Reuse of Fixture Setup between Test Classes

Lucas Pereira da Silva
Informatics and Statistics Department
Federal University of Santa Catarina
Florianópolis, Brazil
pslucasps@gmail.com

Patrícia Vilain
Informatics and Statistics Department
Federal University of Santa Catarina
Florianópolis, Brazil
patricia.vilain@ufsc.br

Abstract—**In this paper, we describe commonly used fixture setup strategies as well as their disadvantages and advantages. We propose a dependency model and a test fixture sharing model that allow the definition of a new fixture setup strategy. This strategy promotes code reuse by sharing fixture setups between test classes. The models are evaluated through a case study where the new fixture setup strategy presented a reduction of 47,62% in the fixture setup code.**

Keywords-software testing, unit testing, fixture setup; test fixture; test dependencies; test code reuse

I. INTRODUCTION

Software testing is an important activity of the software development process. The notion of its importance has evolved in recent years. Software testing is no longer an activity that would only start after the coding phase. It is now carried out during the entire development process [4]. In that sense, software testing incorporates other purposes, such as to prevent bug inclusion [1], and to be a way for specification [3] and documentation design [9]. According to [12], tests should be easy to run (automated, self-checking and repeatable), easy to write/understand (simple, expressive and with separation of concerns), and easy to maintain (robust). The satisfaction of these requirements is important to ensure a cost-effective testing activity.

When the software testing starts at early stages of development, some extra challenges appear. According to [11], not only the production code, but also the test code must be maintained. So, it is also necessary to maintain the tests throughout the entire development process [5] [14]. Test maintenance may end up having a high impact on overall testing costs, even higher than the cost of their initial implementation [2]. Therefore, test requirements mentioned above become even more important.

According to [8], the success of automated testing is strongly influenced by the maintainability of the test code. To promote maintainability, the tests should be clearly structured, well named and small in size [8]. Furthermore, code duplication across the tests must be avoided [8]. Tests that are hard to write and understand may suggest a poor system design.

According to [7], each test case is usually separated in a test method. A test case simulates a use scenario of the SUT. In this sense, a test case has to put the SUT in a state that represents the use scenario, that is, a state of interest to the test [11]. This is done through the execution of the test fixture setup code. According to [12], a test fixture represents anything necessary to exercise the SUT. The code where test fixtures are created is called fixture setup.

Reference [12] presents several strategies for fixture setups. Among these strategies, we highlight the inline setup, implicit setup and delegate setup. Each strategy has a different impact in the following properties of the test code: simplicity (tests should be small and verify one thing at a time), expressiveness (tests should communicate intention, i.e., should be easy to understand what behavior a test wants to verify), separation of concerns (each test should focus only on a single concern of the system) and robustness (overlap between tests should be avoided in such a way that small changes on the production code do not affect a large amount of tests). It is important to note that the robustness is directly influenced by code duplication. Thus, avoiding the duplication of test code can help to reduce the development effort and, consequently, to create tests with better maintainability.

The objective of this work is to propose a new fixture setup strategy that may be used as an option to promote better robustness of the testing code without affecting its other properties. We also propose a model to represent this new fixture setup strategy. Thus, test frameworks may incorporate this fixture setup strategy by implementing the proposed model. It is important to point out that our intention is not to propose a strategy that is superior to the others, but to present a new strategy that helps developers to create better tests with less effort.

The sections of this paper are organized as follows: in Section 2 we present the most known fixture setup strategies in the literature; in Section 3 we propose a model that allows the reuse of fixture setups between test classes; in Section 4 we present a case study; in Section 5 we evaluate, through the use of framework Story implemented, the proposed model; and in Section 6 we present the conclusions of our work.

II. FIXTURE SETUP STRATEGIES

Meszaros [12] introduces the most used fixture setup strategies, mainly in the frameworks of xUnit family. The strategies are categorized as fresh fixture setup, where the fixture setup is run before at each test, or as shared fixture construction, where the fixture setup is run only once before a given group of tests. Both categories have strategies that allow the reuse of fixture setups (i.e., reuse of code). However, only

10.18293/SEKE2017-181

in the shared fixture construction is possible to reuse test fixtures (i.e., reuse of execution).

A. *Fresh Fixture Setup*

The fresh fixture setup category is composed by three strategies: inline setup, implicit setup and delegate setup. According to [12], an adequate combination of these three strategies may increase the code reuse and, even so, preserve the desired properties of simplicity, expressiveness and separation of concerns. Next, these three strategies are presented, as well the main advantages and disadvantages of each one.

