Journal of Computer Sciences

Original Research Paper

Fixture Setup through Object Notation for Implicit Test Fixtures

[1]Douglas Hiura Longo, [2]Beatriz Wilges, [1]Patrícia Vilain and [2]Renato Cislaghi

[1]Department of Informatics and Statistic, Federal University of Santa Catarina, Florianópolis, Brazil
[2]Department of Engineering and Knowledge Management, Federal University of Santa Catarina, Florianópolis, Brazil

Article history
Received: 18-02-2015
Revised: 21-5-2015
Accepted: 02-06-2015

Corresponding Author:
Douglas Hiura Longo
Department of Informatics and Statistic, Federal University of Santa Catarina, Florianópolis, Brazil
Email: douglashiura@gmail.com

Abstract: This paper presents an approach to the development of fixture setup code through an objects notation that is applied on implicit test fixtures. This approach is integrated with a management mechanism to call the fixture setup code from the JUnit test framework. The objective of this work is to enable the reuse of fixtures across multiple test classes avoiding the management and the creation of fixtures within the test itself. The evaluation of this proposal was performed during test-driven development of a Web-based system. Results present approximately 190 fixture setups with a reutilization average of about 13 times, observed in an analysis with 2200 h of development. Initial results show the growing reutilization of fixture setups during test development, with significant test code volume reduction.

Keywords: Fixture Setup, Test Fixture, JUnit, Test-Driven Development, Object Notation

Introduction

Modern practices of software development encourage extensive code testing during development stages (Bagge *et al*., 2010). Additionally, Beck (2003) stated that automatized testing for Test-Driven Development (TDD) is a key factor as it influences the success and improvement of productivity during software design and development (Stober and Hansmann, 2010; Hunt *et al*., 2014). However, automatized testing often depends on the testability of Systems Under Test (SUT) (Blackburn *et al*., 2004).

The xUnit family of automatized tests is widely known for TDD. Each test in the xUnit framework is represented by a test method, which implements 4 stages: Test fixture configuration, SUT exercise, results verification and fixture clean up (Meszaros, 2007).

An important part of the test is the code initializing the SUT (Greiler *et al*., 2013a; 2013b), named fixture setup, which puts all elements on the required state to perform SUT (Beck, 2003; Meszaros, 2007; Louridas, 2005). The necessary elements to perform SUT are called test fixtures (Meszaros, 2007).

Developers find in the xUnit framework several types of builds for fixture setup. Usually, the code to build test fixtures is: In-line setup, delegate setup and implicit setup.

The code for in-line setup is written directly onto the test method. In this sense, fixture setup isolates tests, resulting in code duplication. Duplicate code can be moved onto an auxiliary method. According to Meszaros (2007), auxiliary method can be called by a few test methods named delegate setups.

The xUnit framework also has mechanisms dedicated to the management of calls in code fixture setup. These mechanisms call auxiliary methods by giving them specific names (e.g., setup), annotations (e.g., "@before" in JUnit), or method attributes (e.g., "[Setup]" in NUnit) (Meszaros, 2007; Greiler *et al*., 2013a; 2013b). The calling of the auxiliary method occurs implicitly at a specific moment. According to (Greiler *et al*., 2013a), this is called implicit setup.

This study proposes to involve object notation language to set up fixtures, as well as a mechanism to call and manage implicit fixture setups. Our main objective is to seek a strategy which diminishes code redundancy used in fixture setup, ultimately resulting in a clearer and more cohesive code.

The main contribution of this article is the implementation of this proposal through a tool integrated to the JUnit test automation framework. For its assessment, the tool was used by 5 developers applying TDD on the development of a Web-based system. This article shows how, in practice, fixture setup is increasingly reutilized with the tool during development stage. The greatest impact resulting from the tool is lower maintenance on test code volume.

Our article is organized as follows: Presents studies related to this research; The proposal in detail; The implementation; The assessment method used in verifying the proposal; The results obtained with this study; Discussion about study case and, lastly, final remarks on the proposal are presented.

Related Works

According to Beck (2003), it is recommended to apply TDD with simple tests of test code complexity 1, that is, no method or loop calls. According to Fraser *et al*. (2003) tests must be as simple as possible, written mainly for design and specifications.

Beck (2003), presents two test fixture creation styles named by Meszaros (2007), of in-line and implicit fixture setups. Implicit fixture setup is a style strategy that considers test framework mechanisms for fixture setup. This strategy avoids fixture setup code redundancy, implicitly sharing called auxiliary methods.

Another strategy to avoid code redundancy is delegated fixture setup (Meszaros, 2007). In this strategy, duplicated code is extracted onto a method, which some may call a test.

