# Fluent vs Basic Assertions in Java: An Empirical Study

Maurizio Leotta, Maura Cerioli, Dario Olianas, Filippo Ricca

**Abstract:**

Context. Tests are becoming more and more central to the development process, so that their comprehensibility is of paramount importance. In particular, assertions, which express the test expected results, must be immediately understandable. Thus, recently several libraries emerged for making assertions "fluent", i.e., more comprehensible and easy to develop. However, there is no empirical evidence for the claimed advantages and benefits, that could convince SQA Managers and Testers for their adoption.

Objective. The aim of this work is gauging one of the claimed benefits of fluent assertions, namely improvement in comprehensibility, with respect to basic assertions.

Method. We conducted a controlled experiment involving 51 Bachelor students. AssertJ – a library supporting fluent assertions – is compared with JUnit Basic assertions, in a test comprehension scenario. We analysed the level of comprehension of the assertions, the time required to answer questions, and the overall efficiency in completing the assignments.

Results. The results show that adopting AssertJ has no significant effect on the level of comprehension of the assertions, though it significantly reduces the time required to understand assertions, so that it significantly improves the overall efficiency during comprehension of assertions.

Conclusions. From our study it emerges that fluent assertions are a better choice though they do not provide the expected improvements on understandability. Indeed they could be significantly improved by choosing methods names that better capture the assertion intended meaning.

**Digital Object Identifier (DOI):**

**https://doi.org/10.1109/QUATIC.2018.00036**

# Fluent vs Basic Assertions in Java: An Empirical Study

Maurizio Leotta, Maura Cerioli, Dario Olianas, Filippo Ricca
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
maurizio.leotta@unige.it, maura.cerioli@unige.it, S3717893@studenti.unige.it, filippo.ricca@unige.it

***Abstract*—*Context*. Tests are becoming more and more central to the development process, so that their comprehensibility is of paramount importance. In particular, assertions, which express the test expected results, must be immediately understandable. Thus, recently several libraries emerged for making assertions "fluent", i.e., more comprehensible and easy to develop. However, there is no empirical evidence for the claimed advantages and benefits, that could convince SQA Managers and Testers for their adoption.**

***Objective*. The aim of this work is gauging one of the claimed benefits of fluent assertions, namely improvement in comprehensibility, with respect to basic assertions.**

***Method*. We conducted a controlled experiment involving 51 Bachelor students. AssertJ – a library supporting fluent assertions – is compared with JUnit Basic assertions, in a test comprehension scenario. We analysed the level of comprehension of the assertions, the time required to answer questions, and the overall efficiency in completing the assignments.**

***Results*. The results show that adopting AssertJ has no significant effect on the level of comprehension of the assertions, though it significantly reduces the time required to understand assertions, so that it significantly improves the overall efficiency during comprehension of assertions.**

***Conclusions*. From our study it emerges that fluent assertions are a better choice though they do not provide the expected improvements on understandability. Indeed they could be significantly improved by choosing methods names that better capture the assertion intended meaning.**

***Index Terms*—Testing, Controlled experiment, JUnit, AssertJ**

## I. Introduction

Software testing is becoming more and more central to the development process. Nowadays, tests are of vital importance; not only they are used to ensure the quality of the software before deployment but they often drive the development and design (Test driven Development [4] and Behaviour driven development [20]), integrate or altogether replace software specifications [3], [14] and are a means of documentation [12]. In particular, agile software development methods, such as for example Extreme Programming, advocate the use of tests as a form of documentation [11]. The best way to learn what the code is supposed to do is to read and understand the tests.

Among the different types of testing, unit testing is used to validate that each unit of the software (a class or a method in the context of object-oriented programming) performs as designed. Usually, unit tests are automatically executed throw xUnit testing frameworks (e.g., JUnit for Java), which run the test method(s), and present the users with a report of successes/failures. The expected result is described by an *assertion*, which is a method verifying that some value (the result of the call under test, or the final status of some part of the system) complies with a given condition, and raising an exception in case of failure.

Since unit tests can be considered as a form of "living" documentation, which can be executed when one wants to understand the system functionalities, their readability and understandability is of paramount importance. Thus, in particular, assertions must be easy to grasp without any possible misunderstanding.

Each programming language and every testing framework has its own (often interchangeable) assertion libraries, which help testers to implement different styles of assertions in an easy way. Recently, several libraries emerged for making assertions "fluent", with the goal of improving test code readability and making production and maintenance of tests easier. One of the distinctive features of fluent assertions is the simple intuitive syntax able to structure the assertions as English sentences written in dot notation.

However, there is only anecdotal evidence about the superiority of fluent style over classic assertions [2], [6], [9]. Thus, without any scientific support, and no empirical evidence for the claimed advantages and benefits, it is difficult to convince SQA Managers and Testers to adopt them.

Since we are interested in investigating the claimed advantages and benefits of the fluent style, we decided to apply Evidence-Based Software Engineering [10] to this context. In particular, as a first step in this direction, we conducted a controlled experiment [19] involving 51 Bachelor students to compare fluent assertions against classic ones, the baseline, in a test comprehension scenario. Other styles of assertions will be considered in future experiments.

The paper is organised as follows: Section II describes Fluent and Basic assertions and the concrete implementations used in this study: AssertJ and JUnit Basic. The description of the empirical study and the preliminary results are in Section III and IV respectively. Finally, related works are discussed in Section V while conclusions are given in Section VI.

