Hamcrest vs AssertJ: an Empirical Assessment of Tester Productivity

Maurizio Leotta, Maura Cerioli, Dario Olianas, Filippo Ricca

Abstract:

Context. Extensive unit testing is worth its costs in terms of the higher quality of the final product and reduced development expenses, though it may consume more than fifty percent of the overall project budget. Thus, even a tiny percentage of saving can significantly decrease the costs. Since recently competing assertion libraries emerged, we need empirical evidence to gauge them in terms of developer productivity, allowing SQA Managers and Testers to select the best.

Objective. The aim of this work is comparing two assertion frameworks having a different approach (matchers vs. fluent assertions) w.r.t. tester productivity.

Method. We conducted a controlled experiment involving 41 Bachelor students. AssertJ is compared with Hamcrest, in a test development scenario with the Java language. We analysed the number of correct assertions developed in a tight time frame and used this measure as a proxy for tester productivity.

Results. The results show that adopting AssertJ improves the overall tester's productivity significantly during the development of assertions.

Conclusions. Testers and SQA managers selecting assertion frameworks for their organizations should consider as first choice AssertJ, since our study shows that it increases the productivity of testers during development more than Hamcrest.

Digital Object Identifier (DOI):

https://doi.org/10.1007/978-3-030-29238-6_12

Hamcrest vs AssertJ: an Empirical Assessment of Tester Productivity

Maurizio Leotta[0000−0001−5267−0602], Maura Cerioli[0000−0002−8781−8782], Dario Olianas, Filippo Ricca[0000−0002−3928−5408]

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS)
Università di Genova, Italy
{name.surname}@unige.it

Abstract. *Context*. Extensive unit testing is worth its costs in terms of the higher quality of the final product and reduced development expenses, though it may consume more than fifty percent of the overall project budget. Thus, even a tiny percentage of saving can significantly decrease the costs. Since recently competing assertion libraries emerged, we need empirical evidence to gauge them in terms of developer productivity, allowing SQA Managers and Testers to select the best.
Objective. The aim of this work is comparing two assertion frameworks having a different approach (matchers vs. fluent assertions) w.r.t. tester productivity.
Method. We conducted a controlled experiment involving 41 Bachelor students. AssertJ is compared with Hamcrest, in a test development scenario with the Java language. We analysed the number of correct assertions developed in a tight time frame and used this measure as a proxy for tester productivity.
Results. The results show that adopting AssertJ improves the overall tester's productivity significantly during the development of assertions.
Conclusions. Testers and SQA managers selecting assertion frameworks for their organizations should consider as first choice AssertJ, since our study shows that it increases the productivity of testers during development more than Hamcrest.

Keywords: Hamcrest · AssertJ · Empirical Study.

1 Introduction

In the last two decades, automated software testing has gained the spotlights of software development. Indeed, starting from the *agile revolution*, tests have been an important part not only of quality assessment, but of code development itself, with practices like *test first*[1], *test driven development* (see e.g. [3]) and *behaviour driven development* (see e.g. [28]). Writing tests before developing the corresponding functionalities increases the requirement understanding, reduces the number and impact of defects in final products, and decreases the costs of bug fixing, letting them be spotted in the early phases of development (see e.g. [19]). It provides also *living documentation* (see e.g. [1, 20]), where tests clarify the expectations about the system and inherently document the current version of the system if they pass. Finally, comprehensive test suites, used as *regression tests*, let the system evolve with confidence that no undesired effects will take place.

[1] http://www.extremeprogramming.org/rules/testfirst.html

Though many different types of functional testing exist, the more widespread[2] among them is the basic unit testing, used to validate that each unit of the software (a method/class in the context of object-oriented programming) performs as designed. There is no doubt that extensive unit testing is worth its costs in terms of the higher quality of the final product and reduced development expenses. However, as stated in [11], "studies indicate that testing consumes more than fifty percent of the cost of software development". Thus, even small percentage savings on the testing process can significantly improve the project budget.

The first standard step toward saving on the testing process is to automate it through testing frameworks (e.g. JUnit or TestNG), which run the test method(s) and report successes/failures to the testers. The expected results are described by *assertions*, that are methods checking values (the result of the call under test, or the final status of some part of the system) against given conditions, and raising an exception in case of failure. Both testing frameworks and assertion libraries are currently a hot topic, with their numerous (mostly open-source) development projects showing a high number of commit and downloading, and their choice is far from obvious.

Nowadays, the most popular assertion libraries for the Java language are Hamcrest[3] and AssertJ[4]. The former hit the market first, is more well-established, and still attracts more attention, as for instance shown by Google Trend[5], where for the period from April 2018 to April 2019 on the average AssertJ scores forty-five and Hamcrest scores seventy-three over a maximum of a hundred. AssertJ, on the other hand, has a more active development community, provides more assertion methods out of the box, and adopts a *fluent style*, based on the dot notation, which is supposed to make writing assertions easier and the results more readable.

To investigate the claimed advantages of one of the frameworks w.r.t the other for what concerns the productivity of assertion writing, we applied Evidence-Based Software Engineering [13] to this context. In particular, we conducted a controlled experiment [27] involving forty-one Bachelor students to compare their capabilities in writing assertions in AssertJ and Hamcrest, taking into account both correctness and effort. We also devised a method to select the most practically essential assertion methods, so to restrict our experiment to writing tests involving those methods.