Depending on the fixture setup build strategy adopted during test code development and evolution, problems may appear. Some of these problems are called test smells (Meszaros, 2007; Greiler *et al*., 2013a; 2013b). According to Van Deursen *et al*. (2001), factoring the test code involves factoring other tests and that may cause a set of bad smells. Greiler *et al*. (2013a) has developed a test smell potential analysis tool named Test Hound. In order to avoid test fixture smells during software evolution, (Greiler *et al*., 2013b) proposed a tool named Test Evo Hound.

According to Schuh and Punke (2003), standards such as Object Mother and Test Data Builder are used in fixture setup codes for object creation. These standards promote object reutilization; however, they are written as a set of calls that are often difficult to understand.

Proposal

The proposal developed in this study is named Picon. Picon is a fixture setup strategy that allows the organization and reutilization of test fixtures during TDD. The technique proposed by Picon is a mark-up mechanism for test fixtures implicit in test classes. Therefore, fixture setup is defined through object notation language, which can be shared among several test cases.

Figure 1 presents two test classes with implicit fixtures (Test Flight by Field Setup and Test Flight by Method Setup). Note that the from Brazil to Roma fixture used in these tests was configured with Picon notation language and is illustrated on Fig. 2.

As can be seen in Fig. 1, test class codes Test Flight by Field Setup and Test Flight by Method Setup have not explicitly stated fixture setup within their codes. Alternatively, these test classes have only implicit test fixture mark-ups. These mark-ups are made in two ways:

- Stated attribute within the test class (line 3, Fig. 1 from Brazil To Roma); or
- Parameter (line 13, Fig. 1“from Brazil To Roma”) for the get method within the test method

In this sense, the from Brazil to Roma attribute and the “from Brazil to Roma” parameter in the get method are considered implicit test fixtures. Implicit fixture setup is defined by the Picon notation, as shown in Fig. 2. Thus, the fixture setup of Picon for test performance is linked to the implicit fixtures of the Test Case class. Running of the Test Case class with fixtures is detailed in Subsection A. Specificities in the Picon notation language for fixture setup are presented in Subsection B.

A. Test Case Class Running with Implicit Fixtures

An application which instances a Test Suite object running its Test Case objects is defined in order to run the Test Case class (Meszaros, 2007). However, in order to properly run the Test Case object, at the initial SUT time fixtures defined by the Picon notation are created. Thus, the test automation framework is responsible for managing and properly creating the implicit test fixtures, according to the Test Case class.

Figure 3 is adapted from Meszaros (2007), presenting the mechanism to run the Test Case class according with the proposal of this study.

Test Context object performs the creation of implicit test fixtures. In addition, implicit test fixtures of Test Case class are created only to run the Test Case object. The Picon strategy is to provide a Test Context object for the Test Suite object to run the Test Case object. During test run, the implicit test fixtures are objects properly created according with the fixture setup through the notation. In this sense, the implicit test fixtures of a Test Case class are created as objects by a Test Context object.

Picon fixture setups are named uniquely, which allows links with the implicit test fixtures stated on the Test Case class. This link is made during runtime, when objects are created by reflection according with the Picon fixture setup and later installed onto the Test Case object. Thus, the Test Case object runs properly without the fixture setup code within the Test Case class.

B. The Picon Notation Language for Fixture Setup

Implicit test fixtures are configured using an object notation language based on JSON or ECMA-262, as shown in Fig. 2. Picon fixture setups are completely independent of the programming language, free from algebraic operations, conditions, loops and procedures, utilized specifically for data statement.

```
1  public class TestFlightByFieldSetup extends TestCase {
2       /* implicit  test fixture*/
3       private Flight fromBrazilToRoma;
4       @Test
5       public void testOfImplicitField(){
5               assertTrue(fromBrazilToRoma.isScheduled());
7       }
8  }
9  public class TestFlightByMethodSetup extends TestCase {
10      @Test
11      public void testOfImplicitParameter(){
12              /* implicit test fixture */
13              Flight fromBrazilToRoma = get("fromBrazilToRoma ");
14              assertTrue(fromBrazilToRoma.isScheduled());
15      }
16 }
```

Fig. 1. Implicit test fixture in test class

```
1  Flight { /*ClassName*/
2    fromBrazilToRoma[date="2015/04/13" from="BRA" to="ROMA"] /*Fixture setup*/
3  }
```

Fig. 2. Fixture setup by the Picon object notation language

They are created in *.picon* extension files. Each file must be structured as seen in the example of Fig. 2. Fixtures are listed for each class and for each of these fixtures a set of name/value pairs is defined. In the example on Fig. 2, the "Flight" class (line 1) contains the fixture "from Brazil To Roma" and the fixture "from Brazil To Roma" contains the set of name/value pairs date="2015/04/13", from="BRA", to="ROMA". Therefore, the notation language grammar is structured in classes containing a set of fixtures, which, in turn, store a set of name/value pairs.