## II. Fluent and Standard JUnit Assertions

As a reference testing framework for our experiment, we selected JUnit[1], the leading unit test framework for Java programming [16]. Its most recent version, JUnit 5, supports

[1] https://junit.org

a few libraries for assertions, and has an open extensible architecture ready to include others. Tests written using previous versions (3 and 4) of JUnit can still be run as they are through the *Vintage test engine*[2]; but an improved version of the standard library, called Jupiter, is provided as well. As instance for *Basic* assertions, in our experiment we use a small subset of assertions, compliant with both Jupiter and older versions of JUnit. The entry point is one class, providing many assertion methods. The simplest ones have just one parameter (besides an optional string for error messages), the object to be checked; for instance `assertNull(o)`, stating that the object `o` is expected to be null, or `assertTrue(b)`, stating that some boolean expression `b` should hold. Other common assertion methods have two parameters (again, besides an optional error message) and compare them; for instance if `a1` and `a2` are arrays, then `assertArrayEquals(a1,a2)` states that the expected value `a1` is the same as the actual value `a2` computed by the call under test. Since the provided methods can only cover a small amount of cases, quite often the testers need to write a few lines of code to express complex assertions, or use non-trivial expressions as arguments of some given assertion method.

For instance, to assert that at least one element of the Array `toSearch` appears in the Array `target` (see question *Obj1* Q3* of our experiment) we have the following code, with nested loops to compare elements pairwise, and find a match, if any.

```
boolean match = false;
for(String r : toSearch) {
  for(String e : target) {
    if(e.equals(r)) {
      match=true;
      break;
    }
  }
  if(match) break;
}
assertTrue(match);
```

In this case the assertion itself has a trivial form, but some non-trivial code is needed to correctly initialize its argument. In other cases, vice versa, there is no need for extra lines of code, but the assertion statement is more complex. For instance in question *Obj2* Q8* of our experiment, we use

```
assertTrue((Math.abs(n-10) <= 3) &&
           (Math.abs(n-6) < 3));
```

to state that both $|n-10| \leq 3$ and $|n-6| < 3$. Sometimes both techniques are needed in combination.

The main problem of JUnit Basic assertions is that the available assertion methods are just a handful. Indeed, the limited choice of assertion methods makes the usage of `assertTrue` way too widespread, with complex expressions as argument, or using variables as arguments and non-trivial code to initialize them. This approach leads to assertions difficult to write/read, and moreover in case of failure the error message is not helpful, because it only says that the evaluation of the expression is **`false`**, while **`true`** was expected. On the other hand, introducing a plethora of assertion methods, to cover as many common uses as possible, is not feasible, as the users should memorize all of them, to be able to select the needed one.

Struggling with these problems, a few groups have proposed alternative assertion libraries in the last ten-fifteen years. One of them, Hamcrest[3], has been included in standard JUnit 4 since 2007 (see [1]), while JUnit 5 encourages its users to adopt whatever library they prefer to write *complex* tests, suggesting Hamcrest, AssertJ[4] and Truth[5], as sponsored options. The latter two libraries are quite similar, being both forks of a common ancestor, Fest[6]. As instance of fluent assertions, in our experiment we use AssertJ, but, given the similarity of the two libraries for the simple examples we used, most probably we would get the same results using Truth.

*Fluent* assertions, like those provided by AssertJ and Truth, are based on a different approach w.r.t. basic built-in ones: they are based on the existence of many overloading of an `assertThat` method, any of which takes a unique parameter, the actual value to be tested. The result of `assertThat(T actual)` is an object of a class providing methods to express conditions on values of type `T`. For instance, if `d` is a date, then `assertThat(d)` has type `AbstractDateAssert`, with methods like `hasSameTimeAs(Date date)`, or `isAfter(Date other)`.

Thus, on the one hand asserting a condition is very easy: the tester just has to write `assertThat(_).` and, at the hitting of the dot, the IDE suggests all possible conditions for the given type. Thus, there is no burden of remembering the correct method to use. 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 and hence expressive), and static correctness reduces the risk of errors.

On the other hand, choosing apt names for the assertion methods allows writing assertions which resemble English sentences, hence the *fluent* name. Consider for instance the following examples, taken respectively from questions *Obj1+ Q3* and *Obj1+ Q9* of our experiment:

```
assertThat(target).containsAnyOf(toSearch};
```

and

```
assertThat(date1)
  .isEqualToIgnoringHours(date2);
```

Finally, since the result type of assertion methods is again a class of assertions in AssertJ[7], assertions can be naturally chained. For instance in question *Obj2+ Q16* of our experiment we have three assertions on the same map:

[2]https://junit.org/junit5/docs/current/user-guide/#dependency-metadata-junit-vintage

[3]http://hamcrest.org/,

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

[5]https://google.github.io/truth/,

[6]https://mvnrepository.com/artifact/org.easytesting/fest-assert,

[7]In Truth this feature is not available, while both library support a further chaining capability, that is, selectors for the type on which the assertion works lift to the assertion class, so that it is possible to write something like `assertThat(list).last().isEqualTo(x)`

```
assertThat(gameParticipants)
  .containsOnlyKeys(Color.BLUE,
        Color.BLACK, Color.RED)
  .doesNotContainKey(Color.WHITE)
  .doesNotContainEntry(Color.RED, "micheal");
```

## III. Experiment Definition, Design and Settings

We conceived and designed the experiment following the guidelines by Wohlin *et al.* [19]. Table I summarizes the main elements of the experiment. For replication purposes, the experimental package has been made available[8].

| | |
|---|---|
| **Goal** | Evaluate the effect of adopting AssertJ fluent assertions during test code comprehension tasks |
| **Quality focus** | (i) Comprehension of the assertions<br>(ii) Time<br>(iii) Efficiency |
| **Context** | Objects: two collections of Assertions (*Obj1*, *Obj2*)<br>Subjects: 51 BSc students |
| **Null hypotheses** | (i) No effect on comprehension<br>(ii) No effect on time<br>(iii) No effect on efficiency |
| **Treatments** | AssertJ (Fluent) and Built-in (Basic) Assertions in JUnit |
| **Dependent variables** | (i) TotalComprehension of the provided assertions<br>(ii) TotalTime required to answer the questions<br>(iii) TotalEfficiency in completing the assignments |

TABLE I. Overview of the Experiment

The *goal* of the study is analysing the effects of adopting AssertJ "fluent" assertions rather than conventional JUnit "basic" ones during tasks requiring test code comprehension, like, for instance, test execution (e.g., understanding why a test case fails), as well as, maintenance or debugging (e.g., understanding how to update and correct a test case).