The paper is organized as follows: Section 2 briefly describes the assertion libraries used in this study: AssertJ and Hamcrest. The selection process for singling out the assertion methods more used in practice fills Section 3. The description of the empirical study and the preliminary results are in Section 4 and 5 respectively. Finally, related works are discussed in Section 6 while conclusions are given in Section 7.

2 Assertions: dot notation vs. matchers

JUnit[6], being the leading unit test framework for Java programming[7] is the natural choice as reference testing framework for our experiment. Indeed, its most recent version,

[2] https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

[3] http://hamcrest.org/

[4] https://joel-costigliola.github.io/assertj/

[5] https://trends.google.com/trends/explore?q=Assertj,Hamcrest

[6] https://junit.org

[7] https://redmonk.com/fryan/2018/03/26/a-look-at-unit-testing-frameworks/

JUnit 5, has built-in support for different assertion libraries, including those we want to compare: Hamcrest and AssertJ.

The *first generation* style of assertions used to be simply stating a boolean expression to be `true`. But that approach had a couple of major limitations: the expression to be evaluated is in most cases complex or a variable initialized by non-trivial code (in both cases making the test difficult to understand), and in case of failure, the error message is not much help, because it only says that the evaluation of the expression is `false`, while `true` was expected.

Thus the *second generation* of assertions has been introduced, with methods to state specific properties about the objects under test and the state of the overall system. For instance, `assertEquals` takes two arguments and states that they are equals. If they are not, the automatic error message captures the values of the parameters, giving more informative feedback. The problem with the second generation style is that numerous different methods would be needed to cover several common uses. But if all were provided, the users would have to memorize a plethora of assertion methods, to be able to select the needed one. Therefore, this approach cannot succeed, taken between lack of expressivity and steep learning curve.

More recently a *third generation* assertion style has appeared on the scene. It has just one (overloaded) assertion method, `assertThat`, and uses different mechanisms to provide the needed flexibility depending on the specific library. In the following, we will discuss the basics of Hamcrest and AssertJ approaches.

Hamcrest. The first release of Hamcrest dates back to 2006, and the name is an acronym of *matchers*, from its key concept. In Hamcrest, the `assertThat` method takes two parameters: the object of the assertion, in most cases the result of the call under test, and a matcher, a boolean function *matching* the first parameter against the expected property. For instance, if `c` is a collection of strings `assertThat(c, hasItems("cd","ef"))` succeeds if `c` contains both `"cd"` and `"ef"`.

Technically the matchers are just objects of classes implementing the `Matcher<T>` interface, but seldom they are directly created by a `new`. Good practices suggest providing, together with matcher classes, *factories* for them. Thus, we can use method calls to create new objects getting much more readable assertions. Indeed, the method names in factories, like the `hasItems` call in the previous example, are carefully chosen so that their invocations mimic natural language sentences.

Moreover, using methods to build objects paves the way to a language to compose simple matchers to get more complex ones. For instance, `everyItem` takes a matcher on the type of the elements and produces a matcher on a collection; thus, if `c` is a collection of numbers, `assertThat(c, everyItem(greaterThan(10)))` succeeds if all the elements in `c` are greater than 10. Other useful matcher composition methods are `allOf` and `anyOf`, to state that all/any matcher of a collection of matchers must succeed, and `not`, to negate matchers. Composing matchers greatly improves the expressive power of Hamcrest without dramatically increasing the number of matchers to memorize. However, the standard library defines a non-negligible number of basic matchers (about 90), and users are encouraged to define their own at need. Thus, users have to memorize many matchers to proficiently use Hamcrest and, though the matcher names are in most cases easy to remember/guess, still this need put a burden on the testers.

AssertJ follows a different approach w.r.t. Hamcrest. Indeed, its `assertThat` has a unique parameter, the element under test, and yields an object of a class providing methods to express conditions on values of the type of the parameter. For instance, if `f` is an `InputStream`, then `assertThat(f)` has type `AbstractInputStreamAssert`, with methods like `hasSameContentAs(InputStream expected)`. As for Hamcrest, choosing apt names for the assertion methods allows writing assertions reminiscent of English sentences, like for instance the following[8]: `assertThat(fellowshipOfTheRing).hasSize(9);` Moreover, as the result type of assertion methods is again a class of assertions, assertions can be naturally chained, like in

```
assertThat(fellowshipOfTheRing).hasSize(9)
                               .contains(frodo, sam)
                               .doesNotContain(sauron);
```

providing a logical conjunction, as the assertion passes if all its parts do.

More sophisticated logical manipulations can be expressed using the `is`/`are` and `has`/`have` assertion methods, that take a `Condition` as argument and match it against the element under test. All methods have the same semantics, but using the most appropriate from a linguistic point of view greatly improves readability (the plural forms are for collections and apply the condition to all their elements). Methods like `not`, `allOf`, `anyOf` can be used to create complex `Condition` expressions, as well as `areAtLeast`, `areAtMost`, `areExactly` on collections, taking a further integer argument and verifying that the number of the collection elements satisfying the condition matches the requirement. Hamcrest matchers can be used to create AssertJ conditions thanks to small adapters. Thus, AssertJ fully matches Hamcrest expressive power with a similar syntactic approach, besides providing other assertion methods.