Figure 4 illustrates the grammar of a class, starting with its name, followed by the opening bracket, a set of fixtures and a closing bracket. Figure 5 presents the grammar of a Picon fixture.

The grammar of a Picon fixture starts with its name, followed by its configuration stated within the opening and closing brackets. The configuration is comprised by a set of key/value brackets. The bracket must be a class attribute of the fixture, which is configured with a value. The types of values can be: Strings, integers, floats, dates, booleans, arrays, enums and references for other fixtures.

Fixture references resolve during the test runtime, that is, there must be fixtures linked for each reference. This linking is made when the reference value is equal to the name of a fixture. Therefore, references allow for the composition among fixtures.

Implementation

The Picon proposal was implemented through a tool integrated to the JUnit test automation framework that is available for download. The tool implements an API for .picon file processing and integration with JUnit.

The API developed is based on JSON, which aids file manipulation and object creation. Files are found within the test project and objects are created according to fixture setups.

The objective of integrating JUnit is to create a mechanism to manage and run the test appropriately. To this end, some JUnit framework classes were modified, in which we have overwritten the create Test method of the Block JUnit 4 Class Runner class. This method then began supplying a Test Context object for the framework. The Test Context object is a Test Case object proxy. The Test Context object memorizes the Test Case object through reflection, according with the fixtures supplied by the file manipulation API. After fixtures are built, the test is run by JUnit.

The JUnit framework integrated with the tool can also be used with Eclipse IDE. In this sense, there are no changes on the traditional fixture setup build, only on the availability of the proposed strategy.

Figure 6 presents two test examples run by the Eclipse development IDE with the JUnit framework integrated to the tool.