The *perspective* is of *SQA Managers* and *Testers* interested in evaluating different ways to express assertions in terms of (1) comprehension of the assertions and (2) time required for understanding them. The *context* of the experiment consists of two collections of assertions (respectively *Obj1* and *Obj2*, i.e., the *objects*) and of *subjects*, 51 Computer Science BSc students. In what follows, we present in detail: treatments, objects, subjects, design, hypotheses, variables and other aspects of the experiment.

### A. Treatment

Our experiment has one independent variable (main factor) and two treatments: "*****" (JUnit Basic) or "**+**" (AssertJ). In the first case, the comprehensions tasks are performed on JUnit Basic assertions, while in the second case on AssertJ ones.

[8]http://sepl.dibris.unige.it/FluentVsClassic.php

### B. Objects

The objects of the study are two collections of assertions: *Obj1* and *Obj2*. The two objects have been created by defining an overall list of 20 assertions in which those in odd positions have the same level of complexity as those in the next even position. Then the 10 odd assertions (1,3,5,...) have been assigned to *Obj1* while the 10 even ones (2,4,6,...) to *Obj2*. Thus, the two objects are comparable in complexity and size, because they consist of equivalent collections of assertions on arrays, maps, lists, and dates. The AssertJ assertions have been created by analysing the ones shown in the AssertJ library documentation. Thus, they are all simple, but not *too* simple; indeed, for trivial assertions the basic library is already good enough, and it would difficult to compare the results. Then, we developed equivalent assertions using the JUnit Basic library. Both treatments have been carefully inspected and executed, including them in executable test scripts, in order to guarantee their correctness.

### C. Subjects

The experiment was conducted in a research laboratory under controlled conditions (i.e., online). Subjects were 51 students from the Software Engineering course, in their last year of the BSc degree in Computer Science at the University of Genova (Italy). They had an average programming knowledge (specifically, of Java and C# programming). Participants had, in previous years of their career, mandatory exams for the courses: Programming I and II, Algorithms and Data Structures, Object Oriented Programming, Data Bases and Advanced Object Oriented Programming. Automated testing was explained during the Software Engineering course (i.e., the course in which the experiment was conducted), where detailed explanations on AssertJ and JUnit Basic assertions were provided. The experiment was introduced as a supplementary laboratory assignment. Before it, the students participated in five labs about software engineering including one about unit test automation using both AssertJ and JUnit Basic assertions.
Before the experiment, all the subjects have been (re-)trained on AssertJ and JUnit Basic assertions with a one hour presentation including sample questions and answers like the ones used in the experiment.

### D. Experiment Design

The experiment adopts a counterbalanced design planned to fit two Lab sessions (see Table II). Subjects were split into four groups balancing as much as possible their ability/experience, as ascertained by a previous software engineering lab on development of automated unit tests using the JUnit framework (including both JUnit Basic and AssertJ assertions).

Each subject worked in Lab 1 on an object with a treatment and in Lab 2 on the other object with the other treatment. We choose a design ensuring that each subject works on different objects in the two Labs, receiving each time a different treatment; this is the best choice when a limited number of participants is available. Indeed, it is well-known that a

| | Group A | Group B | Group C | Group D |
|---|---|---|---|---|
| **Lab 1** | *Obj1* + | *Obj1* * | *Obj2* * | *Obj2* + |
| **Lab 2** | *Obj2* * | *Obj2* + | *Obj1* + | *Obj1* * |

TABLE II. Experimental Design ( * = JUnit Basic, + = AssertJ)

counterbalanced design limits as much as possible learning effects [13].

### *E. Dependent Variables and Hypothesis Formulation*

Our experiment has three dependent variables, on which treatments are compared measuring three different constructs: (i) *Comprehension* of the assertions, (ii) *Time* required to answer questions, and (iii) *Efficiency* in completing the assignments. Each construct is measured with a variable (respectively *TotalComprehension*, *TotalTime*, and *TotalEfficiency*) for which we defined the relative metric (as done in [15]).

The comprehension of each assertion was assessed by computing Precision, Recall and the corresponding F-Measure. *TotalComprehension* variable was computed by summing up the 10 values for each subject. Thus, the *TotalComprehension* variable ranges from zero to ten, where ten corresponds to a perfect precision and recall on all 10 questions.

Time was measured by means of time sheets. Students recorded starting time for each question and stopping time of the overall questionnaire. In this way, we were able to compute the time required to answer the questions (the *TotalTime* variable), as difference between the ending time of the questionnaire and the starting time of the first question.

The efficiency is a derived measure that is computed as the ratio between overall assertions comprehension and time to answer the ten questions (more is better).

$$TotalEfficiency = \frac{TotalComprehension}{TotalTime} \quad (1)$$

Thus, we can state the null hypotheses for the study in this way:

– $H_{0a}$: *TotalComprehension* (AssertJ) = *TotalComprehension* (JUnit Basic)

– $H_{0b}$: *TotalTime* (AssertJ) = *TotalTime* (JUnit Basic)

– $H_{0c}$: *TotalEfficiency* (AssertJ) = *TotalEfficiency* (JUnit Basic)

Since we could not find any previous empirical evidence that points out a clear advantage of one approach vs. the other, we formulated $H_{0a}$, $H_{0b}$, and $H_{0c}$ as non-directional hypotheses. The objective of a statistical analysis is to reject the null hypotheses above, so accepting the corresponding alternative ones: $H_{1a}$, $H_{1b}$, and $H_{1c}$.