But the real hit of AssertJ is that testers do not have to remember the names of the assertion methods, because writing `assertThat(_)` and following it with a dot they get help from the IDE (*code completion facility*), listing all the methods in the assertion class, that is, all the assertion methods applicable to the given type. Moreover, as the assertion methods are clustered by the type of the actual value to be tested, their number for each given type is manageable (though the overall collection is impressively large, with about 380 methods, and hence expressive), and static correctness reduces the risk of errors.

3 Practical usage of Hamcrest matchers and AssertJ methods

Both the Hamcrest and AssertJ frameworks are highly expressive, providing, respectively, many matchers and assertion methods. But some of them are more used/popular than others. Hence, to make a fair and significant comparison of the two frameworks, we need to determine which assertion methods/matchers are the most used in practice and focus the experiment mainly on them. With this aim, we applied the following mining procedure to repositories on GitHub, the world's leading repository hosting service:

1. we collected the list of all the Hamcrest matchers and AssertJ assertion methods;
2. we defined the repositories inclusion/exclusion criteria for our analysis;

[8] This example, as the following ones, is taken from https://joel-costigliola.github.io/assertj/

3. we devised an automated procedure to count instances of the matchers/assertion methods in our lists from the repositories satisfying our inclusion criteria;
4. finally, we ran the procedure and reported the results in percentage.

In the following subsections, a detailed explanation of the various steps is provided.

Collecting the lists of Hamcrest matchers and AssertJ methods. To find all the available matchers of Hamcrest (version 2.0.0.0) and the assertion methods of AssertJ (3.9.0), we used reflection on the respective packages. In particular, for Hamcrest, we extracted all the public static method names from the classes in the sub-packages of `org.hamcrest`. For AssertJ, we extracted all public method names from classes whose name ends in "Assert" in the package `org.assertj.core.api`.

Defining repositories selection criteria. Following the work of Kalliamvakou et al. [12] and Vendome et al. [26], to exclude too simple or personal repositories we selected only the ones matching the following criteria: ① at least six commits, ② at least two contributors, ③ at least one star[9] or watcher[10], and ④ not a fork. Indeed according to Kalliamvakou et al. the 90% of projects have less than 50 commits, the median number of commits is six (criterion ①), and the 71.6% of the projects on Github are personal projects (criterion ②). Moreover, criterion ③ guarantees that at least a user besides the owner is interested in the project. Finally, as in the work of Vendome et al., we excluded forks to avoid over-representation of matchers and methods used by highly forked projects (criterion ④).

Mining assertions data from GitHub. To find usages of AssertJ and Hamcrest assertions, we queried the Searchcode[11] API, that returns a list of files in public repositories containing the required text. The API allows filtering for language (Java in our case) and code repository hosting platform (we selected GitHub). Since results are limited to the first 1000 hits, to increase the number of results we searched matcher (method) by matcher (method). We constructed search queries concatenating the following elements:

- Import name for the considered framework (`org.hamcrest` or `org.assertj.core.api.Assertions.assertThat`)
- `@Test` (to be sure to include only occurrences in test scripts)
- Matcher (method) name

For each result (a file), after checking if the corresponding repository matches our criteria[12], the script downloads it and memorizes its Searchcode ID, to avoid downloading it again in case it contains other assertions and so appears in further queries. Then, our script analyses the file, identifies its assertions as the text between an `assertThat(` and a semicolon, cleans them from their parameters, and counts the occurrences of each matchers/methods appearing in them. To correctly identify matcher/method names and

[9] On GitHub users star other users' repositories to express appreciation. [10] Watchers are users who asked to be notified of repository changes. [11] https://searchcode.com/ [12] The information needed to apply the selection criteria is not completely accessible via the GitHub API. For example, it is not possible to directly ask for the number of commits, but only for a list of commits that may be divided into different pages, thus requiring several calls to the API. So, we retrieved the required information from the repository GitHub home page, using Requests-HTML for Python (https://html.python-requests.org/).

	Hamcrest Matcher	frequency	%	cumulative
✗	equalTo	22438	50,32%	50,32%
	is	9269	20,79%	71,10%
✓	notNullValue	2858	6,41%	77,51%
	nullValue	1656	3,71%	81,23%
✓	instanceOf	1470	3,30%	84,52%
✓	containsString	1258	2,82%	87,35%
✓	not	845	1,89%	89,24%
✓	sameInstance	612	1,37%	90,61%
✓	greaterThan	512	1,15%	91,76%
✓	closeTo	510	1,14%	92,90%
✓	hasItem	411	0,92%	93,83%
✓	hasSize	269	0,60%	94,43%
	anyOf	241	0,54%	94,97%
	hasItems	229	0,51%	95,48%
	contains	227	0,51%	95,99%
	greaterThanOrEqualTo	210	0,47%	96,46%
✓	allOf	205	0,46%	96,92%
	lessThanOrEqualTo	136	0,30%	97,23%
✓	startsWith	126	0,28%	97,51%
✓	lessThan	126	0,28%	97,79%
✓	hasEntry	123	0,28%	98,07%
	arrayWithSize	107	0,24%	98,31%
✓	empty	99	0,22%	98,53%
✓	hasKey	83	0,19%	98,72%
✓	endsWith	69	0,15%	98,87%
	containsInAnyOrder	53	0,12%	98,99%
	arrayContaining	48	0,11%	99,10%
	arrayContainingInAnyOrder	47	0,11%	99,20%
	hasToString	44	0,10%	99,30%
	hasItemInArray	40	0,09%	99,39%
	isIn	28	0,06%	99,46%
	isEmptyString	26	0,06%	99,51%
	typeCompatibleWith	25	0,06%	99,57%
	isOneOf	24	0,05%	99,62%
	either	24	0,05%	99,68%
	emptyArray	21	0,05%	99,72%
	hasXPath	21	0,05%	99,77%
	isEmptyOrNullString	17	0,04%	99,81%
	emptyIterable	16	0,04%	99,85%
	both	14	0,03%	99,88%
✓	everyItem	13	0,03%	99,91%
✓	hasValue	9	0,02%	99,93%
	hasProperty	8	0,02%	99,94%
	isA	7	0,02%	99,96%
	emptyCollectionOf	4	0,01%	99,97%