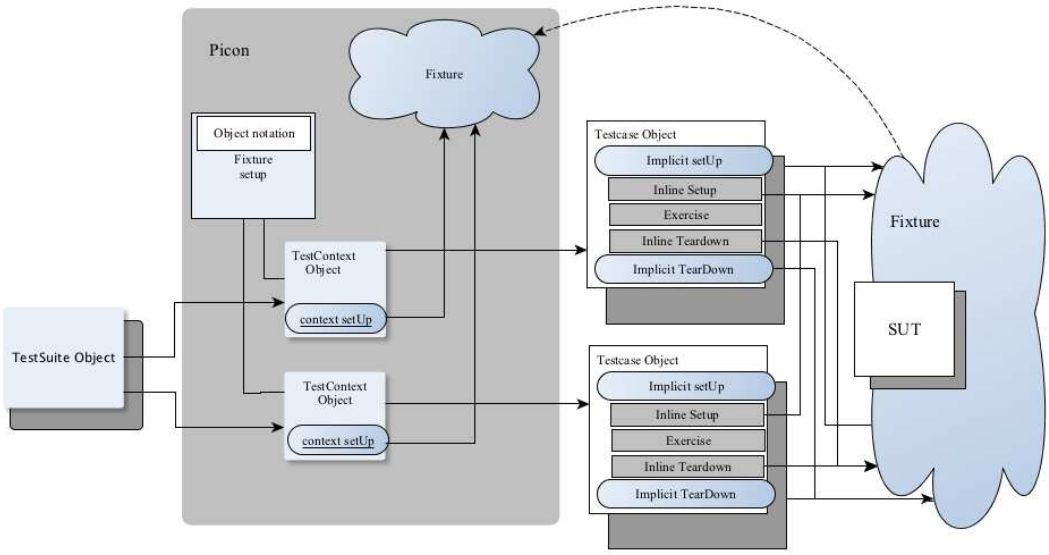

Fig. 3. Picon mechanism for fixture setup of implicit tests during the SUT run

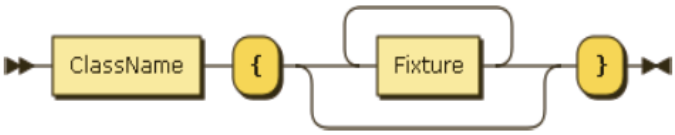

Fig. 4. Grammar of a class with Picon fixture

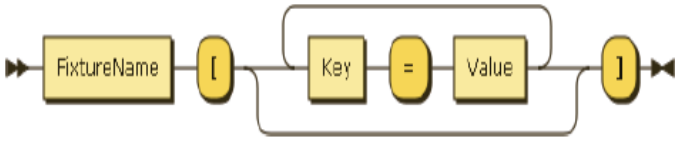

Fig. 5. Grammar of a class with Picon fixture

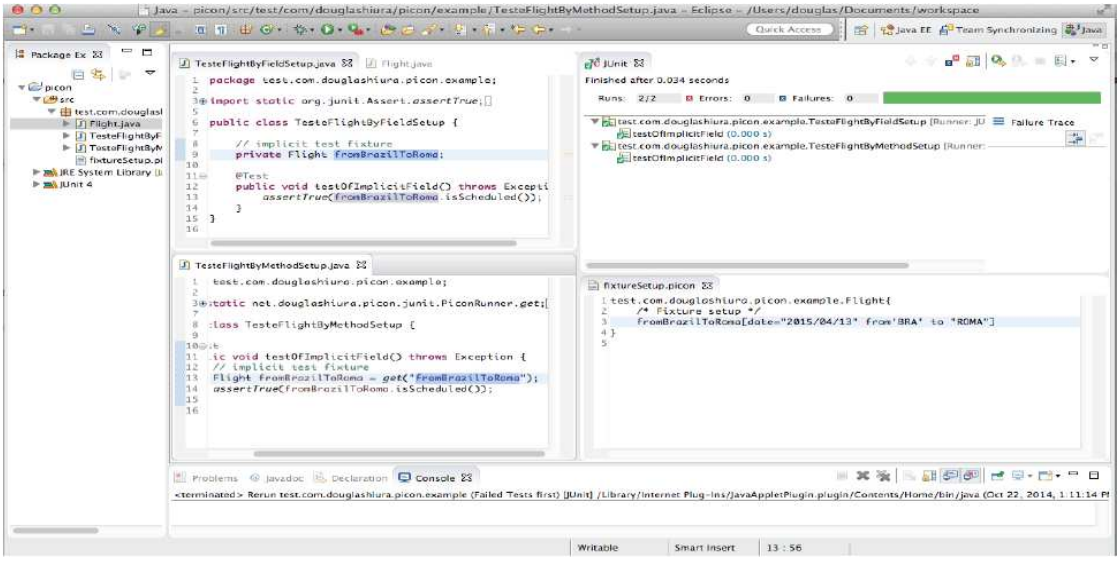

Fig. 6. Example of test run with implicit fixtures configured by object notation

Assessment Method

In order to evaluate the applicability and utility of the proposal, the following questions have been investigated:

- RQ1: Which is the fixture setup reutilization rate?
- RQ2: What is the proposal tendency (frequency) of utilization during development?
- RQ3: What is the test code volume with Picon in relation to test code volume without Picon?

The Picon proposal was assessed in the development of a Web-based system. This system was developed to evaluate and follow up on the e-Tec Brasil network courses (Cislaghi *et al*., 2014). The system holds a data base with questionnaires and results on the evaluations of Brazilian federal institutes on Education, Science and Technology. The system was developed through Java for Web technologies, as well as with the PostgreSql Data Base management system.

The tool integrated to the test automation framework was applied during system development, followed by the TDD practices. The development team was composed of 5 programmers with little to no experience with TDD. In this sense, the programmers were trained for 80 h in order to obtain TDD practice. After the training period, system development activities were started. The development team dedicated an average of 22 h/day.

During development, the programmers were supervised and advised in relation to TDD practices, coding in pairs. In addition, tests were developed through system requirement statements. Thus, fixtures were used in almost every test in order to follow previous conditions for test run. Programmers were advised to write simple tests during development maintenance and evolution state, refactoring both tests code and their application code.

During development, developers have applied other hybrid fixture setup strategies supported by the Junit framework, such as the implicit setup and the in-line setup presented in Fig. 3.

Proposal assessment considered development evolution of a midrange project. According to Ress *et al*. (2003), a midrange project has between 2000 and 3999 h of development. Thus, proposal assessment was performed in 10 intervals of 220 h each, totalizing 2200 h of development. That is, data was collected in the following time intervals:

$$Time = \{220, 440, 660, \ldots, 2200\}$$

For each interval during development, the following variables were extracted:

- Test Case = amount of test cases
- Fixture Setup = amount of fixture setups related to the proposal
- Test Fixture = amount of reutilized fixture setups
- Application code = numbers of application code lines
- With Picon = number of test code lines with Picon
- Without Picon = number of test code lines without Picon

The strategy to obtain the number of test code lines without Picon was to replace a fixture setup proposed by traditional fixture setups.

