An Exploratory Study on the Usage of Gherkin Features in Open-Source Projects

Adwait Chandorkar*, Nitish Patkar†, Andrea Di Sorbo‡, and Oscar Nierstrasz†
*University of Wuppertal, Germany
Email: adwait.chandorkar@uni-wuppertal.de
†Software Composition Group, University of Bern, Switzerland
Email: http://scg.unibe.ch/staff
‡University of Sannio, Italy
Email: disorbo@unisannio.it

***Abstract*—With behavior-driven development (BDD), domain experts describe system behavior and desired outcomes through natural language-like sentences, *e.g.,* using the Gherkin language. BDD frameworks partially convert the content of Gherkin specifications into executable test code. Previous studies have reported several issues with the current BDD practice, for example long repetitive Gherkin specifications and slow-running test suites. Data tables and additional features were added to the Gherkin syntax to express compactly test inputs (*e.g.,* provide different combinations of input values and desired outputs to run tests multiple times) and also to improve the readability of Gherkin files (henceforth called *spec files*). However, there is no empirical evidence about the actual usage of these Gherkin features. To fill this gap, we analyzed the content of 1,572 spec files extracted from 23 open-source projects. For each spec file, we collected a set of metrics modeling the structure and the usage of the different Gherkin features. We found that only a minority of the considered spec files (*i.e.,* 590) used data tables that contain two rows, on average. We also used statistical tests to compare the contents of spec files with and without data tables and found significant differences between the two populations, especially for what concerns the number of lines of code (LoC). On the one hand, our results shed some light on the discrepancies between the recommendations for defining Gherkin specifications and their actual adoption in practice. On the other hand, our findings demonstrate that the adoption of additional features, such as data tables, might only partially help to reduce the length of Gherkin specifications.**

***Index Terms*—BDD, behavior-driven development, collaborative testing, behavior verification**

I. Introduction

BDD is an approach that enables domain experts to specify "live", executable, and testable requirements. Within BDD, domain experts specify application behavior through scenarios that everybody in a team can understand [1]. Domain experts and testers often leverage a constrained natural language, *i.e.,* Gherkin, to write behavior scenarios. For example, in Listing 1, we show a scenario that asserts the sum of two numbers for an arithmetic calculator application to have a particular value.

```
Feature: Basic arithmetic operations
As a user
I want to use a calculator to add numbers
So that I do not need to add them myself
Scenario: Add two numbers 2 and 3
Given I have a Calculator
When I add 2 and 3
Then the result should be 5
```

Listing 1: A sample feature description with a scenario

A typical Gherkin template splits a scenario into three core steps: *Given* (*i.e.,* a context assumed for this scenario execution), *When* (*i.e.,* an action or event that occurs in the given context), and *Then* (the expected outcome of the system for the provided action and context). A step can have additional context, expressed in the template by the keyword *And*. Apart from these four keywords, Gherkin contains several other keywords, such as *Background* or *Rule*. The BDD frameworks automatically tie the steps in scenarios to acceptance test cases (also called step definitions, glue code, or fixtures) to verify the specified functionality. Listing 2 shows the corresponding glue code for the scenario in Listing 1. The developers need to fill in the body of glue code methods.

```
public class CalculatorRunSteps {
private int total;
private Calculator calculator;
@Before
private void init() {
total = 999;
}
@Given("I have a calculator")
public void initializeCalculator() throws Throwable {
calculator = new Calculator();
}
@When("I add {int} and {int}")
public void testAdd(int num1, int num2) throws Throwable {
total = calculator.add(num1, num2);
}
@Then("the result should be {int}")
public void validateResult(int result) throws Throwable {
Assert.assertThat(total, Matchers.equalTo(result));
}
}
```

Listing 2: Glue code for the scenario from Listing 1

Next, developers implement the logic for the calculator application:

```
public class Calculator {
public int add(int a, int b) {
return a + b;
}
}
```

Listing 3: Implementation for the functionality from Listing 1

Finally, when domain experts execute the acceptance tests, the BDD frameworks present them with the test run status, *i.e.,* success or failure.

In a recent survey, software engineers and business analysts highlighted several shortcomings of current BDD practices [2]. They complained that they must write numerous scenarios with minor variations in input parameter values. Additionally, they must also specify the test assertions. They mentioned that when requirements change, a lot of manual effort is needed to maintain the textual scenarios and to manually propagate the changes to acceptance tests, leading them to perceive BDD as only an additional task to writing unit tests [2], [3]. To reduce the redundancy in spec files and improve their readability, keywords, such as *Scenario Outline*, and features, such as data tables—to support the strategy known as data-driven testing—were introduced in 2009 [4]. In Gherkin, a *Scenario outline* is parameterized using *Examples* data tables. In Listing 4, we see how a data table can be used to test several combinations of input numbers against the corresponding result.

```
Scenario Outline: Sample arithmetic additions
  Given I have a Calculator
  When I add "<num1>" and "<num2>"
  Then the result should be "<result>"

  Examples: Numbers
    | num1    | num2  | result    |
    | 1       | 3     | 4         |
    | 5       | 8     | 13        |
    | 7       | 2     | 9         |
```