	AssertJ Method	frequency	%	cumulative
✗	isEqualTo	5555	43,41%	43,41%
✗	isFalse	833	6,51%	49,91%
✗	isTrue	818	6,39%	56,31%
	as	593	4,63%	60,94%
✓	isSameAs	507	3,96%	64,90%
✓	isNotNull	504	3,94%	68,84%
✓	isNotEqualTo	459	3,59%	72,43%
	isNull	438	3,42%	75,85%
	get	356	2,78%	78,63%
✓	hasSize	341	2,66%	81,29%
	containsExactly	247	1,93%	83,22%
	isLessThanOrEqualTo	217	1,70%	84,92%
✓	contains	154	1,20%	86,12%
	isAfterOrEqualTo	140	1,09%	87,22%
	isBeforeOrEqualTo	138	1,08%	88,30%
	isGreaterThanOrEqualTo	133	1,04%	89,33%
	isBefore	129	1,01%	90,34%
✓	isGreaterThan	128	1,00%	91,34%
	isAfter	124	0,97%	92,31%
✓	isEmpty	119	0,93%	93,24%
✓	isLessThan	117	0,91%	94,16%
✓	isInstanceOf	110	0,86%	95,01%
	containsOnly	48	0,38%	95,39%
✓	containsEntry	48	0,38%	95,76%
	isNotZero	44	0,34%	96,11%
	size	44	0,34%	96,45%
✓	isNotEmpty	43	0,34%	96,79%
	isZero	37	0,29%	97,08%
✓	exists	34	0,27%	97,34%
	overridingErrorMessage	31	0,24%	97,59%
✓	startsWith	25	0,20%	97,78%
	containsKey	23	0,18%	97,96%
	hasMessage	22	0,17%	98,13%
	isExactlyInstanceOf	20	0,16%	98,29%
	doesNotContain	20	0,16%	98,45%
	isNotSameAs	19	0,15%	98,59%
	asList	17	0,13%	98,73%
✓	endsWith	14	0,11%	98,84%
✓	containsValues	14	0,11%	98,95%
✓	isNotNegative	11	0,09%	99,03%
	isEqualToComparingFieldByf	9	0,07%	99,10%
	isNullOrEmpty	8	0,06%	99,16%
✓	containsKeys	8	0,06%	99,23%
	matches	7	0,05%	99,28%
	containsExactlyInAnyOrder	7	0,05%	99,34%

Table 1. Usage of Hamcrest matchers and AssertJ methods in real GitHub repositories.

avoid false positive due to substring collision, like for instance `isNot` in `isNotIn` for AssertJ, we search using a regular expression that matches strings:

- starting and ending with any string (the name may appear anywhere in the assertion)
- containing at least one non-alphanumerical character (the separator before the name)
- then the matcher/method name
- then an open round parenthesis (for the matcher/method call)

Results. Analysing the source code of the two frameworks, we found a total of 87 Hamcrest matchers and 376 AssertJ methods available to developers. Then, searching such matchers and methods in GitHub, we found 44592 hits in 2279 files for Hamcrest and 12798 hits in 770 files for AssertJ from 210 repositories overall (other 592 matching repositories did not satisfy the criteria, and were discarded).

Table 1 lists the 45 more frequent Hamcrest matchers and AssertJ methods. It is interesting to note that they achieve a cumulative frequency of 99.97% and 99.34% for Hamcrest and AssertJ, respectively. Moreover, about the 50% of the assertions use `equalTo` or `isEqualTo`, and only 51 out of 87 Hamcrest matchers and 75 out of 376 AssertJ methods are used at least once in the code found on GitHub repositories matching our criteria. These numbers suggest that developers mostly use a small fraction of the available constructs. Moreover, the most used assertion methods in AssertJ have semantic equivalent Hamcrest matchers in the top popular list and vice-versa.

Summary: Even if both frameworks provide many matchers and assertion methods, **in practice developers use only a few of them**. This result permits us to narrow the comparison only to a subset of assertions without significant loss of generality or fairness.

4 Experiment Definition, Design and Settings

Based on the Goal Question Metric (GQM) template [2], the main goal of our experiment can be defined as follows: "*Analyse the use of two different assertion frameworks for Unit Testing of Java programs with the purpose of understanding if there is an impact w.r.t. the production costs of test cases from the point of view of SQA Managers and Testers in the context of Junior Testers executing tasks of assertion development.*"

Thus, our research question is:

RQ. Does the tester productivity vary when using AssertJ instead of Hamcrest (or vice-versa)?