Thus, the number of lines in test code without Picon was collected through existing test classes. These classes were changed to calculate the number of lines in test code without Picon. The change of the test classes required the addition of more lines in the code. This addition of lines was performed using an algorithm. From this code it is possible to extract the number of lines to perform the necessary comparisons. Figure 7 shows an example of a test code to illustrate the operation of the algorithm, which obtains the code of the tests without Picon. Figure 8 shows the Picon Fixture Setup used in the code from Figure 7. Figure 9 shows the same code without Picon.

```
1 public class TestWithPicon {
2         private User mary;
3         private Store store;
4
5         @Before
6         public void setUp_fixtureSharedInClass() {
7                 store = Store.getInstance();
8         }
9
10        @Test
11        public void filterByName() throws Exception {
12                store.add(mary);// in line fixture
13                List<User> users = store.filterByName("mary");
14                assertEquals(1, users.size());
15                assertEquals(mary, users.get(0));
16        }
17 }
```

Fig. 7. An example of a test code developed with the tool

```
1 User{
2          mary [name = "Mary" ]
3 }
```

Fig. 8. Picon Fixture Setup

```
1 public class TestWithoutPicon {
2          private User mary;
3          private Store store;
4
5          @Before
6          public void setUp_fixtureSharedInClass() {
7                    mary = new User();
8                    mary.setName("Mary");
9                    store = Store.getInstance();
10         }
11
12         @Test
13         public void filterByName() throws Exception {
14                   store.add(mary);// in line fixture
15                   List<User> users = store.filterByName("mary");
16                   assertEquals(1, users.size());
17                   assertEquals(mary, users.get(0));
18         }
19 }
```

Fig. 9. Test code generated by the algorithm without Picon

The section highlighted in Fig. 9 (lines 7 and 8) illustrates the lines added in code developed without Picon. It is emphasized that the procedure shown in Fig. 7, 8 and 9 is only an illustration of the algorithm applied in all test code. Thus, we can only find the test code volume relative to development time with the proposal. The test code volume without the proposal is relative to the application code.

A number of software reuse metrics have been suggested in the literature. Various categories of metrics can be found, such as cost-effort analysis, maturity assessment models, amount of reuse, reutilization, among others (Frakes and Terry, 1996; Poulin and Caruso, 1993). According to Patel and Kollana (2014), most metrics can be adapted to measure reuse and have utility from varying points of view, but it would be costly to implement them all.

According to Frakes and Terry (1996) as well as Poulin and Caruso (1993), the metrics calculating the amount of reuse defines a percentage of reuse in relation to the number of case tests reused, divided by the total number of test cases in a project. Thus, in this study, the metrics calculating the amount of reuse defines a percentage of reuse in relation to the number of case tests reused, divided by the total number of test cases in a project.

Research Questions are answered according with the collected data, through the following analyses:

- (RQ1) Fixture setup reuse can be answered by the proportion analysis between data from the fixtures setup and test fixtures variables
- (RQ2) Proposal use tendency during development can be answered by the linear correlation coefficient ("Pearson r") among variable pairs: Time and test fixtures; time and fixture setup
- (RQ3) The difference between code volume is calculated by comparing variables with Picon and without Picon in relation to the same application code

Proposal Assessment Results

Results in this section present a synthesis on the applicability and utility of the proposal, answering the Research Questions. The Fig. 10 presents a line chart with the evolution over time of variables: Test fixture, fixture setup and test case.

The proposal tool was introduced in the project after the first 230 h of development; thus, the first results appear only at data collection time of 440 h. Requisite changes have occurred during the project. The most evident requisite changes occurred between 880 and 1100 h, which caused the removal and adjustments of tests. These changes have impacted the amounts of all three variables of the Fig. 10 chart.

The test fixture variable curve presented on Fig. 10 illustrates the reuse of fixture setups (RQ1). Each fixture setup was used in an average of 13.78 times during the 2200 h of development.

A key factor to correct issues due to multiple case test flaws for editing fixture setups is to maintain the tests simple, with few fixtures. However, each test case uses an average of 2.05 test fixtures and this number increases constantly during development evolution. This means that the test cases may have become more complex during the project.

The Microsoft Excel data analysis tool was utilized to calculate the linear correlation coefficient, as well as to determine the equation to adjust the variable pairs: Time and test fixtures and time and fixture setups. The variables test fixture and time have a correlation coefficient of r = 0.95. The correlation coefficient greater than zero indicates a positive correlation, that is, there was crescent reuse tendency of fixture setups during development evolution. The coefficient of determination (r^2 = 0.91) above 0.70, indicate that the variables of test fixture and time have properly adjusted to the linear equation model (Test fixture = 267 time-104).