1) Inline Setup: in this strategy the fixture setup is placed inside the test method. In general, this strategy is chosen when a test needs a very specific test fixture or during the beginning of the development of the test code when duplication of code does not exist yet. An advantage of this strategy is a better understanding of the relationship between the test fixtures and the behavior that the test is verifying, since the fixture setup is near to the test verifications. However, its major disadvantage is the duplication of code among test methods.

2) Implicit Setup: In this strategy the fixture setup is placed in a special method (usually called setup method) of the test class. The test framework is responsible for running the setup method before the execution of each test method existing in the test class. The reuse of the fixture setup is the major advantage of this strategy. However, higher is the amount of test methods in the same test class, harder is to preserve the properties of simplicity, expressiveness and separation of concerns. When many test methods are created, it is more difficult to understand the relationship between the test fixtures and the intention of each test, once some test fixtures may not be needed to all tests.

3) Delegate Setup: in delegate setup, the code is placed in helper methods. In general, these methods are put in auxiliary classes, apart the test methods. Promoting fixture setup reuse between distinct test classes is the major advantage of this strategy. However, this strategy may demand high labor. Helper methods and classes must be created and maintained by developers. Thus, well-named helper methods are essential in order to preserve the expressiveness. Furthermore, in contrast with inline setup and implicit setup strategies, the framework does not call automatically the delegate setup.

B. *Shared Fixture Setup*

The strategies of the shared fixture category depend on a place to save test fixtures that are created. This place may be a file system, a database or even a static field. The fixture setup runs once for a given group of tests and then the created test fixtures are saved. Thus, the tests of that given group may access the test fixtures. The definition of when the fixture setup will be run varies according to each strategy.

III. PROPOSAL

Test code duplication is related to the robustness property, which in its turn is related to test code maintainability. Thus, reducing the test code duplication may improve the test code maintainability without impacting negatively in simplicity, expressiveness and separation of concerns. Strategies as implicit setup and delegate setup help to reduce the test code duplication. However, sometimes, to reduce the duplication of test code without affecting other properties may be a hard task. Higher is the amount of tests, harder is to maximize the test code reuse by clustering test classes according to the test fixtures (implicit setup) and harder is to maximize the test code reuse by creating many helper methods as test fixtures needed (delegate setup).

In this work we propose the definition of a model to represent the dependence between test classes in order to create a new fixture setup strategy. The proposed model allows the reuse of fixture setups between different test classes. The main objective is to increase the tests robustness by increasing the reuse of fixture setups without reducing the general tests maintainability.

The central assumption of our proposal comes from the simple observation that a given fixture setup execution may drives the SUT exactly to a specific state of another test class. It is crucial to clarify that the explicit definition of dependence between test classes, proposed by this work, does not break the independence principle described in [13].

A. *Dependency Model between Test Classes*

The dependency model between test classes considers that a test class may depend on one or more test classes. The definition of dependence must be done in the dependent class. The dependency/dependent relationship between two test classes means that the dependency class has the fixture setup required by the dependent class.

Fig. 1 shows a hypothetical implementation of the dependency model, using the Java language. The annotation @FixtureSetup is placed in the declaration of the test class UserTest (dependent) to indicate a dependency with the test class CleanDatabaseTest (dependency). Thus, each test run of the dependent class should be preceded, in the given order, first by the run of the fixture setup of the class CleanDatabaseTest and then by the run of the fixture setup of the test class UserTest. This enables the reuse of fixture setup between different test classes.

```
@FixtureSetup(CleanDatabaseTest.class)
class UserTest { ... }
```

Figure 1. Annotation @FixtureSetup.

B. *Independence Principle*

Independence principle, described in [13], says that each test should be independent. It should be possible to run each test individually or through a suite of tests in an arbitrary order. This principle is justified by the fact that a given test run should not affect another test run. If a test fails because a previously test had let the SUT in an inconsistent state, then the independence principle is violated.

It is important to clarify that the proposed dependency model does not violate the independence principle. The dependency model implies only in dependence between test classes, but not between tests. In the dependency model, before each test run, the test framework has to run all the needed fixture setup.

