDD practices and tools, it is possible that those who responded were the ones more inclined towards BDD, or at least more competent. Finally, another possible threat for $\mathbf{RQ}_3$ is that projects might use multiple BDD tools over their evolutionary history. We have mitigated this threat by checking that, for the studied projects, RSpec was used across the entire observed history.

Threats to *conclusion validity* concern the relationship between experimentation and outcome. Wherever possible, in particular in $\mathbf{RQ}_2$, we base our conclusion on statistical procedures, including Granger's test, Wilcoxon signed rank test, and Cliff's delta effect size. Moreover, we cope with multiple tests by adjusting p-values with the Benjamini-Hochberg procedure [28].

Finally, threats to *external validity* concern the generalization of our findings. As for $\mathbf{RQ}_1$, although the sample is large (50,000 projects) and diverse in terms of programming languages, it might not fully represent the whole picture of the open-source community. More importantly, it does not represent the BDD practice in the industry, and in general in a context where there is a better formalization of scenarios than in open source projects where, often, features are added thanks to the contribution of external developers (*i.e.,* through pull requests). Also, in the search for used frameworks, we may have missed some

relevant ones, although we based our analysis on those mentioned in white papers about BDD and our choices are to a major extent in-line with our survey respondents. In addition, for $\mathbf{RQ}_2$ we intendedly decided to do a focused analysis on a fairly limited set of projects developed with a specific language (Ruby) and using a specific BDD framework (RSpec). Therefore, future studies on other languages and frameworks are needed. Finally, for $\mathbf{RQ}_3$ the sample of respondents is made up of experts (as they admitted) that well represent the technologies investigated in $\mathbf{RQ}_1$. However, a larger investigation on more subjects beyond our 31 respondents is surely desirable.

6. Implications

Results of our study have implications for developers, educators, and researchers.

For what concerns **developers**, our study shows that, while a plethora of BDD frameworks exist, their adoption has still a long way to go. Only a small percentage of projects use such tools and, more importantly, when this happens, it is done by a very small percentage of developers. In summary, most of the developers are still not ready (or not willing) to use such a relatively new piece of technology and its related development methodology.

Such low percentages also allow conjecturing that, in general, there is no application of BDD practices at project-level. While we do not have specific evidence of that, it could either happen that in a project only a few developers act as scenario-writers (but this contradicts agile principles fostering a shared knowledge and discouraging a clear separation of roles) or, more likely, BDD tools are opportunistically adopted by a limited number of developers, while the project has not set their usage as a recommended practice.

There are noticeable exceptions, like the Ruby language and its RSpec framework. However, a deeper observation made through a co-evolution analysis ($\mathbf{RQ}_2$) and a survey ($\mathbf{RQ}_3$) indicates that very often developers write/change BDD specifications while coding or even after that. Truly, BDD frameworks are

also suited for other testing purposes, however having a clear separation between acceptance tests and other (*e.g.,* unit, integration) tests is surely desirable [44, 45, 46].

As previous research has shown [6, 10], writing executable scenarios is still overkilling for developers. Therefore, while BDD-related technology is nowadays in place, and while the answers to our survey questionnaire indicate that in most cases the participants are knowledgeable about BDD, the study has shown that such a methodology is not exploited as it should be yet. Moreover, there could be a great potential in using BDD tools as a part of a DevOps Continuous Integration and Delivery (CI/CD) Pipeline, helping the team to monitor the development progress of scenarios assigned to a release. However, once again, the current BDD adoption level, and the way BDD tools are used, make such integration with CI/CD still limited to simply executing test cases, *i.e.,* like any other unit test framework.

To favor a wider and proper adoption of BDD tools, **educators** should properly introduce the BDD methodology when teaching agile development, requirement engineering, and testing, and possibly highlight its strengths, but also open challenges in its applications. This will not only increase the awareness about BDD tools but also let developers be aware of how such tools could properly be applied in the context of a development process (*i.e.,* not limiting their use as they were simple unit test tools). For example, since the results of $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$ indicated that, in many circumstances, developers prefer to write tests after the code, we argue that it is also important for educators to clearly illustrate, also with the support of real-world examples, how the BDD approach can help in clarifying/defining the system's behavior under specific conditions even when it is not yet completely specified, as in the case of open source development. Also, as explained above, educators could properly frame BDD in the context of CI/CD, as a way to monitor development.

Answers provided to our survey seem to indicate that, while developers are aware of how BDD works, the available tool support makes its application too overkilling. More specifically, survey respondents highlighted the difficulty and

the effort needed to write scenarios in the appropriate format, and in writing fixtures to link them onto production code. As a consequence, there are still many open areas on which **researchers** can put their effort to enhance the applicability of BDD in practice, especially with better tool support for writing scenarios (or converting existing scenarios and user stories in the format required by BDD tools), and automatically-generating fixtures.

Based on the obtained results, there is also room for further empirical investigations, for example, surveying projects and organizations encouraging BDD for their development, or studying the benefits of BDD in the context of CI/CD pipelines.