The variables fixture setup and time have a linear correlation coefficient of r = 0.94. This result shows the crescent tendency of fixture setups during development. Additionally, the coefficient of determination (r^2 = 0.90) indicates that the linear equation (Fixture setup=20 time -10) has strong adherence to the variable data.

The chart of Fig. 10 shows some decreasing sequences. However, the proposal utilization tendency during development was crescent through the linear correlation coefficient analysis over time (RQ2).

The 2200 h of development have generated 10540 lines of application code. The line chart on Fig. 11 illustrates the difference of test code volume with and without Picon, in relation to the application code volume (RQ3).

The average difference between the variables with and without Picon was of 9740 lines of code. However, according to the line chart on Fig. 11, the area of the difference between both variables increases according to the volume of application code.

The chart on Fig. 12 presents the code volume after approximately 6000 h of development for variables: Application code, with Picon and without Picon.

The proportion between the number of application code lines and the number of test code lines without Picon was of approximately 1 application code line for each 10 test code lines. However, the proportion between the number of application code lines and the number of test code lines with Picon was of approximately 1 application code line for each 4 test code lines. Thus, by applying the proposal after approximately 6000 h of development, only 40% of test code volume is necessary in relation to traditional test code volumes.

Thus, the proportion between the numbers of test code lines implemented for each application line shows the strong potential that the Picon development proposal can offer. This result is interesting as it diminishes significantly the number of test code lines elaborated.

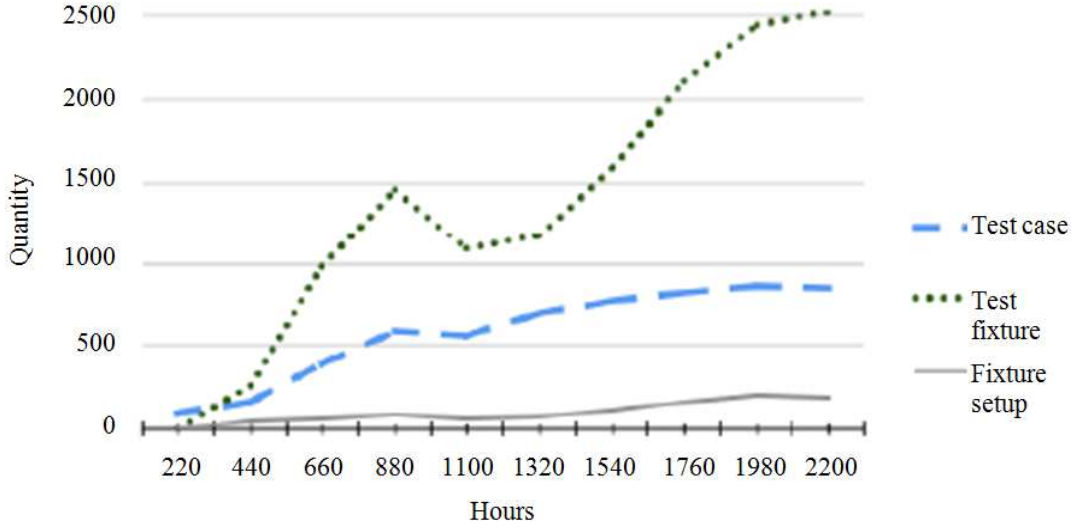

Fig. 10. Line chart of the evolution over time of variables: Test fixture, fixture setup and test case