C. Multiple Dependencies

Fig. 2 shows an example where a test class depends on two test classes. Fixture setup run will follow the order in which the dependency classes are declared (i.e. the order of the dependency classes in the annotation @FixtureSetup).

Fixture setups defined in different test classes can be combined to construct a new fixture setup. In the dependency model, this combination is achieved without any change in the dependency classes.

```
@FixtureSetup({
 UserTest.class,
 EventTest.class
})
class UserEventTest { ... }
```

Figure 2. Multiple dependencies in annotation @FixtureSetup.

D. Multiple Dependencies

The dependence relationship between test classes is transitive. Thus, it is possible to create a chained sequence of test classes. Each test run of a given test class will be preceded by the recursive execution of the fixture setups.

Test class sequences may be especially useful for implementation of evolutionary acceptance test specifications. Acceptance tests, also known as story tests [10], are user tests that verify if a system satisfies the acceptance criteria defined by a client [3]. In an evolutionary specification, the automated acceptance tests model use scenarios of the system where each test has as start point in a previously scenario [6].

E. Test Fixture Sharing Model between Test Classes

In the dependency model, a test class may reuse the fixture setup of another test class. The fixture setup of a given test class is the code that prepares the SUT in order to run the tests included in the test class. Test fixtures are the elements necessary to the tests and are created by the fixture setup run, such as: an object, a record in a database, a file, etc.

The nature of both test fixture and fixture setup strategy determines the way in which the test fixture is available to the test. In the inline setup, implicit setup and delegate setup strategies the ways the test fixture are available are, respectively, a local variable, a test class and a return of the helper method call.

The contributions of the work proposed in [11] were used here. That work presents a tool, called Picon, to promote code reuse in an isolated file. Each fixture setup is associated with a qualifier. The fixture setup run consists in injecting an object in a field (named with the same qualifier) of the test class. The injected object contains the test fixtures described in the fixture setup.The qualifier should be simple and, at the same time, express the fixture setup configuration [11]. Thus, the authors claim that the developer may quickly identify a given fixture setup just reading the qualifier.

Besides the dependency model, we also define a test fixture sharing model between test classes. The purpose of this model is to make the test fixtures of a given test class available to tests of a different test class. A sharing mechanism allows sharing class fields and test fixture qualifiers. So, tests of a dependent class may access a test fixture associated to a field in a dependency class through the declaration of a field named with the desired test fixture qualifier created in the dependent class. Thus, the test framework must inject the field of the dependency class in the field of the dependent class.

F. Graph Model

In the dependency model, each test class can have as many dependency classes as needed. Furthermore, the dependency relationship is transitive. Thus, a directed graph is defined to represent test classes and dependency relationships. The digraph $\mathcal{G} = (\boldsymbol{V}, \boldsymbol{E})$ is defined as follow:

$$\boldsymbol{V} = \{c \mid c \text{ is a test class}\}$$

$$\boldsymbol{E} = \{(c, d) \mid c \text{ directly depends on } d\}$$

We call $\mathcal{G}$ as Static Dependence Graph (SDG). The set $\boldsymbol{V}$ contains all test classes of the system. However, sometimes it is convenient to include only test classes involved in a given test run. Therefore, the subgraph $\mathcal{H} = (\boldsymbol{W}, \boldsymbol{F})$ is defined from the SDG as follow:

$$\boldsymbol{W} = \{c \in \boldsymbol{V} \mid c \text{ is a test class of the running test or } c \text{ is a dependency class of the running test}\}$$

$$\boldsymbol{F} = \{(c, d) \mid c \text{ directly depends on } d\}$$

We will call $\mathcal{H}$ as Dependence Execution Graph (DEG). Both SDG and DEG are digraphs. Thus, the arrow points from the dependent class to the dependency class. In the DEG of the Fig. 3, the vertex $\boldsymbol{\mathcal{TD}}$ is the test class of the running test. The $\boldsymbol{\mathcal{TA}}$, $\boldsymbol{\mathcal{TB}}$ e $\boldsymbol{\mathcal{TC}}$ vertices are the dependency classes of $\boldsymbol{\mathcal{TD}}$. A depth first search starting from $\boldsymbol{\mathcal{TD}}$ gives the Fixture Setup Execution Sequence (FSES) needed for the running test. The FSES gives the order in which the fixture setups should be run to prepare the SUT for the test. The depth first search can visit a vertex more than once. There are two valid FSES for the DEG shown in Fig. 3:

$$\mathbf{S1} = (\boldsymbol{\mathcal{TA}}, \boldsymbol{\mathcal{TB}}, \boldsymbol{\mathcal{TA}}, \boldsymbol{\mathcal{TC}}, \boldsymbol{\mathcal{TD}})$$

$$\mathbf{S2} = (\boldsymbol{\mathcal{TA}}, \boldsymbol{\mathcal{TB}}, \boldsymbol{\mathcal{TC}}, \boldsymbol{\mathcal{TD}})$$