Listing 4: A sample data table

Similarly, data tables can be passed into a step as an input data structure to improve the readability. Nevertheless, practitioners still have difficulty maintaining spec files a decade after these features were introduced. The goal of this exploratory study is to analyze the characteristics of the spec files and explore the usage of various features and especially data tables. Such analysis aims at exploring the actual contents of Gherkin specifications for better understanding the reason why practitioners perceive BDD as an overhead. Based on the aforementioned goal, we outline the following research questions:

- RQ_1: *What are the characteristics of the Gherkin specifications in open-source projects?*
- RQ_2: *What are the main differences between specs using tables and specs without tables?*

The original contributions of this paper are as follows:

- To the best of our knowledge, we are the first to analyze various characteristics of Gherkin specifications from open-source projects.
- We present comparison results between different types of Gherkin specifications, *i.e.,* with and without data tables.

Our results should foster discussion on an unexplored research area. Specifically, they highlight some issues with the current BDD practice that future research should consider improving.

The remainder of the paper is structured as follows: In section II, we present related work in the area of BDD. In section III, we describe the methodology we followed to conduct this analysis. In section IV and section V, we report and discuss our findings, respectively. section VI summarizes the threats to the validity. Finally, in section VII, we summarize our main contributions and we outline a plan to extend this study.

II. Related work

Several studies in recent years have proposed approaches and techniques to automate the BDD process. Soeken *et al.* proposed a technique to semi-automatically generate step definitions and code skeletons from scenarios given in natural language [5]. Patkar *et al.* analyzed the features of the current BDD tools and proposed to specify application behavior through in-IDE graphical interfaces [6]. With their approach, they engage non-technical stakeholders in the BDD process equally. Binamungu *et al.* presented a dynamic tracing based approach for detecting duplication in BDD suites [7].

Apart from these studies, several empirical studies shed light on various aspects of the BDD. Binamungu *et al.* surveyed 75 BDD practitioners to understand the extent of BDD use, its benefits and challenges, and specifically the challenges of maintaining BDD specifications in practice [2]. Their results showed that BDD specifications suffer from maintenance challenges due to the huge size of the BDD suites. They also conducted another survey with BDD practitioners, to hear their opinions on the quality criteria for the BDD specifications established by the authors themselves [8].

Yang *et al.* from 59,933 open-source Java projects retrieved 133 projects containing at least one *.feature* file [9]. They figure out whether and how accurately could they identify co-changes between *.feature* and source code files when either of those changes. In this study, they used natural language processing to check both *.feature* files and the source code files to detect the occurrences of common keywords. They did not study the step definitions, and their results are specific to Cucumber related projects.

Zampetti *et al.* analyzed 20 Ruby projects shortlisted from the top 50,000 projects— ranked in terms of several stars—hosted on GitHub for the five most popular programming languages, *i.e.,* Java, Javascript, PHP, Python, and Ruby [3]. Their goal was to study the extent to which open-source projects use BDD-related frameworks, *i.e.,* the percentage of projects that use one of the several BDD frameworks. They also surveyed 31 developers to understand how these developers use BDD frameworks in practice. They observed a co-evolution between scenarios and fixtures, and source code in about 37% of the projects. Specifically, the authors discovered that changes to scenarios and fixtures often happen together or after changes to source code. Moreover, survey respondents indicated that, while they understand the intended purpose of BDD frameworks, most of them write tests while/after coding rather than strictly applying BDD.

Neither of these studies shed light on the specifics of the spec files. The data published by Yang *et al.* contains repositories with fewer than ten stars and consists of mostly

small personal projects. Zampetti *et al.* offer to download their dataset. Regrettably, it consists of only processed data and not the data from the fetched repositories.

III. Study design

The goal of this study is to investigate the characteristics of the spec files in open-source projects. The study context consists of a total of 23 open-source repositories that use Gherkin to specify application behavior hosted on GitHub.

A. Data collection procedure

Figure 1 summarizes the data gathering process we followed. First, we used the GitHub search API to retrieve a list of repositories sorted by popularity. The limitation of this approach is that we must add at least one restriction for the search API to provide any results. A similar approach is applied in another study where the authors collected open-source projects hosted on GitHub ranked in terms of the number of stars [3]. We decided to retrieve only repositories with more than 500 stars, which resulted in a total of 54,277 repositories (as of 29 January 2021). A similar strategy has been used in several recent empirical analysis studies [10], [11], [12], [13]. This strategy allowed us to include projects with a certain level of popularity. Furthermore, it allowed us to reduce the scope of the projects to be processed and allowed us to respect the permitted quota of API requests for more in-depth qualitative analysis.

For all the identified repositories, we then fetched the list of programming languages. We selected only those that contain the Gherkin language. In this step, we retrieved a total of 318 repositories. If repository languages contain Gherkin, then it means they contain a spec file with the *.feature* extension.