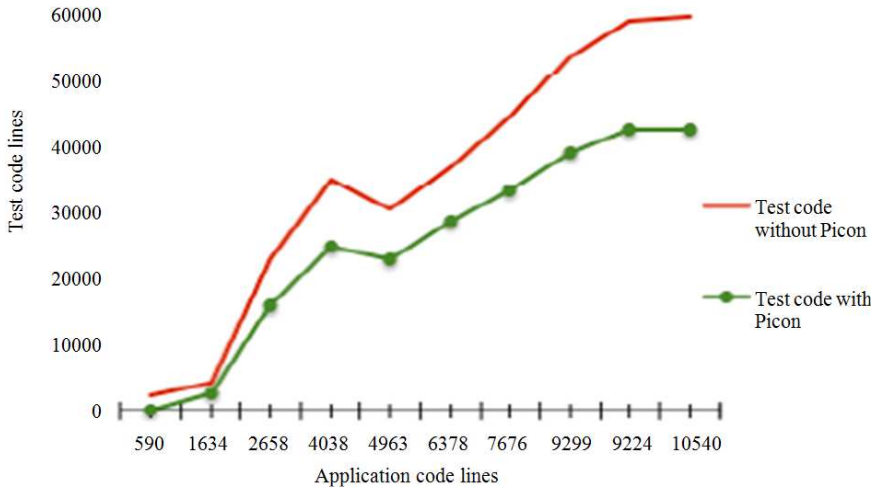

Fig. 11. Line chart of code volume: Without Picon versus with Picon

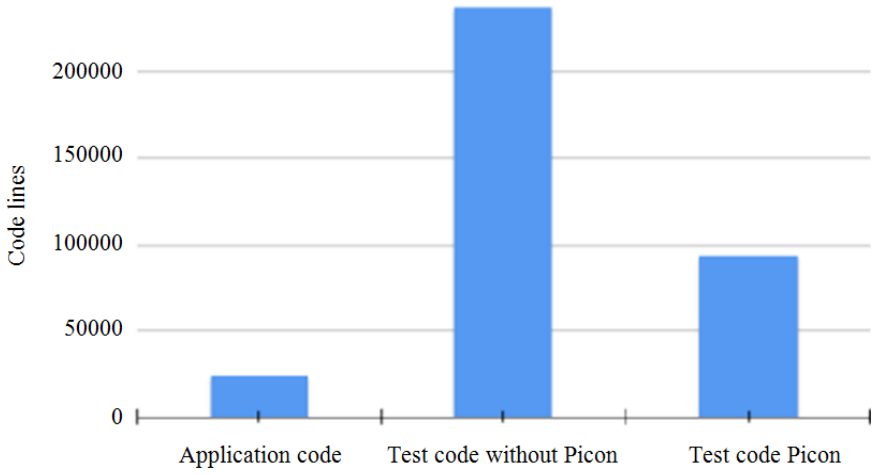

Fig. 12. Chart of code volume

Discussion

During the development, TDD practices (Beck, 2003) were adopted. However, it was not possible to design isolated tests because shared fixtures were used. Two positive points on reusing the fixture setups were: Less time between test writing and running; and fewer code lines within the test class. However, shared fixture setups have broken isolation among tests; that is, if there were fixture setup edits, flaws can occur in several test cases.

The main principles adopted in TDD:

- Assert first-this technique has a powerful simplifying effect during test development (Beck, 2003). This technique was a good practice to avoid unnecessary fixtures
- Factoring-was adopted both in the test code and in the application code. Thus, it was possible to reduce the volume of code maintained
- Simple tests-the test code must be linear, without deviation, conditions, loop or branches
- Simple Assertions-simple assertions are easy to read and maintain. It is therefore discouraged the use of narrative assertions, for example, assert That (new Array List().size(), is (0))
- Code coverage test - it is important to test 100% of the application code. Considering that testing 100% of the application states is impossible. Thus, it is recommended to avoid extensive tests and include tests that fail

In the context of the proposed fixtures, developers usually adopt the general name "qualifier" to identify the fixture Name (Fig. 5). In this project it is encouraged to use proper names for "qualifiers", as in the fixture Mary (Fig. 9). Avoiding to include the fixture type for the fixture name, for example: User Mary.

In addition, it can be understood that fixture setup reuse enables the code to be cleaner and more cohesive, since it is no longer necessary to rewrite each fixture used on tests. Thus, it is possible that semantic errors occurring during project feature development can be more easily found and more importantly, avoided.

The evaluation of the proposal explicitly involves the use of TDD practices. In particular, the proposal is unaware of the results of its use with other types of automation tests, such as test after.

Conclusion

In this study we have presented a proposal for fixture setup through object notation for implicit test fixtures. The proposal was implemented through a tool integrated to the JUnit framework. The tool was applied on software development with TDD practices. During software development, tool application was assessed. At this stage, we have investigated the reuse of fixture setups, the tendency of use and the test code volume. These factors can be perceived mainly on the graphs illustrating the amount of fixtures that are reusable over time during implementation.

Results show the growth of the use of the proposal tool, which indicates its need during development. It is possible to observe that the application of this proposal results in increasing reuse of fixture setups, diminishing redundant code in test classes. The most evident impact is the reduction of test code volume, avoiding efforts to create and maintain it.

The main contribution of this study is the implementation of a tool, according to the proposal, as well as the investigation in test fixture reuse through object notation with TDD practices.

