Journal of Computer Science

Original Research Paper

Estória: A Framework for Code and Execution Reuse between Test Classes

Lucas Pereira da Silva and Patrícia Vilain

Departamento de Informática e Estatística, Programa de Pós-Graduação em Ciências da Computação, Universidade Federal de Santa Catarina, Brasil

Article history
Received: 25-11-2020
Revised: 06-02-2021
Accepted: 09-02-2021

Corresponding Author:
Lucas Pereira da Silva
Departamento de Informática e Estatística, Programa de Pós-Graduação em Ciências da Computação, Universidade Federal de Santa Catarina, Brasil
Email: lucas.pereira@ufsc.br

Abstract: Manual implementation of test code can lead to maintainability problems such as test code duplication. Strategies to avoid test code duplication and to promote test code reuse exist in the literature. However, it becomes increasingly difficult to apply these strategies as the system evolves and the number of test increases. In this study, we present a novel strategy to reuse test setup code from other test classes without breaking the independency principle of the Test Automation Manifesto. We implemented this strategy in a framework called Estória. Beyond test code reuse, the framework also enables the reuse of setup execution. Test execution reuse is important because it reduces testing runtime and allows a quicker feedback to test developers. The overall impact in promoting both code and execution reuse of test setup is an improvement in test maintainability and a reduction in testing effort. Experimental results showed a reduction of 47.62% of duplicate test lines using our strategy. In addition, execution reuse allowed to reduce the execution time by a ratio of 8.

Keywords: Software Testing, Test Code, Test Code Reuse, Test Execution Reuse, Fixture, Fixture Setup, Test Class

Introduction

Software testing is an activity that plays an important role in the software development process (Bourque *et al*., 2014). It is primarily used to provide evidence that the software works as expected (Bertolino, 2007). In addition, other benefits can also be achieved, such as detecting defects (Tiwari and Goel, 2013), preventing bug insertion (Agrawal *et al*., 1993), giving feedback about design (Freeman and Pryce, 2009) and serving as a means for software documentation (Haugset and Hanssen, 2008) and specification (Alvestad, 2007). Software testing can be automated or manual (Meszaros, 2007). Test automation is the process of creating automated tests in order to reduce the effort needed to manually test the system. We can cite three different approaches to create automated tests. The most straightforward approach consists in manually write the test code. The second approach is the Model-Based Testing (MBT), where the test code is automatically generated from the software requirements (Dalal *et al*., 1999). The third approach is the one that uses recording tools to generate the test code from the manual use of the System Under Test (SUT) (Adamoli *et al*., 2011). In this study, we focus on the first approach, in which the quality of test code has special importance. Unlike the MBT and the recording-tools approaches, the manual codification of test code demands constant maintenance by programmers (Ramler and Wolfmaier, 2006). In this scenario, the test code must be maintained during all the project and a good maintainability must be ensured in order to reduce the development effort.

The testing activity can take up 50 to 60% of the total project development effort and 30% of total project effort (Tsai *et al*., 2003; Kumar and Mishra, 2016). This demonstrates the importance of test code maintainability to reduce the total development effort. In order to achieve a good maintainability, test methods should be clearly structured, well named, small and most importantly, code duplication should be avoided (Greiler *et al*., 2013b). In general, tests tend to be repetitive with a high potential of reuse (Berner *et al*., 2005). As the system evolves, the number of tests increases, and it is easier to find code duplication across

tests. Thus, a key aspect for reducing testing effort is the development of test through a reusable-oriented approach (Tiwari and Goel, 2013). Test development effort in approaches without code reuse is lower in the beginning; however, as the number of tests increases, the effort of developing without reuse also increases (Berner *et al*., 2005). On the other hand, when one uses reusable components, the effort of developing tend to be reduced as the number of test increases (Meszaros, 2007). To achieve an effective cost-benefit trade-off, the effort to promote code reuse must be offset by the improvement in the maintainability. Thus, the approach used to promote code reuse is crucial to ensure this positive trade-off (Garousi and Küçük, 2018).