Next, we grouped the identified repositories according to the primary programming language. For our analysis, we focused on Java as the primary language and selected the corresponding 37 repositories. From these 37 repositories, we identified spec files by searching for the *.feature* extension. We only selected those repositories that had more than one file with the *.feature* extension, as we speculate the projects with only one spec file probably only tried BDD and did not practice it rigorously. Consequently, we shortlisted a total of 27 repositories. By manually analyzing the search results, we found that some repositories contained files with the *.feature* extension that were not actual Gherkin files. To reduce the likelihood of considering false positives, for each repository we compared the list of files with the *.feature* extension and the list of files containing the keyword *Given*. If at least one file occurs in both lists, it is very likely that the result is a true Gherkin file. Eventually, we eliminated a total of four false positives from 27 repositories. We then manually cross-verified the obtained results. This process resulted in a total of 23 repositories for our qualitative analysis. From these 23 repositories, we collected a total of 1,572 spec files. We ran several scripts to analyze the contents of the identified spec files. All scripts together with the dataset to reproduce the results can be found in the replication package.[1]

B. Data analysis procedure

To answer the research questions, we collected data at two levels: the repository level and the spec-file level. We collected data against nine metrics listed below. All values are numeric unless mentioned in the brackets, and are correct as of June 23, 2021. Each metric is given an identifier starting with *m*.

1) Repository-level metrics: To provide the community with meta-level information about the analyzed repositories, we collected data for the following five metrics:

- *m1*: number of spec files,
- *m2*: date of the last commit on the repository (*a date*),
- *m3*: date of the last commit on the spec file (*a date*),
- *m4*: number of contributors to the repository,
- *m5*: number of contributors to the spec files.

2) Spec file-level metrics: To provide the community with information regarding the characteristics of Gherkin specifications, we collected data for the following twelve metrics:

- *m6*: LoC,
- *m7*: whether the spec file contains data table/s (*yes, no*),
- *m8*: number of data tables,
- *m9*: number of occurrences of the *Scenario* keyword,
- *m10*: number of of the *Given* keyword,
- *m11*: number of the *When* keyword,
- *m12*: number of the *Then* keyword,
- *m13*: number of the *And* keyword,
- *m14*: number of the *Scenario outline* keyword,
- *m15*: number of the *Examples* keyword,
- *m16*: number of the *Background* keyword,
- *m17*: number of the @ keyword.

Gherkin supports two types of syntactically similar data tables. They are added to the steps to improve readability, while they are added together with *Scenario outline* keyword to reduce redundancy. We used the resulting values for metrics m6 to m17 to answer RQ_1 and RQ_2.

C. Validity procedure

To answer RQ_1, we collected the metrics described in subsection III-B to have an overview of the content-related characteristics of spec files in open-source projects.

To answer RQ_2, we formulated the following null hypothesis:

> *For metrics* m6 *and* m17*, two distributions,* i.e., *spec files with and without tables, are similar.*

To test this hypothesis, we divided the selected 1,572 spec files into two groups: those with and without data tables. We compared the resulting values for metrics *m6* and *m9* for these two groups. Specifically, in order to assess if each considered metric presents different distributions for the two

[1] https://figshare.com/s/bc390cdcb12c11ce14b4

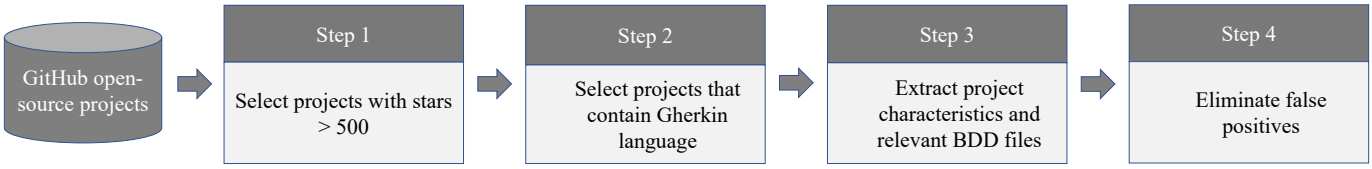

Fig. 1: The data collection procedure

distinct clusters with statistical evidence, we used the Mann-Whitney test (with the p value fixed to 0.05). This is a widely used nonparametric test to compare differences between two independent distributions [14]. In addition, to quantitatively assess the extent to which these groups are different, we used Cliff's delta, a measure of how often the values in one distribution are larger than the values in a second distribution. Following the guidelines of Grissom *et al.*, we interpreted the effect size as negligible for $|d| < 0.147$, small for $0.147 \leqslant |d| \leqslant 0.33$, medium for $0.33 \leqslant |d| \leqslant 0.474$, and large for $|d| \geq 0.474$ [15]. We used box plots to graphically visualize the distributions.

IV. Study Results

A. Descriptive statistics

In Table I, we group in descending order the top 10 programming languages of repositories that used BDD (from a total of 318 repositories). As we can see, projects with Ruby as the main programming language have used BDD the most, followed by PHP and Java.