In future research we plan to investigate:

- Fixture setup outside of test class, presenting solutions and practices in order to improve the test project
- The execution time of the tests with the implementation of fixtures mechanism. Thus it may be possible to improve the implementation to reduce execution time

Acknowledgement

We thank the development team and the Federal University of Santa Catarina that provided insight and expertise. Special thanks to Professor Luiz Fernando Melgarejo Bier by introduction of TDD in classes of programming oriented objects. Thanks to Rogerio Bagatini by Picon development collaboration.

Funding Information

Authors would like to thank the Ministry of Education for providing the financial support through scholarship from Coordination for the Improvement of Higher Education Personnel (CAPES).

Author's Contributions

Douglas Hiura Longo: Design the research and prepare the workflow. Undertake the required experiments and analyse the obtained results.

Beatriz Wilges: Organizes the writing and structure of the manuscript.

Patricia Vilain: Research advisor.

Renato Cislaghi: Development project coordinator.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript and no ethical issues involved.

References

Bagge, A.H., V. David and M. Haveraaen, 2010. The axioms strike back: Testing with concepts and axioms in C++. ACM Sigplan Notices, 45: 15-24. DOI: 10.1145/1837852.1621612

Beck, K., 2003. Test Driven Development: By Example. 1st Edn., Addison-Wesley, Boston, ISBN-10: 0321146530, pp: 220.

Blackburn, M., R. Busser and A. Nauman, 2004. Why model-based test automation is different and what you should know to get started. Proceedings of the International Conference on Practical Software Quality and Testing, (PSQT' 04), pp: 212-232.

Cislaghi, R., B. Wilges, S.M. Nassar, D.L. Hiura and G.P. Mateus, 2014. Avaliação de polos sob uma perspectiva georreferenciada. Proceedings of the 11th Congresso Brasileiro de Ensino Superior a Distância (ESUD' 14), Florianópolis, pp: 771-781.

Frakes, W. and C. Terry, 1996. Software reuse: Metrics and models. ACM Comput. Surveys, 28: 415-435. DOI: 10.1145/234528.234531

Fraser, S., D. Astels, K. Beck, B. Boehm and J. McGregor, 2003. Discipline and practices of TDD: (Test Driven Development). Proceedings of the Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. ACM, pp: 268-270. DOI: 10.1145/949344.949407

Greiler, M., A. Van Deursen and M.A. Storey, 2013a. Automated detection of test fixture strategies and smells. Proceedings of the 6th International Conference on Software Testing, Verification and Validation, Mar. 18-22, IEEE Xplore Press, Luembourg, pp: 322-331. DOI: 10.1109/ICST.2013.45

Greiler, M., A. Zaidman, A. Van Deursen and M.A. Storey, 2013b. Strategies for avoiding text fixture smells during software evolution. Proceedings of the 10th IEEE Working Conference on Mining Software Repositories, May 18-19, IEEE Xplore Press, San Francisco, pp: 387-396. DOI: 10.1109/MSR.2013.6624053

Hunt, C.J., G. Brown and G. Fraser, 2014. Automatic testing of natural user interfaces. Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation, March 31-April 4, IEEE Xplore Press, Cleveland, pp: 123-132. DOI: 10.1109/ICST.2014.25

Louridas, P., 2005. JUnit: Unit testing and coiling in tandem. Proceedings of the IEEE Software, 22: 12-15. DOI: 10.1109/MS.2005.100

Meszaros, G., 2007. XUnit Test Patterns: Refactoring Test Code. 1st Edn., Pearson Education, ISBN-10: 0132797461, pp: 944.

Patel, S. and R.K. Kollana, 2014. Test case reuse in enterprise software implementation-an experience report. Proceedings of the IEEE Seventh International Conference on Software Testing, Verification and Validation, Mar. 31-Apr. 4, IEEE Xplore Press, Cleveland, pp: 99-102. DOI: 10.1109/ICST.2014.22

Poulin, J.S. and J.M. Caruso, 1993. A reuse metrics and return on investment model. Proceedings of the Advances in Software Reuse., Selected Papers from the 2nd International Workshop on Software Reusability, Mar. 24-26, IEEE Xplore Press, Lucca, pp: 152-166. DOI: 10.1109/ASR.1993.291707

Ress, A.P., R. Oliveira Moraes, M.S. Salerno, 2003. Test-driven development as an innovation value chain. J. Techn. Manage. Innovation, 8: 10-10.

Stober, T. and U. Hansmann, 2010. Agile Software Development. 1st Edn., Springer, ISBN: 10- 9783540708308. pp: 179.

Schuh, P. and S. Punke, 2003. Easing Test Object Creation in XP. XP Universe.

Van Deursen, A., L. Moonen, A. van den Bergh and G. Kok, 2001. Refactoring test code. CWI.