Figure 1 shows an example of code duplication between two tests. Lines 3 to 5 in the first test are the same as lines 13 to 15 in the second test. In this naive example, code duplication does not seem to be a big problem. However, as the system evolves and the number of tests increases, this type of duplication can be harmful to the test code maintainability, e.g., a refactoring in the User constructor would demand modification in both tests.

Another factor that affects testing activity is the time needed to run a given suite of tests. The first impact of slow tests is a reduction in the productivity of the person running the tests (Meszaros, 2007). Also, test suites that take too long to run will be run less frequently and test automation may fail when the automated tests are not run frequently (Berner *et al*., 2005). Because test suites have a strong tendency to be forgotten when not running for a while, this leads to the degradation of tests over time. It is worth mentioning that part of test runtime is spent with fixture setup. Hence, it is possible to reduce testing runtime by reusing fixture setup executions across tests. We will explore this subject in more detail later in the paper.

In this study, we intend to address two problems: (1) Test code duplication; and (2) slow test execution. Our approach to solve both problems is different from the well-known approaches in the literature. In our approach, we consider that each test class has its own implicit setup (Meszaros, 2007) code and that both code reuse and execution reuse can be promoted through a definition of dependency between test classes. Thus, with code reuse we can reduce code duplication across tests and with execution reuse we can speed up test execution. To promote both code and execution reuse between tests from different test classes, we proposed a dependency model between test classes. We also implemented a framework called Estória, which supports the proposed model.

The remainder of this paper is structured as follows: Section 2 presents core concepts regarding test code and execution of tests and highlights the main characteristics of approaches in the literature to solve test code duplication and slower test execution problems. In section 3, related works are presented and discussed. In section 4, our proposal is presented along with the dependency model between test classes and the Estória framework. In section 5, we evaluate our proposal and compare it with well-known approaches in the literature. Finally, section 6 shows the conclusions and future works.

```
1  public class CreateUserTest {
2      @Test public void create() {
3          User john = new User();
4          john.setName("John Doe");
5          john.setCareer("Programmer");
6          assertNull(john.getId());
7          assertEquals("John Doe", john.getName());
8          assertEquals("Programmer", john.getCareer());
9      }
10 }
11 public class InsertUserTest {
12     @Test public void insert() {
13         User john = new User();
14         john.setName("John Doe");
15         john.setCareer("Programmer");
16         UserDao dao = new UserDao();
17         Long id = dao.insert(john);
18         assertEquals(id, john.getId());
19     }
20 }
```

Fig. 1: Test classes with code duplication

Background

In this section, we present fundamental concepts for our work.

Test Code Smells

Code smells correspond to a poor solution to a recurring implementation and design problem (Fowler, 2018). Test code and production code share common code smells. However, due to its characteristics, test code can involve specific code smells (Meszaros, 2007; Greiler *et al*., 2013a; 2013b; Garousi and Küçük, 2018; Lambiase *et al*., 2020). Test code smells affects the quality of test code in a long run and can reduce test code maintainability by different factors (Garousi and Küçük, 2018). For example, one common smell is obscure test (Meszaros, 2007), which occurs when we have difficult to understand what behavior a test is verifying. It can be caused by either too much information or too little information. Another important test smells are code duplication and slow test execution (Meszaros, 2007).

Four-Phase Test Pattern

The four-phase test pattern separates test execution into four phases: (1) Setup, (2) exercise, (3) verification and (4) tear down (Meszaros, 2007). In the setup phase, test fixtures are created, i.e., everything needed to exercise the SUT is put in place. In the exercise phase, the test interacts with the SUT by calling the operation or the feature to be tested. Next, in the verification phase, the test must verify if the expected outcome corresponds to the obtained behavior. Finally, in the last phase, the state of the SUT is put back into the initial state. Some projects do not use the last phase of this pattern because, in this case, for each test run, the SUT is prepared from scratch in the setup phase. The main advantage of this approach is to avoid fragile tests (Meszaros, 2007) that fail due to a previous test run that let the SUT in an inconsistent state.