In $\mathbf{S1}$ the approach includes each vertex $\boldsymbol{v}$ whenever $\boldsymbol{v}$ is found. So, the fixture setup run of $\boldsymbol{\mathcal{TA}}$ is included twice. In other side, in $\mathbf{S2}$ the approach includes each vertex $\boldsymbol{v}$ only the first time that $\boldsymbol{v}$ is found.

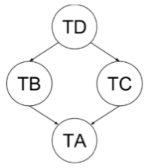


Figure 3. Dependence Execution Graph.

G. Oneness of Fixture Setup

A given fixture setup has the oneness property when it should run only once during a same test run. To repeat a fixture setup run when the oneness property is true can cause an unwanted failure in the test. The dependency model will add two new constraints: (1) the default value for the oneness property of a fixture setup is false; and (2) if the oneness property of a fixture setup is true, then it should be explicitly declared in the test class. In the framework Story, to define as true the oneness property we have to annotate the test class with the annotation @Singular.

IV. CASE STUDY

This section presents a case study where the proposal of this work was applied. Fig. 4, 5, 6 and 7 present test classes of a system for event scheduling and tracking. Many users can participate of one event and each user should inform his available schedule for the given event. Only the tests for the entities User and Event where considered in this case study. The test classes used in the case study were specifically chosen in order to exemplify the most important aspects of the proposal. Our intention is to show how the approach could be applied in a real development scenario.

The tests for the entities User and Event are included in test class UserTest (Fig. 4) and test class EventTest (Fig. 5), respectively. The test class UserEventTest (Fig. 6) contains the tests for the associative entity UserEvent.

The fixture setup UserTest is defined by the implicit setup method createAndInsertJohn(). Both tests of the test class UserTest depend on having the test fixture john registered in the database. The registration of the test fixture john is done by the UserTest fixture setup.

```
@FixtureSetup(NoDataTest.class)
class UserTest {
 @Fixture UserDao userDao;
 User john;
 Long id;

 @Before void createAndInsertJohn() {
  john = new User();
  john.setName("John");
  john.setCareer("Teacher");
  id = userDao.insert(john);
  assertNotNull(id);
 }

 @Test void get() {
  User user = userDao.getEntity(id);
  assertEquals(id,user.id());
  ... }

 @Test void list() {
  List<User> list = userDao.list();
  ...}
}
```

Figure 4. UserTest test class.

The tests of the test class UserTest have, as starting point, a scenario where the database is empty. This scenario is expected in order to have a greater control over the tests and to avoid fragile tests [12]. There are two common approaches to put the SUT in the expected starting scenario: (1) clean all database before each test run; or (2) remove the persisted entities after each test run. The second approach could be achieved with the use of the @After JUnit annotation. This annotation is analogous to @Before but the difference is that @After is run after the test method. To simplify the study case, we do not take into account the use of the @After annotation in this work. The annotation, however, could be easily applied to the dependency model. To do that, it would be only necessary to run the methods annotated with @After in the inverse order of the methods annotated with the @Before.

```
@FixtureSetup(NoDataTest.class)
class EventTest {
 @Fixture EventDao eventDao;
 Event lecture;
 Long id;

 @Before void configure() {
  lecture = new Event();
  lecture.setName("Lecture");
  id = service.insert(lecture);
  assertNotNull(id);
 }

 @Test void get() {
  Evento event = eventDao.entity(id);
  assertEquals(id,event.id());
  assertEquals(lecture.id(),event.id());
  assertEquals("Lecture",event.name());
}

 @Test void list() {
  List<Event> list = eventDao.list();
  ... }
}
```

Figure 5. UserTest test class.