TABLE I: Identified repositories according to programming languages

Language	# Repositories	Language	# Repositories
Ruby	82	Javascript	33
PHP	38	Go	16
Java	37	Emacs Lisp	11
Python	36	C#	10
C	17	TypeScript	5

Table II summarizes the meta information for the selected repositories. Columns *m1-5* present results for the corresponding metrics. It is evident from Table II that 11 out of 23 (about 48%) repositories have fewer than 20 spec files, whereas only three repositories have more than 100. Notably, one of those repositories is the Cucumber project itself, making the high number not so surprising. The number of contributors to the identified repositories is also quite diverse, ranging from 3 to 736. However, the number of contributors to the spec files is quite low, ranging from 6 to 15, meaning only a small fraction of all contributors are involved in modifying them. Most of the repositories (about 83%) have rather recent commits (in the year 2021), meaning these are actively maintained. However, only about 39% of the total repositories had last commits on spec files in 2021. Importantly, a similar number of repositories have the last commit on spec files before 2019, which makes us conclude:

Although repositories are maintained actively, teams might stop using BDD in their project. In these projects, BDD is perhaps not adopted as the main testing strategy.

B. What are the characteristics of the Gherkin specifications in open-source projects?

In Table III, we report values for various parameters derived for metrics *m6-8*. Each spec file, on average, contains about 3.47 scenarios. The average use of the *Background* (0.25 times per spec file) and *Feature* keyword (0.94 times per spec file) seem reasonable, meaning a negligible number of steps were common among a very few scenarios present in any spec file, and each spec file will typically test a single feature. A total of 590 spec files (*i.e.,* about 37.5%) contain tables. On average, each file contained 11 tables. Notably, each table has a mean of 2.6 rows and 1.7 columns, a median of 2.0 rows, and 1.0 columns.

Table IV further summarizes results for individual repositories. It is evident from Table IV that the average number of tables per specification is higher than expected because only four repositories, *i.e.,* R16, R21, R13, and R2, contain significantly more number of tables per spec file than other repositories, on average.

The average number of LoC in spec files varies greatly among selected repositories: from 8.51 to 217. The average number of scenarios per spec file also varies significantly across those containing tables. For example, if we compare repository R9 that has on average 21.2 scenarios per spec file without tables, in fact, has 32 scenarios per spec file containing tables. Finally, although the average number of scenarios per spec file is low, *i.e.,* 3.47 times (for the top three repositories, *i.e.,* R9, R19, R8, with a maximum number of scenarios per spec file, which is 21.2, 6.03, and 4.39 times), we expected more use of the @ keyword (*i.e.,* 0, 1.32, and 1 times), meaning that users did not particularly attempt to optimize the test run time by marking a subset of tests to be run independently.

Data tables are used in the minority of the analyzed specifications. Although we cannot determine the test objectives from our analysis, we can still report that data tables are not heavily used to improve readability and maintenance.

C. What are the main differences between specs using tables and specs without tables?

Table V summarizes the obtained results for RQ_2. Cliff's

TABLE II: The meta-level details of the selected repositories, *m1*: number of spec files, *m2*: date of the last commit on the repository (*a date*), *m3*: date of the last commit on the spec file (*a date*), *m4*: number of contributors to the repository, *m5*: number of contributors to the spec files.

Repo name	Index	*m1*	*m2*	*m3*	*m4*	*m5*
eugenp/tutorials	R1	15	08 Jun 2021	30 May 2021	736	06
neo4j/neo4j	R2	26	07 Jun 2021	04 Mar 2021	207	15
geoserver/geoserver	R3	04	08 Jun 2021	19 Sep 2017	266	01
apache/servicecomb-pack	R4	30	03 Apr 2021	16 Mar 2021	56	04
microservices-patterns/ftgo-application	R5	02	02 Jun 2021	10 Jul 2018	3	01
apache/tinkerpop	R6	59	07 Jun 2021	18 Jun 2021	142	06
iotaledger/iri	R7	05	18 Aug 2020	07 May 2020	58	04
SmartBear/soapui	R8	31	09 Dec 2020	07 Jul 2014	63	08
w3c/epubcheck	R9	36	15 Mar 2021	26 Feb 2021	11	03
aws/aws-sdk-java-v2	R10	53	07 Jun 2021	14 Aug 2018	70	01
bugsnag/bugsnag-android	R11	48	07 Jun 2021	15 Jun 2021	128	06
blox/blox	R12	10	12 Mar 2018	12 Feb 2018	22	03
ddd-by-examples/factory	R13	02	24 Apr 2021	22 Dec 2017	06	01
FluentLenium/FluentLenium	R14	11	08 Jun 2021	14 Jul 2019	62	02
AppiumTestDistribution/AppiumTestDistribution	R15	03	08 May 2021	19 Dec 2020	35	04
mzheravin/exchange-core	R16	02	25 Apr 2021	07 Jun 2020	05	02
iriusrisk/bdd-security	R17	11	08 Aug 2018	24 May 2018	10	01
jbangdev/jbang	R18	18	07 Jun 2021	24 May 2021	56	04
SoftInstigate/restheart	R19	31	07 Jun 2021	11 Jun 2021	27	04
intuit/karate	R20	394	24 May 2021	16 Mar 2021	54	09
cucumber/common	R21	439	07 Jun 2021	-	111	10
cucumber/cucumber-jvm	R22	92	06 Jun 2021	-	225	06
JetBrains/intellij-plugins	R23	103	07 Jun 2021	09 Apr 2021	208	06

TABLE III: General statistics across all spec files, *a*: a total number of spec files containing tables, *b*: an average number of LoC in all spec files, *c*: an average number of LoC in all spec files with tables, *d*: an average number of LoC in scenarios in all spec files, *e*: an average number of LoC in scenarios in all spec files with tables *f*: an average number of tables per spec file, *g*: an average number of scenarios per spec file, *h*: an average number of scenarios per spec file with tables.