To quantitatively investigate the research question, we measured the productivity of the participants as the number of correct assertions developed in a limited amount of time (i.e., the number of correct assertions is a proxy for measuring the productivity construct).

The *perspective* is of *SQA Managers* and *Testers* interested in selecting the better framework for improving productivity. The *context* of the experiment consists of two collections of assertions (respectively *Obj1* and *Obj2*, i.e., the *objects*) both to be implemented in both frameworks and of *subjects*, 41 Computer Science bachelor students.

We conceived and designed the experiment following the guidelines by Wohlin *et al.* [27]. Table 2 summarizes the main elements of the experiment. For replication purposes, the experimental package has been made available: http://sepl.dibris.unige.it/HamcrestVsAssertJ.php.

In the following we present in detail: treatments, objects, subjects, design, hypotheses, variable and other aspects of the experiment.

Treatment. Our experiment has one independent variable (main factor) and two treatments: "H" (Hamcrest) or "A" (AssertJ). Thus, the tasks require adopting, in the former (latter) case, the Hamcrest (AssertJ) framework, that is, developing the assertions by Hamcrest matchers (AssertJ assertion methods).

Goal	Analyse the use of Hamcrest and AssertJ during test assertion development tasks to understand if there is a difference in terms of productivity
Quality focus	Correctness of the developed assertions
Context	Objects: two collections of Assertions (*Obj1*, *Obj2*) Subjects: 41 BSc students in Computer Science (3rd year)
Null hypothesis	No effect on productivity (measured as number of correct assertions developed in a limited time slot)
Treatments	Hamcrest and AssertJ frameworks in JUnit
Dependent variable	Total number of correctly developed assertions

Table 2. Overview of the Experiment

Objects. The objects of the study are two collections of seven *assertions descriptions* (*Obj1* and *Obj2*) included in a test suite for a simple JSON to CSV converter[13]. These assertions require working with lists, hash-maps, and other Java non-trivial types. The object design strives at balancing complexity as much as possible. For this reason, each assertion description in an object has a correspondent one in the other object with the same complexity, that is the same linguistic complexity and comparably straightforward to implement using our target assertion methods/matchers in both treatments. .

In details, starting from the list of the most used Hamcrest matchers and AssertJ methods (see Table 1) we conceived 14 assertions descriptions whose implementations are expected to use at least one of them. Indeed, our *reference implementation* of the 14 assertions descriptions in Hamcrest (AssertJ) uses 18 matchers (17 assertion methods) in the top-45 list (some assertion descriptions require more than one matcher (method) to be implemented), and a couple of less popular (`equalToIgnoringCase` and `anExistingFile` in Hamcrest; `isCloseTo` and `isEqualToIgnoringCase` in AssertJ). In Table 1 the green ✓ marks the specific matchers/methods expected to be used by the subjects participating in our experiment. Vice versa, red ✗ marks the matchers/methods forbidden unless expressly allowed in the development of the assertions. We added this constraint to the experiment to force the students to use the specific matchers (assertion methods) provided by the frameworks, preventing them from using complex expressions and `equalTo`/`isEqualTo` (always a viable, though in many cases low-quality, solution adopted by testers in about 50% of cases, as shown by our analysis).

Assertion Examples. Let us see an example of the tasks given to our subjects. The provided code includes the test setup and comments both to clarify the setup and to specify the assertion to be implemented by the students. For the sake of understandability, we have translated the comments in English, that in the experiment were in the student mother tongue (i.e., Italian).

```
@Test
public void testCsvContentAtIndex() {
 // Load a JSON file in the Hashmap f
 List<Map<String, String>> f =
   JSONFlattener.parseJson(new File("f/mysmall.json"), "UTF-8");

 // Assert that the forth element of flatJson:
 // (1) has a field named "user" with the value "John"
 // (2) has some field with 26462 as value
```

For the above example, for instance, we expected something like the following solutions.

```
 // Hamcrest reference implementation
 assertThat(f.get(4), allOf(hasEntry("user", "John"),
    hasValue("26462")));

 // AssertJ reference implementation
 assertThat(f.get(4)).containsEntry("user", "John")
    .containsValues("26462");
}
```

Subjects. The experiment was conducted in a research laboratory under controlled conditions (i.e., online). Subjects were 41 students from the Software Engineering course,

[13] https://github.com/Arkni/json-to-csv

	Group A	**Group B**	**Group C**	**Group D**
Lab 1	*Obj1* A	*Obj1* H	*Obj2* H	*Obj2* A
Lab 2	*Obj2* H	*Obj2* A	*Obj1* A	*Obj1* H

Table 3. Experimental Design (H = Hamcrest, A = AssertJ)

in their last year of the BSc degree in Computer Science at the University of Genova (Italy). They had a common Java programming knowledge, matured through a course of the previous year with significant project activity. Automated testing was explained during the Software Engineering course (i.e., the course in which the experiment was conducted), where detailed explanations on both Hamcrest and AssertJ were provided. Students participated into our experiment on a voluntary base, after five mandatory labs about software engineering, including one about unit test automation using basic JUnit assertions. Before the experiment, all the subjects have been trained on Hamcrest and AssertJ assertions with a one-hour presentation including assertions development (similar to the ones required in the experiment).

Experiment Design. The experiment adopts a counterbalanced design planned to fit two Lab sessions (see Table 3). Subjects were split into four groups balancing as much as possible their ability/experience, as ascertained by the previous mandatory software engineering labs. Each subject worked in Lab 1 on an object with a treatment and in Lab 2 on the other object with the other treatment.