The test class NoDataTest (Fig. 7) contains tests that verify the SUT when the persistence layer is empty. For this, the fixture setup of the test class NoDataTest removes all possible entities persisted in the database. It is noteworthy that the starting point expected by the test class UserTest is exactly the state of the SUT after running the fixture setup of the test class NoDataTest. Initially, two strategies are considered in order to promote code reuse: (1) move the tests of the test class UserEventTest to the test class NoDataTest; or (2) extract the fixture setup of the test class NoDataTest to a helper class and promote the code reuse through a delegate setup. The first approach has some limitations regarding the organization of the test classes. Besides this, the implicit setup method createAndInsertJohn cannot be used because it would cause an unwanted failure in the test method emptyData(). The second approach requires an effort to create auxiliary code artifacts. Furthermore, the approach can contribute with obscure tests, since the fixture setup of the test class NoDataTest should be moved to a different place from that where it was originally defined. This makes harder to understand the relationship between the test fixture and the expected behavior of the test.

Using the dependency model we are proposing, the test class UserTest can reuse the fixture setup NoDataTest without affecting the structure of the involved test classes. In the example, this is achieved using the annotation @FixtureSetup. The annotation @FixtureSetup indicates to the test framework that the run of the fixture setup of the test class NoDataTest should be done before each test run of the test class UserTest. We highlight the fact that the test fixture userDao is created in the fixture setup NoDataTest and is shared with the tests of the test class UserTest through the class fields. The class field UserTest.userDao is annotated with the annotation @Fixture and is named according to the test fixture qualifier userDao. The test fixture userDao is associated to the class field NoDataTest.userDao during the run of the fixture setup NoDataTest. After that, the test framework has to inject the class field NoDataTest.userDao into the class field UserTest.userDao.

The test class EventTest is analogous to the test class UserTest, then the same considerations about the test class UserTest are also applied to the test class EventTest.

The tests of the test class UserEventTest depend on the test fixtures john and lecture. The test fixture john is created in the fixture setup UserTest and the test fixture lecture is created in the fixture setup EventTest. Using the dependency model, the test class UserEventTest can incorporate the needed fixture setups and, thus, increase code reuse.

```
@FixtureSetup({
 UserTest.class,
 EventTest.class
})
class UserEventTest {
 @Fixture User john;
 @Fixture Event lecture;
 @Fixture UserEventDao dao;
 UserEvent johnLecture;
 Long id;

 @Before void configure() {
  johnLecture = new UserEvent();
  johnLecture.setUser(john);
  johnLecture.setEvent(lecture);
  id = dao.insert(johnLecture);
  assertNotNull(id);
 }

 @Test void get() {
  UserEvent entity = dao.entity(id);
  ...   }

 @Test void list() {
  List<UserEvent> list = dao.list();
  ... }
}
```

Figure 6. UserTest test class.

The test class NoDataTest has a particularity regarding the other test classes. Its fixture setup has the oneness property (i.e. the fixture setup run should not repeat for a same test run). The annotation @Singular is used in order to declare the oneness property of the fixture setup NoDataTest. The absence of the annotation causes an unwanted failure at the tests of the test class UserEventTest because the NoDataTest fixture setup run would be run twice: before the fixture setup UserTest and before the fixture setup EventTest. In this case, the second run of fixture setup NoDataTest would remove the test fixture john (persisted in the UserTest fixture setup run). The annotation @Singular prevents the test framework from repeating the NoDataTest fixture setup run, avoiding the test fixture john be removed.

```
@Singular
class NoDataTest {
 EventDao eventDao;
 UserDao userDao;
 EventUserDao dao;

 @Before void configure() {
  eventDao = new EventDao();
  userDao = new UserDao();
  dao = new EventUserDao();
  eventDao.removeAll();
  userDao.removeAll();
  dao.removeAll();
 }

 @Test void emptyData() {
  assertTrue(eventDao.list().isEmpty());
  assertTrue(userDao.list().isEmpty());
  assertTrue(dao.list().isEmpty());
 }  }
```

Figure 7. UserTest test class.

In the case study, we observe that the dependency model and the test fixture sharing model facilitate the development of tests in an iterative and evolutionary approach. It allows reusing fixture setups between test classes without affecting their structures.

V. Evaluation

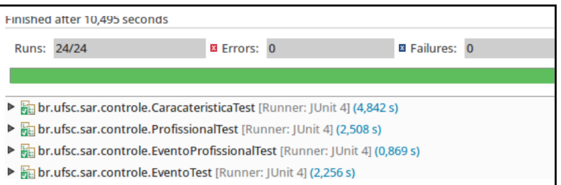
In order to evaluate the applicability of the dependency model and the test fixture sharing model we realized an experiment. The aim of the evaluation was to identify any difference between the code reuse of a set of tests using the conventional fixture setup strategies and the code reuse of a set of equivalent tests using the proposed fixture setup strategy. This experiment was realized through the development of the system presented in Section IV.