a	b	c	d	e	f	g	h
590	38.53	73.41	35.53	71.11	11.41	3.47	5.85

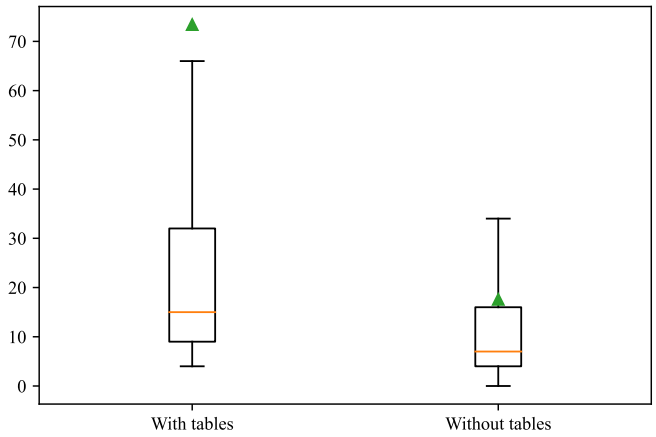

Fig. 2: Lines of code in Gherkin spec files

delta is large for the LoC and the count of the *Examples* keyword. In Figure 2, we show a box plot for the total number of LoC for spec files with and without tables. Although the mean value is around 73 (with tables), the median value is around 11-12. This result suggests that not all the repositories that used data tables write descriptive spec files. The mean is high because only a few repositories have a very high LoC. In Figure 3, we show a box plot for the count of the *Examples* keyword in spec files with and without tables. A large Cliff's delta for the *Examples* keyword seems reasonable as spec files without data tables most likely did not have the *Examples* keyword in them.

Cliff's delta is medium for the count of the following keywords: *Scenario outline* and *Given*. In Figure 4, we show a box plot for the count of the *Scenario outline* keyword in spec files with and without tables. A medium Cliff's delta for the *Scenario outline* keyword seems reasonable as spec files without data tables most likely did not have *Scenario outline* keyword in them. Looking at these results from a different perspective, spec files with tables did not extensively use the *Scenario outline* keyword, meaning in a small number of spec files, tables were used independently of the *Scenario outline* keyword. This indicates that they were used in the steps to improve the readability of the specification. In Figure 5, Figure 6, and Figure 7, we show a box plots for the count of the *Given*, *When*, and *Then* keywords in Gherkin files with and without tables.

It is evident that the median and mean values for *When* and *Then* keywords are approximately similar for spec files with and without tables. Ideally, the statistics should be approximately similar for the *Given*, *When*, and *Then* keywords in both spec files with and without tables. Although it is true for spec files without tables, we can see that the mean and median value for *Given* is higher than that for the other two keywords, meaning that *When* and *Then* were not always used with *Given*. Finally, Cliff's delta is negligible for the count

TABLE IV: Repository-wise details, *a*: an average number of LoC per spec file, *b*: an average number of LoC in scenarios per spec file, *c*: an average number of LoC per spec file with tables, *d*: an average number of LoC in scenarios per spec file with tables, *e*: an average number of tables per spec file, *f*: an average number of scenarios per spec file, *g*: an average count of the *Scenario outline* keyword per spec file, *h*: an average count of scenarios per spec file with tables, *i*: an average count of the *Scenario outline* keyword per spec file with tables, *j*: an average count of the *Given* keyword per spec file, *k*: an average count of the *When* keyword per spec file, *l*: an average count of the *Then* keyword per spec file, *m*: an average count of the *And* keyword per spec file, *n*: an average count of the *Feature* keyword per spec file, *o*: an average count of the @ (*i.e.,* tags) keyword per spec file, *p*: an average count of the *Background* keyword per spec file, *q*: an average count of the *Examples* keyword per spec file.