### *F. 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 one BSc student in Computer Science at University of Genova was accomplished. The student finished the assignment in 26 and 32 minutes for AssertJ and JUnit Basic respectively and gave us some information on how improving the experimental material.

The experiment took place in a laboratory room and was carried on using paper-based questionnaires. First, students completed the training session. Then, they participated into two laboratory sessions (Lab 1 and Lab 2), with a short break between them. Finally, students were asked to compile a post experiment questionnaire.

For each group (see Table II), the questionnaire contained 10 questions, each consisting of an assertion expressed in AssertJ or JUnit Basic and five possible answers (out of which only one or two correct), and the subjects had 40 minutes to complete it.

For each Lab session, the experiment execution steps were as follows:

1) We delivered the sheet containing the questionnaire to each subject.
2) Subjects filled their personal data in the delivered sheet.
3) For each question:
   a) Subjects recorded the starting time.
   b) Subjects answered the question (possibly accessing the online documentation).
3) Subjects recorded the questionnaire ending time.

Finally, subjects were asked to complete a post-experiment questionnaire aimed at both gaining insights about the students' behaviour during the experiment, and finding motivations for the quantitative results. It included questions about: availability of sufficient time to complete the questions, documentation clarity, exercise usefulness, perceived comprehensibility of the provided assertions, willingness to adopt AssertJ assertions in some future real-world projects, competencies required to answer. The post-experiment questionnaire is outlined in Table IV. Answers are on a Likert scale ranging from one (strongly agree) to five (strongly disagree).

### *G. Analysis Procedure*

Because of the sample size and mostly non-normality of the data (measured with the Shapiro–Wilk test [17]), we adopted non-parametric test to check the three null hypotheses. This choice is in accordance with the suggestions given in [13, Chapter 37]. Since subjects answered to the questions of two different objects with the two possible treatments (i.e., AssertJ and JUnit Basic), 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 [8]. The effect size is considered small for $0.148 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.

In all the performed statistical tests, we decided, as it is customary, to accept a probability of 5% of committing Type-I-error [19], i.e., rejecting the null hypothesis when it is actually true.

| Dependent Variable | Subjects | AssertJ | | | JUnit Basic | | | p-value | Cliff's Delta |
|---|---|---|---|---|---|---|---|---|---|
| | | Median | Mean | SD | Median | Mean | SD | | |
| TotalComprehension | 51 | 9.00 | 8.85 | 0.92 | 8.50 | 8.43 | 1.21 | 0.086 | + 0.197 (S) |
| TotalTime | 51 | 18.00 | 18.63 | 6.04 | 24.00 | 24.61 | 7.55 | <0.001 | – 0.468 (M) |
| TotalEfficiency | 51 | 0.52 | 0.53 | 0.18 | 0.35 | 0.39 | 0.17 | <0.001 | + 0.489 (L) |

TABLE III. Descriptive Statistics per Treatment and Results of paired Wilcoxon test

## IV. Preliminary Results

This section starts with a short description of the results from the experiment, analysing the effect of the main factor on the dependent variables. Then, it discusses some results from the post-experiment questionnaires.

Table III summarizes the essential descriptive statistics (i.e., median, mean, and standard deviation) of *Comprehension*, *Time*, and *Efficiency*, and the results of paired statistical analyses conducted on data from the experiment with respect to these dependent variables.

### A. $H_{0a}$: Comprehension

Fig. 1 summarizes the distribution of *TotalComprehension* by means of boxplots. Observations are grouped by treatment (AssertJ or JUnit Basic). The y-axis represents the average comprehension measured with the F-measure on the 10 questions: score = 10 represents the maximum value of comprehension and corresponds to provide correct answers to all the 10 questions. The boxplots show that the subjects achieved a slightly better correctness level when accomplishing the tasks with AssertJ (median 9.00) with respect to JUnit Basic (median 8.50).

By applying a Wilcoxon test (paired analysis), we found that the difference in terms of comprehension is not statistically significant, as p-value=0.086. The effect size is small d=0.197. Therefore, we cannot reject the null hypothesis $H_{0a}$.

**Summary**: The adoption of AssertJ has no significant effect on the level of comprehension of the assertions.

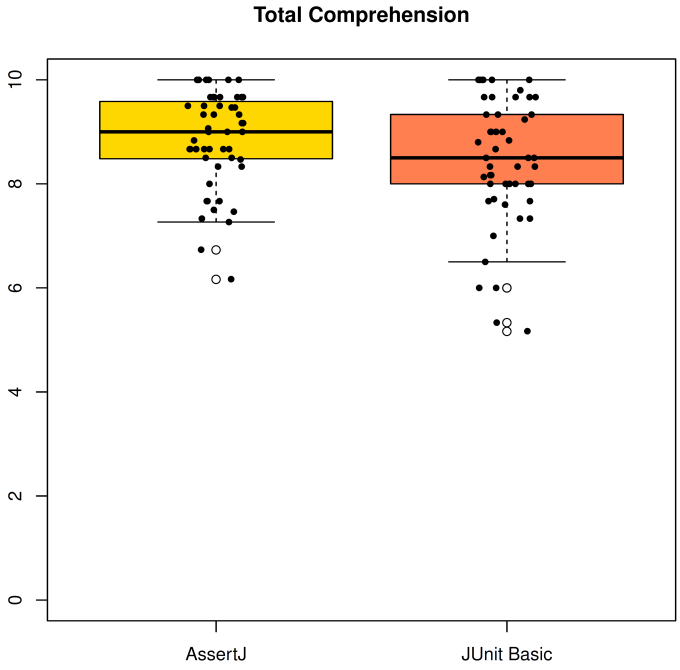


Fig. 1. Boxplots of Comprehension

### B. $H_{0b}$: Time

The second hypothesis can be easily tested by looking at the time needed to answer all the ten questions. Fig. 2 summarizes the distribution of the *TotalTime* variable by means of boxplots, and the y-axis represents the total time to answer the 10 questions. The boxplots show that using JUnit Basic assertions students needed more time than using AssertJ to answer the 10 questions (24 vs 18 minutes respectively in the median case).