The experiment was conducted as follow. First of all, tests were developed following an iterative development. The tests should run in the framework JUnit and use the conventional test fixture setup strategies. The conclusion of the system resulted in 77 tests and 11 test classes. A subset, composed by 24 tests and 4 test classes, was selected from the total tests. This subset was named as control group. The tests selected to be included in the control group should include real database operations. Then, the control group was manually rewritten in order to use the dependency model and the test fixture sharing model of the framework Story, that implements our proposal. The set of rewritten tests, named as experimental group, was composed by 24 tests, 14 test classes and 1 helper class. The creation of the experimental group had consider the following constraints: (1) for each test of the control group should exist an equivalent test in the experimental group; (2) the test coverage should not change between the two groups; (3) the test fixture set and assertion set for each test of the control group should be the same for the equivalent test in the experimental group; (4) names of variables, methods and class fields should be preserved whenever possible; (5) Story annotations should be placed in an individual line; and (6) the tests can be freely reorganized since the previous restrictions are respected.

After the experiment, the follow measurements were collected in the control group and experimental group: (1) amount of code lines of test classes and helper classes; (2) sum of the amount of repeated lines, excluding assertions; (3) sum of distinct repetitions, excluding assertions; (4) sum of the amount of repeated lines, including assertions; and (5) sum of distinct repetitions, including assertions. In all measurements we ignored: blank lines, package declarations and import declarations. In the measurements 2, 3, 4 and 5 the following symbols were not counted as repetitions: annotation @Test, annotation @Before, annotation @Fixture, identic method declaration and block delimiter symbol.

Fig. 8 shows the results for the execution of the three distinct test groups: (a) tests of the control group; (b) tests of the experimental group; and (c) tests of both groups. As we can see in Fig. 8 (a) and in Fig. 8 (b), the test execution time between the two groups is consistently alike. This behavior is a reasonable proof that the third restriction was not violated.

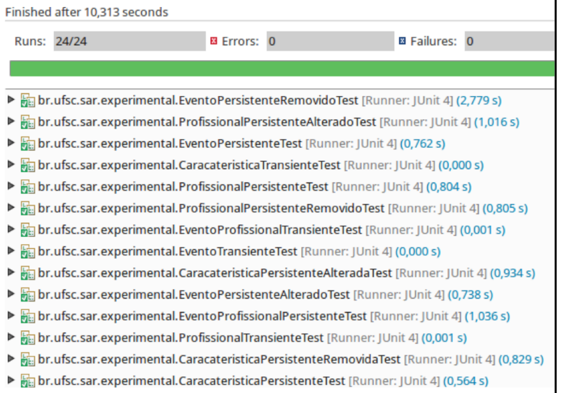
The experiment results are shown in Fig. 9. Each time reported is the result of a single execution. Black bars represent control group measurements while gray bars represent the experimental group measurements. The experimental group had an amount of test code lines slightly larger. The extra annotations of Story can justify the increase in the amount of test code lines. However, the experimental group presented a considerable reduction in the amount of repeated lines (126 for the control group and 66 for the experimental group). Considering the proportionality of test code lines, the control group had 40,91% of repeated lines while the experimental group had 20,37%. Comparing the control group with the experimental group, the last one had, considering absolute values, a reduction of 47,62% of the repeated lines.



```
Finished after 10,495 seconds
Runs: 24/24    Errors: 0    Failures: 0

▶ br.ufsc.sar.controle.CaracateristicaTest [Runner: JUnit 4] (4,842 s)
▶ br.ufsc.sar.controle.ProfissionalTest [Runner: JUnit 4] (2,508 s)
▶ br.ufsc.sar.controle.EventoProfissionalTest [Runner: JUnit 4] (0,869 s)
▶ br.ufsc.sar.controle.EventoTest [Runner: JUnit 4] (2,256 s)
```