Index	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
R1	13.93	11.6	19.6	15.8	2	2.33	0.07	2.6	0.2	1.8	2.4	2.4	0.8	1	0.2	0.13	0.07
R2	217.15	215.11	217.15	215.11	13.04	12	0.22	12	0.22	7.67	12.89	12.3	22.59	1	0.33	0.33	0.22
R3	62	58.75	-	-	0	4.25	0	-	-	4.25	4.25	4.25	16.25	1	0.65	0	0
R4	25.97	24.97	25.97	24.97	3.63	1.17	0	1.17	0	1.63	1.17	2.57	4.47	1	0	0	0
R5	15	12	-	-	0	1.5	0	-	-	4.5	1.5	2.5	2	1	0	0	0
R6	157.39	156.39	160.5	159.5	10.81	11.95	0	12.43	0	11.95	11.64	11.93	19.47	1	0	0	0
R7	75.4	71.6	118.5	116.5	5.5	4.8	0	7.5	0	4.8	3.4	6.6	6.4	1	0	0	0
R8	31.87	29.45	29	26.67	2	4.39	0.45	1	1.67	3.1	2.97	3.16	12.03	0.94	1	0.03	0.45
R9	97.69	89.58	145.38	137.54	1.54	21.22	0	32	0	1.89	21.22	21.75	23.69	1	0	1	0
R10	10.51	8.42	10.51	8.42	1.17	2	0	2	0	0	2	2	0.23	1	1	0	0
R11	38.2	37.04	50.17	49	2.17	2.49	0	4	0	0	2.55	2.29	28.94	1	0.33	0	0
R12	23.4	19.1	29.8	24.8	4.4	2.9	0	3	0	2.4	3	3	3.2	1	2.2	0.3	0
R13	84.5	58.5	84.5	58.5	16.5	4.5	0	4.5	0	7	6	16.5	0.5	1	0	0	0
R14	8.82	7.82	-	-	0	1.91	0	-	-	1.55	1.91	1.91	0.36	1	0.18	0	0
R15	12.67	7.67	17	12	1	1.33	0.33	1	1	1	1.67	1.33	0.67	1	1	0	0.33
R16	89	82.5	89	82.5	24	1.5	0.5	1.5	0.5	3.5	10.5	10.5	18.5	1	2	0.5	0.5
R17	42.09	37.18	23.4	18.8	1.4	3.82	0.64	1.2	1	2.09	3.64	4.73	13.36	1	5.27	0.27	0.64
R18	14.83	13.22	-	-	0	3.17	0	-	-	0	2.94	3	0.28	1	0.06	0.28	0
R19	101.87	90.16	-	-	0	6.03	0	-	-	12.1	11.61	11.42	26.13	1	1.32	1	0
R20	19.02	16.04	45.47	41.34	2.07	1.92	0.19	2.87	1.18	0.81	0.88	1	1.74	1	0.44	0.44	0.22
R21	39.78	36.6	80.55	78.72	21.04	3.11	0.43	5.92	0.94	3.54	0.06	0.05	8.71	0.85	0.66	0.19	0.48
R22	11.48	9.06	21.38	18.88	2.31	1.49	0.27	2.16	0.78	1.62	1.16	1.28	0.3	0.95	0.35	0.13	0.39
R23	8.51	7.07	15.17	14.06	1.2	1.39	0.29	2.43	0.74	2.1	0.54	0.22	0.44	0.96	0.15	0.04	0.3

TABLE V: Mann-Whitney U test results

Metric	U-statistic	p-value	Cliff's delta
Total LoC	427753.0	5.727e-57	0.478
Count keyword "Scenario"	306273.0	0.0416	0.058
Count keyword "Scenario outline"	416816.0	1.357e-103	0.440
Count keyword "Given"	388831.0	4.864e-33	0.343
Count keyword "When"	320212.5	5.665e-05	0.106
Count keyword "Then"	330898.0	5.835e-08	0.143
Count keyword "And"	315600.0	0.000	0.090
Count keyword "@"	304969.0	0.019	0.054
Count keyword "Feature"	316044.5	7.034e-14	0.092
Count keyword "Background"	263536.0	5.817e-05	-0.089
Count keyword "Examples"	439738.0	2.569e-135	0.519

of the following keywords: *Scenario*, *When*, *Then*, *And*, @, *Feature*, and *Background*. From these statistical results we reject our null hypothesis and conclude that:

> *The characteristics of the Gherkin specifications with and without data tables are different for all the considered metrics.*

V. Discussion

Our results show that although Gherkin is syntactically equipped with keywords and features to reduce the length of the specifications, those features are not yet fully exploited by the users. The most prominent feature is data tables. The primary motivation of using data tables is to provide different combinations of input values and desired outputs to run scenarios multiple times. However, they can also be used to improve the readability and the maintainability of the Gherkin specifications. The average number of rows and columns in data tables indicates that tables were not used to reduce redundancy. We speculate that data tables could be hard to specify inputs and outputs when they grow in size. The shift to support Gherkin in Markdown format indicates the same [16]. Nevertheless, this shift is very recent, and with Markdown being the de-facto format to write Gherkin scenarios, we might see an increase in using various features of Gherkin.

Furthermore, using data tables to specify various combinations of input values and desired outputs make us wonder about the cognitive effort required when a scenario needs several inputs. If domain experts need to create a table with numerous columns and specify the corresponding outputs for each row, we can speculate that they need to put considerable effort into