To test the second hypothesis, we used a Wilcoxon test (paired analysis) as done for the first hypothesis. The results show that the overall difference is statistically significant, as p-value<0.001. The effect size is medium d=–0.468.

Overall, we can reject the second null hypothesis $H_{0b}$.

**Summary**: The adoption of AssertJ significantly reduces the time required to understand assertions.

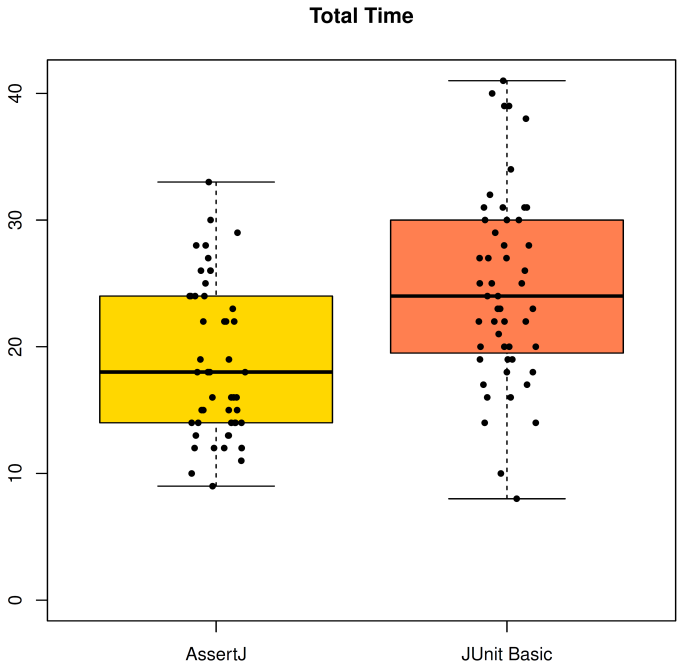


Fig. 2. Boxplots of Time

### C. $H_{0c}$: Efficiency

Figure 3 summarizes the distribution of the TotalEfficiency variable by means of boxplots. As for the previous cases, observations are grouped by treatment (AssertJ and JUnit Basic).

The boxplots in Fig. 3 show that students working with AssertJ assertions outperformed in terms of efficiency students using JUnit Basic ones (median 0.52 vs 0.35 respectively). The difference is statistically significant since the Wilcoxon test provides p-value<0.001. The effect size is large d=0.489. Therefore, we can reject the null hypothesis $H_{0c}$.

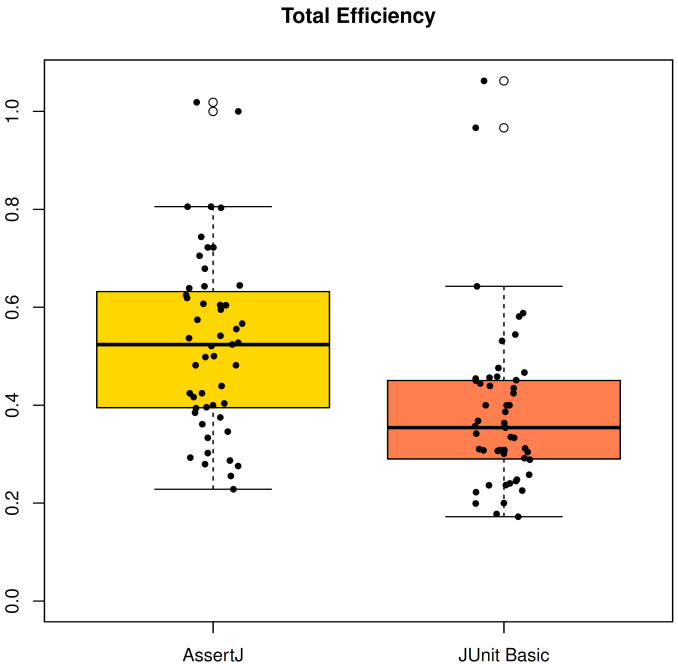


Fig. 3. Boxplots of Efficiency

**Summary**: The adoption of AssertJ significantly increases the overall efficiency during comprehension of assertions.

### D. Post-Experiment Questionnaire

The post-experiment questionnaire is summarized in Table IV, together with the medians of the answers given by the students. The possible choices for each answer, on a 5-point Likert scale, were: Strongly Agree, Agree, Unsure, Disagree, Strongly Disagree.