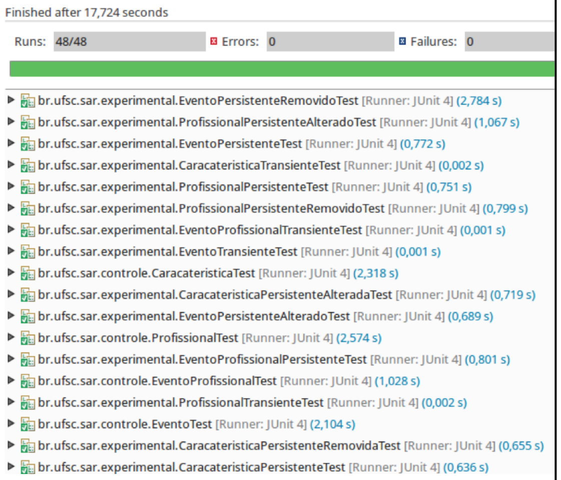
(a) Control group.



```
Finished after 10,313 seconds
Runs: 24/24    Errors: 0    Failures: 0

▶ br.ufsc.sar.experimental.EventoPersistenteRemovidoTest [Runner: JUnit 4] (2,779 s)
▶ br.ufsc.sar.experimental.ProfissionalPersistenteAlteradoTest [Runner: JUnit 4] (1,016 s)
▶ br.ufsc.sar.experimental.EventoPersistenteTest [Runner: JUnit 4] (0,762 s)
▶ br.ufsc.sar.experimental.CaracateristicaTransienteTest [Runner: JUnit 4] (0,000 s)
▶ br.ufsc.sar.experimental.ProfissionalPersistenteTest [Runner: JUnit 4] (0,804 s)
▶ br.ufsc.sar.experimental.ProfissionalPersistenteRemovidoTest [Runner: JUnit 4] (0,805 s)
▶ br.ufsc.sar.experimental.EventoProfissionalTransienteTest [Runner: JUnit 4] (0,001 s)
▶ br.ufsc.sar.experimental.EventoTransienteTest [Runner: JUnit 4] (0,000 s)
▶ br.ufsc.sar.experimental.CaracateristicaPersistenteAlteradaTest [Runner: JUnit 4] (0,934 s)
▶ br.ufsc.sar.experimental.EventoPersistenteAlteradoTest [Runner: JUnit 4] (0,738 s)
▶ br.ufsc.sar.experimental.EventoProfissionalPersistenteTest [Runner: JUnit 4] (1,036 s)
▶ br.ufsc.sar.experimental.ProfissionalTransienteTest [Runner: JUnit 4] (0,001 s)
▶ br.ufsc.sar.experimental.CaracateristicaPersistenteRemovidaTest [Runner: JUnit 4] (0,829 s)
▶ br.ufsc.sar.experimental.CaracateristicaPersistenteTest [Runner: JUnit 4] (0,564 s)
```

(b) Experimental group.



```
Finished after 17,724 seconds
Runs: 48/48    Errors: 0    Failures: 0

▶ br.ufsc.sar.experimental.EventoPersistenteRemovidoTest [Runner: JUnit 4] (2,784 s)
▶ br.ufsc.sar.experimental.ProfissionalPersistenteAlteradoTest [Runner: JUnit 4] (1,067 s)
▶ br.ufsc.sar.experimental.EventoPersistenteTest [Runner: JUnit 4] (0,772 s)
▶ br.ufsc.sar.experimental.CaracateristicaTransienteTest [Runner: JUnit 4] (0,002 s)
▶ br.ufsc.sar.experimental.ProfissionalPersistenteTest [Runner: JUnit 4] (0,751 s)
▶ br.ufsc.sar.experimental.ProfissionalPersistenteRemovidoTest [Runner: JUnit 4] (0,799 s)
▶ br.ufsc.sar.experimental.EventoProfissionalTransienteTest [Runner: JUnit 4] (0,001 s)
▶ br.ufsc.sar.experimental.EventoTransienteTest [Runner: JUnit 4] (0,001 s)
▶ br.ufsc.sar.controle.CaracateristicaTest [Runner: JUnit 4] (2,318 s)
▶ br.ufsc.sar.experimental.CaracateristicaPersistenteAlteradaTest [Runner: JUnit 4] (0,719 s)
▶ br.ufsc.sar.experimental.EventoPersistenteAlteradoTest [Runner: JUnit 4] (0,689 s)
▶ br.ufsc.sar.controle.ProfissionalTest [Runner: JUnit 4] (2,574 s)
▶ br.ufsc.sar.experimental.EventoProfissionalPersistenteTest [Runner: JUnit 4] (0,801 s)
▶ br.ufsc.sar.controle.EventoProfissionalTest [Runner: JUnit 4] (1,028 s)
▶ br.ufsc.sar.experimental.ProfissionalTransienteTest [Runner: JUnit 4] (0,002 s)
▶ br.ufsc.sar.controle.EventoTest [Runner: JUnit 4] (2,104 s)
▶ br.ufsc.sar.experimental.CaracateristicaPersistenteRemovidaTest [Runner: JUnit 4] (0,655 s)
▶ br.ufsc.sar.experimental.CaracateristicaPersistenteTest [Runner: JUnit 4] (0,636 s)
```