Dependent Variables and Hypothesis Formulation. Our experiment has only one dependent variable, on which treatments are compared measuring the productivity construct for which we defined the relative metric (as done, e.g., in [21]). The number of correct assertions was used as a proxy to measure productivity. For each subject and lab, the *TotalCorrectness* variable was computed by summing up: *one* if the developed assertion is correct and *zero* if wrong, incomplete, or missing. Thus, the *TotalCorrectness* variable ranges from zero to seven, where seven corresponds to seven correct assertions. Since we could not find any previous empirical evidence that points out a clear advantage of one approach vs. the other, we formulated H_0 as non-directional hypothesis:

H_0. *The use of a framework w.r.t. the other does not improve the total correctness of the produced assertions*

The objective of a statistical analysis is to reject the null hypotheses above, so accepting the corresponding alternative one H_1 (stating instead that an effect exist).

Material, Procedure and Execution. To assess the experimental material and to get an estimate of the time needed to accomplish the tasks, a pilot experiment with two BSc students in Computer Science at University of Genova was performed. The students finished both tasks in 118 and 127 minutes (also producing some incorrect assertion implementations) and gave us some information on how improving the experimental material, in particular concerning the description of the assertions to develop. Given the times of the students and the time constraint of the labs, we set the total time of the entire experiment to 2 hours (1 hour for treatment).

The experiment took place in a laboratory room and was carried on using Eclipse. The subjects participated in two laboratory sessions (Lab 1 and Lab 2), with a short break between them.

For each group (see Table 3), each lab session required to develop seven assertions adopting Hamcrest or AssertJ respectively, and the subjects had 60 minutes to complete it. For each Lab session, we assigned a specific Eclipse project to each subject, and for each assertion to be developed, subjects: (a) recorded the starting time; (b) developed the assertion using the online documentation; (c) recorded the ending time.

Analysis Procedure. Because of the sample size and the non-normality of the data (measured with the Shapiro–Wilk test [23]), we adopted non-parametric tests to check the null hypothesis. This choice follows the suggestions given by [18, Chapter 37].

In particular, after computing descriptive statistics, we used a two-step analysis procedure. First, we calculated a contingency table displaying the frequency distribution of the variable *TotalCorrectness* for the two treatments. The table provides a basic picture of the interrelation between the treatment and the obtained correctness and provides a first rough insight into the results. To investigate the statistical significance, we applied the Fisher test.

Second, since subjects developed assertions of two different objects with the two possible treatments (i.e., Hamcrest and AssertJ), we used a paired Wilcoxon test to compare the effects of the two treatments on each subject. While the statistical tests allow checking the presence of significant differences, they do not provide any information about the magnitude of such a difference. Therefore, we used the non-parametric Cliff's delta (d) effect size [10]. The effect size is considered small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.

Since we performed two different analyses on the *TotalCorrectness* dependent variable, we cannot use $\alpha = 0.05$; we need to compensate for repeated statistical tests. While several techniques are available, we opted for the most conservative one, the Bonferroni correction [7]. In a nutshell, the conclusion will be taken by comparing the p-value to a corrected significance level $\alpha_B = \alpha/nt$, where nt is the number of statistical tests performed. Thus, in our case, $\alpha_B = 0.05/2 = 0.025$.

5 Results

Let us start with a short description of the results from the experiment, analysing the effect of the main factor on the dependent variable. Table 4 summarizes the essential descriptive statistics (i.e., median, mean, and standard deviation) of *Correctness*.

Table 5 presents the contingency table of *Correctness*, considering the tasks performed by the same subject as independent measures.

Comparing the results, it is evident that adopting AssertJ the participants were able to develop a greater number of correct assertions: 133 vs. 107. We applied the Fisher test on the contingency table that returned a p-value=0.017. From these results emerges a preliminary statistically significant influence of the treatment on the capability to develop correct assertions.

Subjects	Hamcrest			AssertJ			p-value	Cliff's Delta
	Mean	Median	SD	Mean	Median	SD		
41	2.600	2.000	1.172	3.325	3.000	1.289	0.0036	- 0.3131 (S)

Table 4. *Correctness*: descriptive statistics per Treatment and Results of paired Wilcoxon test

		Correct	
		Yes	**No**
Treatment	**Hamcrest**	107	180
	AssertJ	133	154

Table 5. *Correctness*: contingency table

Fig. 1 summarizes the distribution of *TotalCorrectness* by means of boxplots. Observations are grouped by treatment (Hamcrest or AssertJ). The y-axis represents the Total correctness achieved on the seven assertions to develop: score = 7 represents the maximum value of correctness and corresponds to developing seven correct assertions. The boxplots confirm the previous analysis: the participants achieved a better correctness level when developing the assertion using AssertJ (median 3) than Hamcrest (median 2). By applying a Wilcoxon test (paired analysis), we found that in this case too, the difference in terms of correctness is statistically significant, as p-value=0.0036, see Table 4. The effect size is small d=-0.3131. Therefore, we can reject the null hypothesis H_0 and accept H_1.

Summary: The adoption of AssertJ instead of Hamcrest significantly increase the number of correct assertions developed in a limited amount of time (60 minutes in our experiment).