Students perceived the time allowed to answer the questions to be sufficient both in the case of AssertJ (strong agree) and JUnit Basic (agree). They found the AssertJ assertions immediate to comprehend, but they also perceived the JUnit Basic ones as quite simple to understand. Concerning the documentation, students found it useful and clear (PQ5), and they found the exercise useful, in the context of their current courses (PQ6). Students in both lab sessions found questions concerning AssertJ assertions simpler to answer than those on JUnit Basic (PQ7). Students would use the AssertJ assertions in an industrial project (PQ8) and perceived AssertJ assertions to be suitable for people with their level of knowledge (PQ9).

| ID | Question | Median |
|---|---|---|
| PQ1 | I had enough time to answer AssertJ questions | Strongly Agree |
| PQ2 | I had enough time to answer JUnit Basic questions | Agree |
| PQ3 | I had no problems to understand the provided AssertJ assertions | Strongly Agree |
| PQ4 | I had no problems to understand the provided JUnit Basic assertions | Agree |
| PQ5 | The provided documentation was useful and clear | Agree |
| PQ6 | I found the exercise useful | Agree |
| PQ7 | Answer AssertJ questions is simpler than answer JUnit Basic ones | Agree |
| PQ8 | I would adopt AssertJ in an industrial project | Agree |
| PQ9 | I had enough knowledge to answer the questions | Agree |

TABLE IV. Post-experiment Questionnaire and Medians of the Answers

For a more complete picture, Figure 4 graphically shows the distribution of the answers to the post-experiment questionnaire by means of a bar chart.

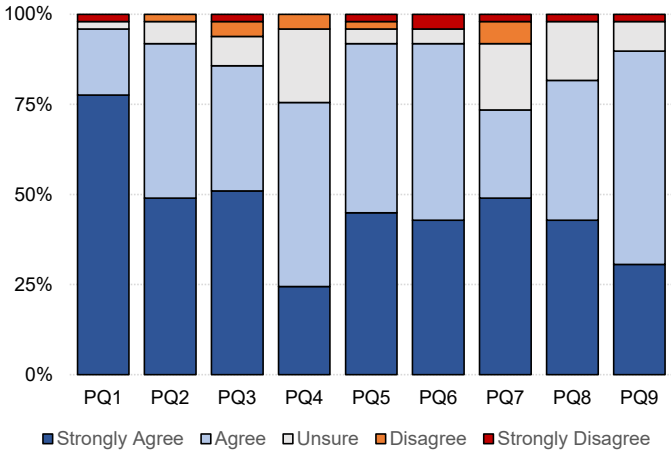


Fig. 4. Distributions of the Answers to the Post-experiment Questionnaire

### E. Discussion

Overall, hypothesis $H_{0b}$ and $H_{0c}$ concerning respectively the time and efficiency can be rejected. On the contrary, hypothesis $H_{0a}$ concerning comprehension cannot be rejected even if, considering the medians, the direction is in favour of AssertJ w.r.t. JUnit Basic (9.00 vs 8.50 respectively, see Table III). Therefore, we can infer that AssertJ reduces the time required for a tester to understand the provided assertions without affecting the comprehension level. We postulate that this improvement derives from the higher level of abstraction

| *Obj1* | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Q1 | Q3 | Q5 | Q7 | Q9 | Q11 | Q13 | Q15 | Q17 | Q19 |
| + AssertJ | 0.77 | 0.91 | 0.97 | 1 | 0.77 | 0.97 | 0.92 | 0.9 | 1 | 0.99 |
| * JUnit Basic | 0.65 | 0.88 | 0.9 | 1 | 0.98 | 1 | 0.62 | 1 | 1 | 1 |
| gap | 0.11 | 0.04 | 0.07 | 0 | **-0.21** | -0.03 | **0.30** | -0.10 | 0 | -0.01 |

| *Obj2* | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Q2 | Q4 | Q6 | Q8 | Q10 | Q12 | Q14 | Q16 | Q18 | Q20 |
| + AssertJ | 0.75 | 1 | 0.85 | 0.8 | 0.71 | 0.63 | 0.93 | 0.91 | 0.97 | 0.94 |
| * JUnit Basic | 0.69 | 1 | 0.78 | 0.94 | 0.95 | 0.41 | 0.81 | 0.78 | 0.95 | 0.59 |
| gap | 0.06 | 0 | 0.07 | -0.14 | **-0.24** | **0.22** | 0.12 | 0.13 | 0.02 | **0.34** |

TABLE V. F-measures of AssertJ and JUnit Basic for the individual questions.

of AssertJ, whose assertions provide in a more compact and less scattered way the same information given in the JUnit Basic case by extra code, or more complex arguments. These quantitative results are in accordance with the qualitative impression of the students (see PQ1-4 and PQ7 in Table IV), and with the opinion that AssertJ assertions are easier to read, largely diffused in the practitioners' community[9].

However, the practitioners' and students' expectation that AssertJ assertions are consistently more understandable is not supported by our results on hypothesis $H_{0a}$. To investigate the reasons for this misperception, in Table V we compare the F-measures of AssertJ and JUnit Basic for the individual questions. The *gap* is the difference between AssertJ and JUnit Basic F-measures; we emphasize gaps in bold when their absolute value is greater than 0.2.

Since the mean of TotalComprehension for JUnit Basic is 8.43, with a standard deviation of 1.21 (see Table III), its results are quite good and there is a small margin of improvement. Thus, for AssertJ to succeed over JUnit Basic, it should have performed better on many questions. Instead, out of 20 questions (see Table V), AssertJ has better results only in 11 (gap positive), barely more than half, while JUnit Basic wins on 6 (gap negative). Thus, let us briefly discuss the questions where JUnit Basic significantly out-performs AssertJ, that is, questions Q9 in *Obj1*, and Q10 in *Obj2*.

Let's compare the assertions in the case of question Q9:

```
assertThat(date1)                    // Obj1+ Q9
  .isEqualToIgnoringHours(date2);

assertTrue (                         // Obj1* Q9
  date1.getDay() == date2.getDay()
  && date1.getMonth() == date2.getMonth()
  && date1.getYear() == date2.getYear());
```

Analysing the wrong answers, the problem seems to be in the choice of name for the assertion method. Indeed, `isEqualToIgnoringHours` has been interpreted by some responders as "the dates may have different hours, but must have the same minutes and seconds" while the real meaning is that actual and given dates have same year, month and day fields (while hour, minute, second and nanosecond fields are ignored in comparison).