7. Conclusion

This paper has investigated the use of Behavior-Driven Development (BDD) frameworks in the open-source community. The study has been conducted by analyzing ($\mathbf{RQ}_1$) the adoption of BDD frameworks in 50,000 GitHub projects written in five different programming languages, ($\mathbf{RQ}_2$) the co-evolution of BDD specifications and production files in 20 Ruby projects using RSpec [47], and ($\mathbf{RQ}_3$) feedback provided by 31 developers about their knowledge and usage of BDD frameworks.

Results of the study indicate how the adoption of BDD tools is still limited, except for cases such as Ruby-RSpec where the tool is also used for unit testing. The co-evolution analysis we performed makes it evident how, in practice, developers do not write "scenarios-first", but they mostly use BDD tools as any other unit testing tool, and they mostly write tests together or after the related production code. The feedback we collected through the survey indicated how the current BDD technology is still too overkilled to be properly adopted in practice.

As work-in-progress, we are investigating the extent to which the introduction of BDD in an ongoing project has any effect on the product and process quality, but also the use of BDD in combination with other practices, such as Continuous

Integration and Delivery. In the future, we envision to carry out a quantitative and qualitative investigation on the costs and benefits deriving from the adoption of BDD-like approaches. In this context, since our research empirically pointed out the adoption of BDD-related frameworks such as RSpec also for non-BDD purposes, we also plan to shed some light on the motivations and practical advantages of using BDD-related tools in such cases.

References

[1] C. Solís, X. Wang, A study of the characteristics of behaviour driven development, in: 37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, 2011, pp. 383–387.

[2] D. North, How to sell bdd to the business, `https://skillsmatter.com/skillscasts/923-how-to-sell-bdd-to-the-business` (last access: 01/10/2019).

[3] A. Zaidman, B. Van Rompaey, S. Demeyer, A. van Deursen, Mining software repositories to study co-evolution of production & test code, in: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, 2008, pp. 220–229.

[4] M. Beller, G. Gousios, A. Panichella, A. 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, Bergamo, Italy, 2015, pp. 179–190.

[5] R. Mugridge, W. Cunningham, Fit for Developing Software: Framework for Integrated Tests., Prentice Hall, 2005.

[6] G. Melnik, F. Maurer, Multiple perspectives on executable acceptance test-driven development, in: Agile Processes in Software Engineering and Extreme Programming, 8th International Conference, XP 2007, Como, Italy, 2007, pp. 245–249.

[7] G. Melnik, K. Read, F. Maurer, Suitability of FIT user acceptance tests for specifying functional requirements: Developer perspective, in: Extreme Programming and Agile Methods - XP/Agile Universe 2004, 4th Conference on Extreme Programming and Agile Methods, Calgary, Canada, 2004, pp. 60–72.

[8] S. Park, F. Maurer, Communicating domain knowledge in executable acceptance test driven development, in: Agile Processes in Software Engineering and Extreme Programming, 10th International Conference, XP 2009, Pula, Sardinia, Italy, 2009, pp. 23–32.

[9] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, P. Tonella, Using acceptance tests as a support for clarifying requirements: A series of experiments, Information & Software Technology 51 (2) (2009) 270–283.

[10] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, M. Ceccato, C. A. Visaggio, Are fit tables really talking?: a series of experiments to understand whether fit tables are useful during evolution tasks, in: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, 2008, pp. 361–370.

[11] L. P. Binamungu, S. M. Embury, N. Konstantinou, Maintaining behaviour driven development specifications: Challenges and opportunities, in: 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, 2018, pp. 175–184.

[12] M. Rahman, J. Gao, A reusable automated acceptance testing architecture for microservices in behavior-driven development, in: 2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015, San Francisco Bay, CA, USA, 2015, pp. 321–325.

[13] A. Z. H. Yang, D. A. da Costa, Y. Zou, Predicting co-changes between functionality specifications and source code in behavior driven development, in: Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, Montreal, Canada., 2019, pp. 534–544.

[14] F. Zampetti, A. Di Sorbo, C. A. Visaggio, G. Canfora, M. Di Penta, Demystifying the use of behavior-driven development tools in open source projects. Replication Package `https://tinyurl.com/y9yacwyx`.

[15] G. S. Mahalakshmi, V. Vani, Theoretical verification of test cases for behavior driven development, in: 2017 Second International Conference on Recent Trends and Challenges in Computational Models (ICRTCCM), 2017, pp. 309–314.

[16] M. Soeken, R. Wille, R. Drechsler, Assisted behavior driven development using natural language processing, in: Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, 2012, pp. 269–287.

[17] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, S. Brinkkemper, D. Zowghi, Behavior-driven requirements traceability via automated acceptance tests, in: IEEE 25th International Requirements Engineering Conference Workshops, RE 2017 Workshops, Lisbon, Portugal, 2017, pp. 431–434.