(c) Both control and experimental groups.

Figure 8. Execution of test groups through the Story.

VI. Conclusion

This work focused on presenting models to help increasing code reuse without negatively affecting code simplicity, expressiveness and separation of concerns.

The dependency model and test fixture sharing model presented in this work contribute with a new fixture setup strategy. Through this strategy, it is possible to promote test code reuse without losing the freedom of reorganizing test classes.

Furthermore, we presented a study case where we qualitatively evaluate how code simplicity, expressiveness and separation of concerns are well preserved. Another contribution was the definition of the oneness property applied to fixture setups.

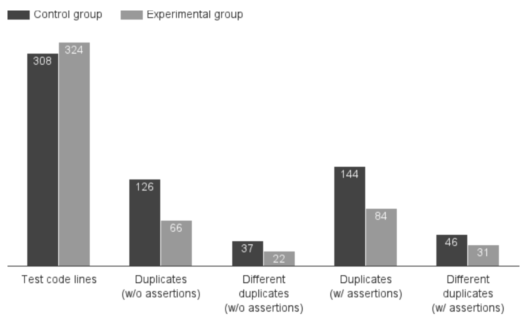


Figure 9. Experiment results.

References

[1] Beizer, B. 1990. *Software Testing Techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

[2] Berner, S., Weber, R., and Keller, R.K. 2005. *Observations and lessons learned from automated testing*. In Proceedings of the 27th International Conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 571-579.

[3] Bertolino, A. 2007. *Software Testing Research: Achievements, Challenges, Dreams*. In Procedings of the 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 85-103.

[4] Borg, R., and Kropp, M. 2011. *Automated acceptance test refactoring*. In Proceedings of the 4th Workshop on Refactoring Tools (WRT '11). ACM, New York, NY, USA, 15-21.

[5] Canfora, G., Cimitile, A., Garcia, F., Piattini, M., and Visaggio, C.A. 2006. *Evaluating advantages of test driven development: a controlled experiment with professionals*. In Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06). ACM, New York, NY, USA, 364-371.

[6] Freeman, S., and Pryce, N. 2009. *Growing Object-Oriented Software, Guided by Tests (1st ed.)*. Addison-Wesley Professional.

[7] Zaidman, A., Deursen, A., and Storey, M.A. 2013. *Strategies for avoiding text fixture smells during software evolution*. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13). IEEE Press, Piscataway, NJ, USA, 387-396.

[8] Hanssen, G.K., Haugset, B. 2009. *Automated Acceptance Testing Using Fit*. In Proceedings of the 42nd Hawaii International Conference System Sciences (HICSS '09). IEEE Computer Society, 1-8.

[9] Kamalrudin, M., Sidek, S., Aiza, M.N., and Robinson, M. 2013. *Automated Acceptance Testing Tools Evaluation*. In Agile Software Development. Sci. Int, 4, 1053-1058.

[10] Longo, D.H., Wilges, B., Vilain, P., and Cislaghi, R. 2015. *Fixture Setup through Object Notation for Implicit Test Fixtures*. Journal of Computer Science 11, 6, 794.

[11] Meszaros. G. 2006. *xUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[12] Meszaros, G., Smith, S., and Andrea, J. 2003. *The test automation manifesto*. In Proceedings of the 3rd XP Universe Conference (XP'03). New Orleans, LA.

[13] Pinto, L.S., Sinha, S., and Orso, A. 2012. *Understanding myths and realities of test-suite evolution*. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, Article 33, 11 pages.

[14] Tiwari, R., and Goel, N. 2013. *Reuse: reducing test effort*. SIGSOFT Softw. Eng. Notes 38, 2 (March 2013), 1-11.