Both versions of assertion in question Q10, *Obj2+ Q10* and *Obj2* Q10*, are short and easy to read:

```
assertThat(date1)                    // Obj2+ Q10
 .isInSameHourWindowAs(date2)
 .isInSameHourAs(date3);

assertTrue(                          // Obj2* Q10
  Math.abs(date1.getTime()-date2.getTime())
    < 3600000  // milliseconds in one hour
  && date3.getHours() == date1.getHours());
```

However, almost a fourth of the students filling *Obj2+* answered *Obj2+ Q10* as if the second method call had `date2` as receiver, instead than `date1`. This could point out a cognitive problem with nested assertions: misperceiving the last object encountered as the subject of the next call. This hypothesis is worth investigating by a different experiment, but cannot be ascertained on the basis of just one question in our experiment.

The choice of names affects comprehension not only for question *Obj1+ Q9* (F-measure = 0.77), but also for question *Obj2+ Q2* (F-measure = 0.75):

```
assertThat(colors)
 .containsOnlyElementsOf(elems);
```

where the meaning of `containsOnlyElementsOf`, defined in the documentation as "*verifies that actual* (e.g., colors) *contains all the elements of the given iterable* (e.g., elems) *and nothing else, in any order and ignoring duplicates*", has been mistaken for any of

- actual contains *some of* the elements of the given iterable and nothing else
- actual contains the same elements of the given iterable, in any order and *with the same multiplicity*.

[9]See e.g. http://www.vogella.com/tutorials/AssertJ/article.html, https://dzone.com/articles/writing-tests-like-a-novelist, https://allegro.tech/2014/10/java-testing-toolbox.html, https://professional-practical-programmer.com/2016/06/26/assert-j/

See also the issue[10] opened on GitHub.

The choice of names affects comprehension in both ways; for instance, the AssertJ version of the question Q13 of *Obj1*:

```
assertThat(arr1).isSubsetOf(arr2);
```

grossly outperformed its counterpart in JUnit Basic style, both in term of comprehension (0.92 vs. 0.62 respectively) and elapsed time (1.54 vs. 3.54 minutes respectively), as almost all participants correctly understood the meaning of `isSubsetOf`. Analogous result we have for question *Obj2+ Q20*, where almost all the responders correctly interpreted the method `containsOnlyOnce` between lists, while its JUnit Basic counterpart confused many.

A borderline case is question Q12 of *Obj2*:

```
                                    // Obj2+ Q12
assertThat(a).containsOnlyOnce(b);

for(int t : b) {                    // Obj2* Q12
  int found = 0;
  for(int i : a) {
    if(i == t) found++;
  }
  assertEquals(found,1);
}
```

Answering *Obj2+ Q12*, 28% of the responders interpreted the `containsOnlyOnce` method as meaning *exactly one value of the argument* (b) *belongs to the receiver* (a) instead of *the actual group* (a) *contains the given values* (b) *only once*, as stated by the method specification; thus, we have a further example of unfortunate name choice. But performances on *Obj2* Q12* were even worse: 33% of the responders made the same mistake (probably misreading the final assertion as to be placed after the end of the external loop); moreover, another 26% scattered their (all wrong) answers without a significant pattern, suggesting the subjects' incapability of understanding this code, though it is quite simple. Thus, the poor performances in both treatments, due to different causes, partially mask each other.

**Summary**: *testers* should carefully analyse the documentation of the chosen assertion library in order to fully understand the real meaning of the available methods. At the same time, *assertion libraries developers* should pose more attention in the choice of the methods names. This could reduce misunderstanding and make the learning curve gentler.

### F. Threats to validity

This section discusses the threats to validity that could affect our results: *internal*, *construct*, *conclusion* and *external* validity threats [19].

*Internal validity* threats concern factors that may affect a dependent variable (in our case, *TotalComprehension*, *TotalTime*, and *TotalEfficiency*). Since the students had to participate in two labs (ten 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. Another threat derives from the subjectivity involved in the construction of the JUnit Basic versions of the assertions that we implemented by starting from the AssertJ ones. We applied a standard approach to implement JUnit Basic assertions but we cannot be sure that changing the implementation would not affect the outcome of the experiment. We plan to verify this as part of our future work.

*Construct validity* threats concern the relationship between theory and observation. This threat is related to how comprehension and time were measured. The transcription of the selected answers for each question has been double-checked by two of the authors. Precision, Recall and F-measure have been automatically computed using Excel. Time was measured by means of time sheets, and was validated qualitatively by two of the authors, who were present during the experiment.

Threats to *conclusion validity* can be due to the sample size of the experiment (51 BSc students) that may limit the capability of statistical tests to reveal any effect.

Threats to *external validity* can be related to: (i) the choice of simple assertions as objects and (ii) the use of students as experimental subjects. We cannot expect students to perform as well as testers, but we expect to be able to observe similar trends. Further controlled experiments with different sets of assertions (e.g., more complex) and more experienced developers (e.g., software practitioners) are needed to confirm or contrast the obtained results.

## V. Related Work

There are several empirical studies about understandability of JUnit tests. For instance, Grano et al. [7] use an empirical approach to show that in many cases the code under test is more readable than its tests, and that human produced tests are more readable than automatically generated ones. Dak et al. [5] compare the readability of manually vs. automatically generated tests, and use empirical experiments to prove effectiveness of the proposed technique to improve readability of automatically generated tests. Analogously, Tahir et al. [18] compare understandability of parametric and standard concrete test cases by an experiment measuring time and correctness ratio, and find that, as expected, standard concrete tests are easier to read and understand.

Though these papers addresses readability of test code by means quite similar to those used in this paper, their focus is on comparing different categories of tests disregarding the assertion styles in them, and we were not able to find any academic work specifically on assertion understandability, nor on comparison of different assertion styles.