Threats to validity. This section discusses the threats to validity that could affect our results: *internal*, *construct*, *conclusion* and *external* validity threats [27].
Internal validity threats concern factors that may affect a dependent variable (in our case, *TotalCorrectness*). Since the students had to participate in two labs (seven questions each), a learning/fatigue effect may intervene. However, the students were previously trained and the chosen experimental design, with a break between the two labs, should limit this effect.

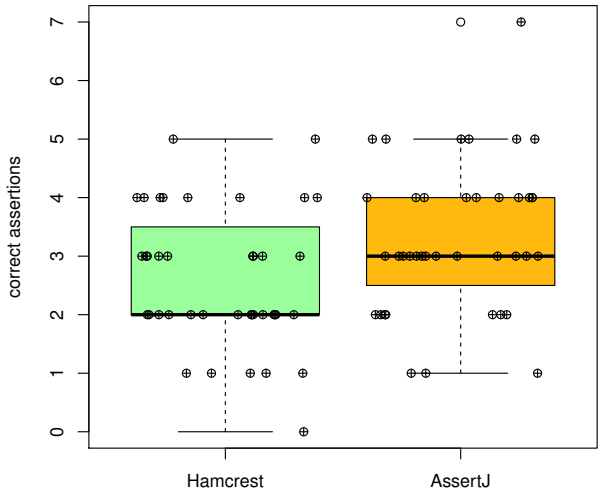

Fig. 1. Boxplots of Correctness (number of correct assertions developed in 60 minutes)

Construct validity threats are related to the evaluation of correctness, that is the only manual part of the data processing. Indeed, we used a script to extract from the student projects the assertions and their timing and did a spot-check of the script results. Then we run the student tests on the reference implementation and considered wrong the failing ones. Two of the authors independently manually evaluated the successful tests to identify those unable to spot bugs as required by their specifications, and a third author compared the evaluations. The controversial assertions were further analysed to reach a consensus. The overall scores have been automatically computed using a spreadsheet.
Threats to *conclusion validity* can be due to the sample size of the experiment (41 BSc students) that may limit the capability of statistical tests to reveal any effect, and the object size, that could be insufficient to significantly cover the assertion spectrum.
Threats to *external validity* can be related to the use of students as experimental subjects. We cannot expect students to perform as well as professionals, but we expect to be able to observe similar trends. Further controlled experiments with different sets of assertions and/or more experienced developers (e.g., software practitioners) are needed to confirm or contrast the obtained results.

6 Related Work

Since, as [6] states "*Where the creation, understanding, and assessment of software testing and regression testing techniques are concerned, controlled experimentation is an indispensable research methodology*", the empirical studies aiming at the reduction of testing costs are many. However, the approaches most studied to save effort during testing are (1) writing only those tests with maximum expectations of capturing a bug, (2) limiting regression testing to the tests with higher probability of finding errors (see e.g. [25]), and (3) improving automatization of testing (see e.g. [4, 15–17]). For the first two categories, researches focus on empirically assessing the probability of capturing bugs for given tests (see e.g. [24, 8, 25]), while papers in the last category compare the efficacy of automatic and manual testing techniques or of different automatic approaches (see e.g. [4, 22, 15]). Our study addresses an independent issue: (4) the influence of the choice of assertion style on the costs of developing tests. Thus, our work helps optimizing test development and adds to the optimization in category (3) and (1), as it applies to automated tests that have already been deemed necessary.

Another research topic somehow related to this paper is studying the understandability of tests. Most works in this area compare different categories of tests, like for instance Grano et al. [9] and Dak et al. [5], discussing readability of human-produced tests vs. automatically generated ones. A few papers, like for instance [14], and many web pages and posts[14] compare and discuss different styles of assertions. In particular, in [14] AssertJ, the same library used in our empirical study, is compared with JUnit Basic assertions, in a test comprehension scenario.

Understandability, though extremely important, is relevant mostly when *reading* tests as part of the code documentation or for their maintenance. Here, we focus instead on the costs of *developing* tests.

[14] https://www.blazemeter.com/blog/hamcrest-vs-assertj-assertion-frameworks-which-one-should-you-choose, https://www.ontestautomation.com/three-practices-for-creating-readable-test-code/, https://www.developer.com/java/article.php/3901236/

7 Conclusions and Future Work

In this paper, we have presented a controlled experiment comparing Hamcrest and AssertJ assertion styles from the point of view of development costs in practice. We analysed the number of correctly implemented assertions in a given time to gauge the overall productivity in completing the assignments. Even if the experiment has been conducted with two specific frameworks and with the Java language, we believe that our results can be generalized to other programming languages and other assertions frameworks belonging to the same categories of AssertJ and Hamcrest (i.e., able to provide fluent assertions and matchers). To get our comparison as fair and as useful in practice as possible, we first studied the usage distribution of Hamcrest matchers and AssertJ assertion methods, to be able to focus our experiment only on the most widely adopted (and not trivial).

The results indicate that adopting AssertJ significantly increases the number of correct tests so that AssertJ is a better choice over Hamcrest when development productivity is sought. This piece of empirical evidence can be exploited by Testers and SQA managers for a better selection of assertions frameworks for JUnit in their organizations. Even if we have no data supporting it, we believe that the aspect more relevant of AssertJ is its code completion facility that has simplified the activity of participants outperforming, in this way, Hamcrest. We intend to study this aspect in future works. Moreover, we plan to replicate this experiment with professional subjects to confirm our results; of course, we will use more challenging assertions as tasks, given the greater knowledge of the participants.