[18] M. Wynne, A. Hellesoy, S. Tooke, The cucumber book: behaviour-driven development for testers and developers, Pragmatic Bookshelf, 2017.

[19] L. P. Binamungu, S. M. Embury, N. Konstantinou, Detecting duplicate examples in behaviour driven development specifications, in: 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, 2018, Campobasso, Italy, 2018, pp. 6–10.

[20] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, K. Matsumoto, Assessing the cost effectiveness of fault prediction in acceptance testing, IEEE Transactions on Software Engineering 39 (10) (2013) 1345–1357.

[21] A. Zaidman, B. Van Rompaey, A. van Deursen, S. Demeyer, Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining, Empirical Software Engineering 16 (3) (2011) 325–364.

[22] Z. Lubsen, A. Zaidman, M. Pinzger, Using association rules to study the co-evolution of production & test code, in: Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, 2009, pp. 151–154.

[23] C. Marsavina, D. Romano, A. Zaidman, Studying fine-grained co-evolution patterns of production and test code, in: 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Victoria, BC, Canada, 2014, pp. 195–204.

[24] G. Canfora, A. Cimitile, F. García, M. Piattini, C. A. Visaggio, Evaluating advantages of test driven development: a controlled experiment with professionals, in: 2006 International Symposium on Empirical Software Engineering (ISESE 2006) Rio de Janeiro, Brazil, 2006, pp. 364–371.

[25] C. W. J. Granger, Investigating causal relations by econometric models and cross-spectral methods, Econometrica 37 (3) (1969) 424–438.

[26] J. D. Hamilton, Time Series Analysis, Princeton University Press, 1994.

[27] H. Akaike, Information theory and an extension of the maximum likelihood principle, in: 2nd International Symposium on Information Theory, 1973, pp. 267–281.

[28] Y. Benjamini, Y. Hochberg, Controlling the false discovery rate: a practical and powerful approach to multiple testing, Journal of the Royal Statistical Society, Series B 57 (1) (1995) 125–133.

[29] A. Zeileis, T. Hothorn, Diagnostic checking in regression relationships, R News 2 (3) (2002) 7–10.

[30] B. Pfaff, Analysis of Integrated and Cointegrated Time Series with R, 2nd Edition, Springer, New York, 2008.

[31] B. Pfaff, Var, svar and svec models: Implementation within R package vars, Journal of Statistical Software 27 (4).

[32] F. Wilcoxon, Individual comparisons by ranking methods, Biometrics Bulletin 1 (6) (1945) 80–83.

[33] R. J. Grissom, J. J. Kim, Effect sizes for research: A broad practical approach, 2nd Edition, Lawrence Earlbaum Associates, 2005.

[34] S. L. Pfleeger, B. A. Kitchenham, Principles of survey research: part 1: turning lemons into lemonade, ACM SIGSOFT Software Engineering Notes 26 (6) (2001) 16–18.

[35] B. A. Kitchenham, S. L. Pfleeger, Principles of survey research: part 5: populations and samples, ACM SIGSOFT Software Engineering Notes 27 (5) (2002) 17–20.

[36] B. A. Kitchenham, S. L. Pfleeger, Principles of survey research part 4: questionnaire evaluation, ACM SIGSOFT Software Engineering Notes 27 (3) (2002) 20–23.

[37] B. A. Kitchenham, S. L. Pfleeger, Principles of survey research: part 3: constructing a survey instrument, ACM SIGSOFT Software Engineering Notes 27 (2) (2002) 20–24.

[38] B. A. Kitchenham, S. L. Pfleeger, Principles of survey research part 2: designing a survey, ACM SIGSOFT Software Engineering Notes 27 (1) (2002) 18–20.

[39] B. A. Kitchenham, S. L. Pfleeger, Principles of survey research part 6: data analysis, ACM SIGSOFT Software Engineering Notes 28 (2) (2003) 24–27.

[40] R. M. Groves, F. J. J. Fowler, M. P. Couyper, J. M. Lepkowski, E. Singer, R. Tourangeau, Survey Methodology, 2nd edition, Wiley, 2009.

[41] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, D. E. Damian, An in-depth study of the promises and perils of mining github, Empirical Software Engineering 21 (5) (2016) 2035–2071.

[42] G. Robles, J. M. González-Barahona, Developer identification methods for integrated data from various sources, in: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, 2005.

[43] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, A. Swaminathan, Mining email social networks, in: Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, 2006, pp. 137–143.

[44] P. Duvall, S. M. Matyas, A. Glover, Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series), Addison-Wesley Professional, 2007.

[45] G. Orellana, G. Laghari, A. Murgia, S. Demeyer, On the differences between unit and integration testing in the travistorrent dataset, in: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, 2017, pp. 451–454.

[46] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, S. Panichella, A tale of CI build failures: An open source and a financial organization perspective, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, 2017, pp. 183–193.

[47] RSpec. `http://rspec.info` (last access: 01/10/2019).