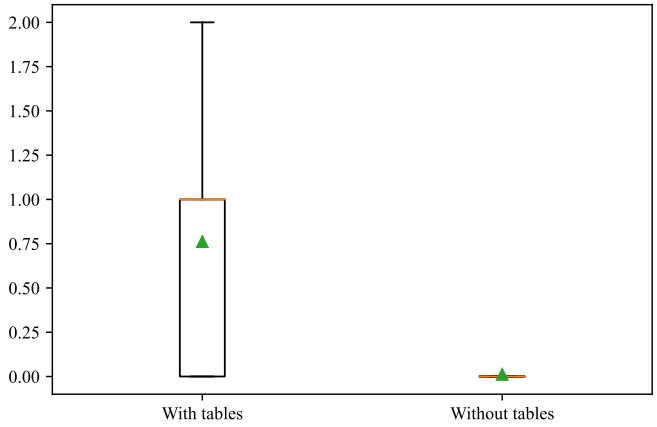

Fig. 3: Examples keyword count in Gherkin spec files

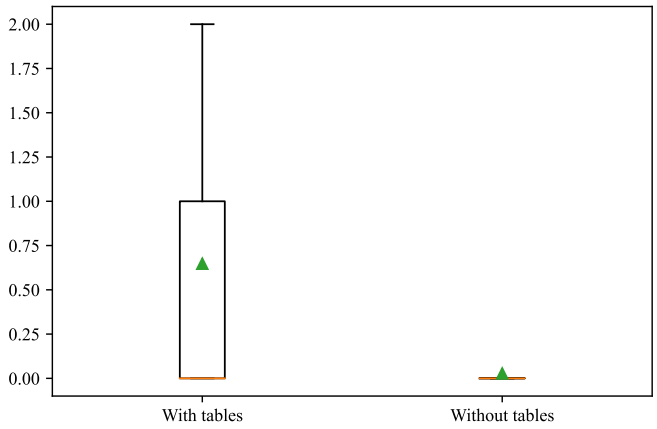

Fig. 4: Scenario outline keyword count in Gherkin spec files

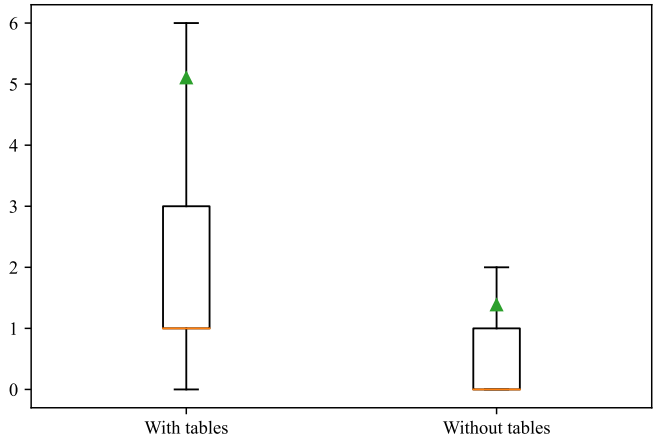

Fig. 5: Given keyword count in Gherkin spec files

writing long tables. There is also a high risk of specifying the wrong desired output when several input values are involved. To the best of our knowledge, no empirical study has looked into quantifying the cognitive effort yet.

Finally, there are two main implications of our results. First, it motivates us to explore the practitioners' view on the various

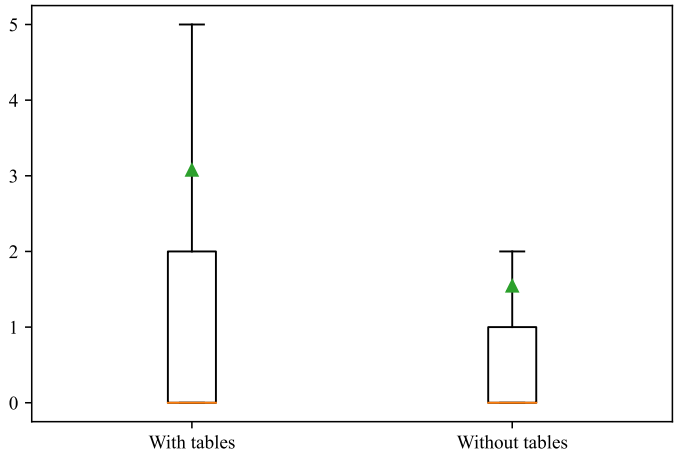

Fig. 6: When keyword count in Gherkin spec files

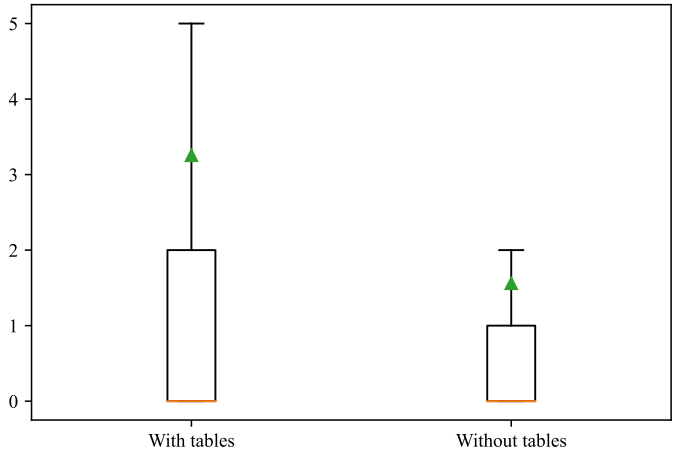

Fig. 7: Then keyword count in Gherkin spec files

feature of Gherkin. We intend to report their experience with features, such as data tables, to validate our speculations concerning the required effort. Second, our results make us curious to investigate the implications of Gherkin features on maintaining the glue code. We intend to undertake a large-scale commit analysis to study the co-evolution of Gherkin specifications and the corresponding glue code.

VI. Threats to the Validity

Several factors may have influenced the results of this study. These factors may have influenced the search, the repository selection, and the extraction of the data from the selected spec files. The four threats to the validity that apply in our case are discussed below.