References

1. Adzic, G.: Specification by Example: How Successful Teams Deliver the Right Software. Manning Publications Co., 1st edn. (2011)
2. Basili, V.R., Caldiera, G., Rombach, H.D.: The goal question metric approach. In: Encyclopedia of Software Engineering. Wiley (1994)
3. Beck, K.: Test-driven development: by example. Addison-Wesley (2003)
4. Berner, S., Weber, R., Keller, R.: Observations and lessons learned from automated testing. In: Proc. of 27th Int. Conf. on Software engineering. pp. 571–579. ICSE 2005, ACM (2005)
5. Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W.: Modeling readability to improve unit tests. In: Proceedings of 10th Joint Meeting on Foundations of Software Engineering. pp. 107–118. ESEC/FSE 2015, ACM (2015). https://doi.org/10.1145/2786805.2786838
6. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering **10**(4), 405–435 (Oct 2005). https://doi.org/10.1007/s10664-005-3861-2
7. Dunn, J., Dunn, O.J.: Multiple comparisons among means. ASA pp. 52–64 (1961)
8. Garousi, V., Özkan, R., Betin-Can, A.: Multi-objective regression test selection in practice: An empirical study in the defense software industry. Inf. Softw. Technol. **103**, 40–54 (2018)
9. Grano, G., Scalabrino, S., Oliveto, R., Gall, H.: An empirical investigation on the readability of manual and generated test cases. In: Proceedings of 26th International Conference on Program Comprehension. ICPC 2018, ACM (2018)
10. Grissom, R.J., Kim, J.J.: Effect sizes for research: A broad practical approach. Lawrence Earlbaum Associates, 2nd edn. (2005)

11. Harrold, M.J.: Testing: A roadmap. In: Proceedings of 22nd International Conference on Software Engineering. pp. 61–72. ICSE 2000, ACM (2000)
12. Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D.: The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories. pp. 92–101. MSR 2014, ACM (2014)
13. Kitchenham, B.A., Dyba, T., Jorgensen, M.: Evidence-based software engineering. In: Proceedings of 26th Int. Conf. on Software Engineering. pp. 273–281. ICSE 2004, IEEE (2004)
14. Leotta, M., Cerioli, M., Olianas, D., Ricca, F.: Fluent vs basic assertions in java: An empirical study. In: Proceedings of 11th International Conference on the Quality of Information and Communications Technology. pp. 184–192. QUATIC 2018, IEEE (2018). https://doi.org/10.1109/QUATIC.2018.00036
15. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In: Proceedings of 20th Working Conference on Reverse Engineering. pp. 272–281. WCRE 2013, IEEE (2013). https://doi.org/10.1109/WCRE.2013.6671302
16. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Approaches and tools for automated end-to-end Web testing. Advances in Computers **101**, 193–237 (2016). https://doi.org/10.1016/bs.adcom.2015.11.007
17. Leotta, M., Stocco, A., Ricca, F., Tonella, P.: PESTO: Automated migration of DOM-based Web tests towards the visual approach. Journal of Software: Testing, Verification and Reliability **28**(4), e1665 (2018). https://doi.org/10.1002/stvr.1665
18. Motulsky, H.: Intuitive biostatistics: a non-mathematical guide to statistical thinking. Oxford University Press (2010)
19. Nagappan, N., Maximilien, E.M., Bhat, T., Williams, L.: Realizing quality improvement through test driven development: Results and experiences of four industrial teams. Empirical Software Engineering **13**(3), 289–302 (Jun 2008). https://doi.org/10.1007/s10664-008-9062-z
20. Ricca, F., Torchiano, M., Di Penta, M., Ceccato, M., Tonella, P.: Using acceptance tests as a support for clarifying requirements: A series of experiments. Information and Software Technology **51**(2), 270–283 (Feb 2009). https://doi.org/10.1016/j.infsof.2008.01.007
21. Ricca, F., Torchiano, M., Leotta, M., Tiso, A., Guerrini, G., Reggio, G.: On the impact of state-based model-driven development on maintainability: A family of experiments using UniMod. Empirical Software Engineering **23**(3), 1743–1790 (2018). https://doi.org/10.1007/s10664-017-9563-8
22. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (t). In: Proceedings of 30th International Conference on Automated Software Engineering. pp. 201–211. ASE 2015, IEEE (2015)
23. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). Biometrika **3**(52) (1965)
24. Soetens, Q.D., Demeyer, S., Zaidman, A., Pérez, J.: Change-based test selection: an empirical evaluation. Empirical Software Engineering **21**(5), 1990–2032 (2016)
25. Suri, B., Singhal, S.: Evolved regression test suite selection using BCO and GA and empirical comparison with ACO. CSI transactions on ICT **3**(2-4), 143–154 (2015)
26. Vendome, C., Bavota, G., Penta, M.D., Linares-Vásquez, M., German, D., Poshyvanyk, D.: License usage and changes: a large-scale study on github. Empirical Software Engineering **22**(3), 1537–1577 (2017). https://doi.org/10.1007/s10664-016-9438-4
27. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers (2000)
28. Wynne, M., Hellesøy, A.: The cucumber book: behaviour-driven development for testers and developers. Pragmatic Bookshelf (2012)