On the other hand, there are many web pages and posts stating the superiority of fluent (or Hamcrest) style over classic JUnit Basic assertions, for instance [2], [6], [9], but without any

[10] https://github.com/joel-costigliola/assertj-core/issues/1075

scientific support, and missing a clear statement of the aspects improved or a specific quantification of the improvements.

## VI. Conclusions and Future Work

In this paper, we have presented a controlled experiment aimed at comparing basic and fluent assertions (in particular JUnit Basic and AssertJ). We analysed the level of comprehension of the assertions, the time required to answer questions, and the overall efficiency in completing the assignments.

The results indicate that adopting AssertJ: (1) has no significant effect on the level of comprehension of the assertions; (2) significantly reduces the time required to understand assertions; (3) significantly increases the overall efficiency during comprehension of assertions. Therefore, fluent assertions are a better choice over basic ones.

From the results it also emerges that: (1) testers should carefully analyse the documentation of the chosen assertion library in order to fully understand the real meaning of the available methods; (2) developers of assertion libraries should pose more attention in the choice of the methods names. This could reduce misunderstanding and make the learning curve gentler.

We see the experiment presented here as the first step of an exploration of assertion understandability encompassing different styles and languages. Thus, we plan to expand it first by a three ways comparison of JUnit Basic, AssertJ and Hamcrest assertions. The general practitioners' vision is that the AssertJ style is more readable than the Hamcrest one, because the dot notation is more readable than the nested calls used to build complex matchers in Hamcrest. However, we found the issues about method name choice to have a large impact on understanding. Thus, it is well possible that Hamcrest assertions are more understandable, worse notation notwithstanding, if the names of the matchers better express their meaning, w.r.t. the names of the assertion methods in AssertJ.

Another follow-up in depth study we plan is replicating this experiment with professional subjects, both tester and developer, to confirm our results; of course, we will use more complex assertions, given the greater knowledge of the subjects. Moreover, we want to compare the impact of familiarity with a testing style (in the case of testers), which could increase the fluent assertion score, vs. proficiency of code writing and reading (in the case of developers), which could make JUnit Basic the winner.

Finally, we plan to study the impact of the programming language on the understandability of assertions, by comparing analogous styles of assertions, or even better different versions of the "same" library, across different languages, like Java and C#. Another way to gauge the language impact, is to make analogous experiments in the two languages and compare the results. Thus, we plan to compare built-in assertions in nUnit[11], which are technically close to those in Hamcrest and Fluent Assertions[12] for C# language, to see if we get the same results as for Java.

[11] http://nunit.org/

[12] https://fluentassertions.com/

## References


[1] JUnit summary of changes in version 4.4, August 2007 (retrieved on 12th April 2018). http://junit.sourceforge.net/doc/ReleaseNotes4.4.html.

[2] Hamcrest vs AssertJ assertion frameworks - which one should you choose?, 2017 (retrieved on 15th April 2018). https://www.blazemeter.com/blog/hamcrest-vs-assertj-assertion-frameworks-which-one-should-you-choose.

[3] G. Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2011.

[4] K. Beck. *Test-driven development : by example*. Addison-Wesley, Boston, 2003.

[5] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer. Modeling readability to improve unit tests. In *Proceedings of 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 107–118. ACM, 2015.

[6] B. Dijkstra. Three practices for creating readable test code, 2016 (retrieved on 15th April 2018). https://www.ontestautomation.com/three-practices-for-creating-readable-test-code/.

[7] G. Grano, S. Scalabrino, R. Oliveto, and H. Gall. An empirical investigation on the readability of manual and generated test cases. In *Proceedings of 26th International Conference on Program Comprehension*, ICPC 2018, 2018.

[8] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition, 2005.

[9] S. Gulati. Write more understandable Java tests with matcher objects and FEST-assert, 2010 (retrieved 15th April 2018). https://www.developer.com/java/article.php/3901236/Write-More-Understandable-Java-Tests-with-Matcher-Objects-and-FEST-Assert.htm.

[10] B. A. Kitchenham, T. Dyba, and M. Jorgensen. Evidence-based software engineering. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE 2004, pages 273–281. IEEE, 2004.

[11] J. Kohl and B. Marick. Agile tests as documentation. In C. Zannier, H. Erdogmus, and L. Lindstrom, editors, *Proceedings of 4th Conference on Extreme Programming and Agile Methods*, pages 198–199. Springer, 2004.

[12] R. Mee and E. Hieatt. Going faster: Testing the web application. *IEEE Software*, 19:60–65, 03 2002.

[13] H. Motulsky. *Intuitive biostatistics: a non-mathematical guide to statistical thinking*. Oxford University Press, 2010.

[14] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, and P. Tonella. Using acceptance tests as a support for clarifying requirements: A series of experiments. *Inf. Softw. Technol.*, 51(2):270–283, Feb. 2009.

[15] F. Ricca, M. Torchiano, M. Leotta, A. Tiso, G. Guerrini, and G. Reggio. On the impact of state-based model-driven development on maintainability: A family of experiments using UniMod. *Journal of Empirical Software Engineering*, 23(3), 2018.

[16] F. Ryan. A look at unit testing frameworks, March 2018. https://redmonk.com/fryan/2018/03/26/a-look-at-unit-testing-frameworks/.

[17] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 3(52), 1965.

[18] T. Tahir, A. Jafar, S. Zaheer, M. K. Afzal, M. Ahmad, and J. Shafi. Understandability assessment of concrete and parametric test cases with respect to time and correctness ratio measures. *Journal of Basic and Applied Scientific Research*, 2013.

[19] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering - An Introduction*. Kluwer Academic Publishers, 2000.

[20] M. Wynne and A. Hellesøy. *The cucumber book : behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.