Construct validity. Construct validity is used to determine how well the collected metrics measure what it is supposed to measure. Binamungu *et al.* proposed 14 aspects to determine the quality of BDD specifications. These 14 aspects could help determine the quality of BDD specifications in open-source projects, and may represent a threat to construct validity. However, these aspects require investigating the semantics

of the content and manually reading Gherkin specifications may help a little to understand the overall context of the project under consideration. With seventeen metrics presented in subsection III-B, we alleviate this threat as they help us collect data for all possible usage patterns concerning the syntax.

Internal validity. This type of validity concerns whether the study results follow from the data [17]. The accuracy of the results and conclusions suffers from the fact that we collected data for all metrics until 23 June 2021. However, given the time required to design the study, collect data, interpret the results, and report them, our results are still very recent and relevant.

Conclusion validity. This type of validity focuses on how sure we are that the treatment one used in an experiment is related to the actual outcome observed. We studied two distributions (spec files with data tables and without data tables, respectively). To alleviate this threat, we compared the values for distribution of each metric in the two groups, through the Mann-Whitney test and Cliff's delta effect size, reporting and discussing the differences which resulted statistically significant (p-value $\leqslant 0.05$) and exhibiting at least a small effect size ($|d| \geq 0.147$).

External validity. This type of validity concerns whether claims for the generality of the results are justified. We analyzed the contents of 1,572 spec files from 23 open-source projects that use Java as a primary programming language. We are aware that several other projects with the primary programming language other than Java exist. However, the choice of primary programming language does not compromise our results as our primary interest was the Gherkin specifications. The number of projects we selected might appear to be small. However, the selection criteria allowed us to include the projects with a certain level of popularity and enabled us to exclude the projects that are not maintained, which makes our findings reliable. Finally, we are aware that our results may not be generalizable to the private repositories. We can imagine that industrial projects have established standards for writing Gherkin specifications. To mitigate this threat, we intend to analyze proprietary projects in the future and compare our current results to paint a more realistic picture.

VII. Conclusion and future work

We analyzed 1,572 Gherkin specification files from 23 open-source projects that use BDD. The average number of LoC in spec files varies greatly among selected repositories, meaning BDD is likely used as main testing strategy only in the minority of cases. We found that data tables, which improve the readability and reduce the length of the Gherkin specifications, are not widely used yet. We applied the Mann-Whitney test and Cliff's delta to characterize two types of specifications: those with and without tables. We report that both the distributions are different for all the considered metrics. Notably, Cliff's delta is large for the LoC and the count of the *Examples* keyword. Our results shed light on the discrepancies between the recommendations for Gherkin specification and the actual usage in practice, fostering discussion on this unexplored area of research. We are currently conducting a practitioner survey to understand their viewpoint on the pros and cons of using Gherkin for behavior specification. In the future, we plan to investigate whether the usage of data tables could have negative effects on the maintenance of Gherkin specifications and glue code.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project no. 200020-181973, Feb. 1, 2019 - April 30, 2022). We thank Pooja Rani and Nataliia Stulova for their support during brainstorming.

References

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

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

[3] F. Zampetti, A. Di Sorbo, C. A. Visaggio, G. Canfora, and M. Di Penta, "Demystifying the adoption of behavior-driven development in open source projects," *Information and Software Technology*, p. 106311, 2020.

[4] "Gherkin reference," https://github.com/cucumber/common/blob/main/gherkin/CHANGELOG.md, accessed: 2021-04-03.

[5] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2012, pp. 269–287.

[6] N. Patkar, A. Chis, N. Stulova, and O. Nierstrasz, "Interactive behavior-driven development: a low-code perspective," 2021.

[7] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Detecting duplicate examples in behaviour driven development specifications," in *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*. IEEE, 2018, pp. 6–10.

[8] ——, "Characterising the quality of behaviour driven development specifications," in *International Conference on Agile Software Development*. Springer, Cham, 2020, pp. 87–102.

[9] A. Z. Yang, D. A. da Costa, and Y. Zou, "Predicting co-changes between functionality specifications and source code in behavior driven development," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 534–544.

[10] M. Altherwi and A. M. Gravell, "A large-scale dataset of popular open source projects." *J. Comput.*, vol. 14, no. 4, pp. 240–246, 2019.

[11] H. Ruan, B. Chen, X. Peng, and W. Zhao, "Deeplink: Recovering issue-commit links based on deep learning," *Journal of Systems and Software*, vol. 158, p. 110406, 2019.

[12] D. Favato, D. Ishitani, J. Oliveira, and E. Figueiredo, "Linus's law: More eyes fewer flaws in open source projects," in *Proceedings of the XVIII Brazilian Symposium on Software Quality*, 2019, pp. 69–78.

[13] A. Muna, "Assessing programming language impact on software development productivity based on mining oss repositories," *ACM SIGSOFT Software Engineering Notes*, vol. 44, no. 1, pp. 36–37, 2019.

[14] W. J. Conover, *Practical nonparametric statistics*. john wiley & sons, 1999, vol. 350.

[15] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[16] "Gherkin changelog," https://github.com/cucumber/common/blob/main/gherkin/CHANGELOG.md, accessed: 2021-04-03.

[17] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to advanced empirical software engineering*. Springer, 2008, pp. 285–311.