Fixture Setup Strategies

As stated before, fixtures correspond to everything needed to exercise the SUT. A fixture can be an object in memory, a record in a database and even a file in a filesystem. The code responsible for creating fixtures is called fixture setup. There are distinct fixture setup strategies and each one has its pros and cons. The most straightforward fixture setup strategy is inline setup (Meszaros, 2007). In inline setup, the code needed to setup fixtures is put directly in the test method. The tests shown in Fig. 1 use this strategy, which produces code duplication smell (Meszaros, 2007). At a first glance, the inline setup can establish a good relationship between fixtures and the expected outcome. However, large amount of setup code can lead to irrelevant information smell (Meszaros, 2007), a subdivision of the obscure test smell, making tests hard to understand.

Another fixture setup strategy is implicit setup (Meszaros, 2007). It can be used to reduce code duplication from inline setup. In the implicit setup, tests are placed in a same test class and a special method, usually called setup method, is created. The setup code common to all tests of the class is placed in the setup method. The setup method is run before each test method of the class. Although this strategy promotes code reuse, it does not promote execution reuse. Moreover, as the implicit setup moves some fixture setup code to a method that is not the test method, the establishment of relationship between fixtures and expected outcomes is reduced when compared to inline setup. Also, as the implicit setup includes the tests in the same test class, it may not scale well because to keep increasing reuse it would be necessary to increase the number of tests in the same test class. Grouping tests in the same class in order to promote code reuse through implicit setup can lead to several test code smells, such as general fixture, test maverick and lack of cohesion of test methods (Greiler *et al*., 2013b).

In the delegated setup strategy (Meszaros, 2007), setup code is placed in helper methods that can be called by test methods from different test classes. This can be used to promote code reuse. However, delegated setup hides details of test fixtures. This is a double-edged sword characteristic, because it can simplify and improve the readability of test code by avoiding irrelevant information smell, but at the same time it can also lead to mystery guest smell (Meszaros, 2007), a subdivision of the obscure test smell that makes harder to understand the relationship between fixtures and expected outcomes. Unlike implicit setup, delegated setup does not require to include the setup code in the same test class of the test methods. However, the main disadvantage is that the delegated setup method returns only one fixture object to the test method that called it. In order to enable access to several fixtures objects and avoid the hard-coded test data smell (Meszaros, 2007), the delegated setup method must be broken into multiple methods and this can be a time-consuming and error-prone task.

Implicit setup and delegated setup only promote code reuse. However, there is a category of strategies, called shared fixture construction, which also promotes execution reuse in addition to code reuse (Meszaros, 2007). In shared fixture construction, common fixtures are persisted between different test executions. The fixtures are created once and can be reused several times by different tests that depend on them. In this way, shared fixture construction decreases the time needed to run tests. However, it has two major problems: (1) A

given test may dirty a fixture and leave it in an inconsistent state for the next test run; and (2) test frameworks typically do not have built-in mechanisms to manage shared fixtures and it is up to the programmer to ensure the consistency of the life cycle of tests and fixtures. The second problem breaks the Hollywood principle (Sweet, 1985; Sobernig and Zdun, 2010) which states that the framework must call the application and not the contrary. To break the Hollywood principle is dangerous because it may affect the internal state of the framework, interfering in the correct execution of the tests (Mattsson *et al*., 1999). Moreover, this can increase the effort needed to write tests, because developers need to ensure that the used shared fixture setup strategy do not cause an unwanted failure in the tests.

Each fixture setup strategy has its own advantages and disadvantages. In fact, they are complementary regarding the characteristics of code reuse, execution reuse, readability and complexity of implementation. In general, the weak point of one strategy is the strong point of the others. The most experienced programmers can balance the use of the most adequate strategy for each situation. However, as the number of tests increases it is more difficult, even for experienced programmers, to find the right balance between the strategies.

Related Works

In this section, we present some works that propose new fixture setup strategies for specialized contexts. We investigate works that also focus on the improvement of test code reuse, test execution reuse, or both.

DbUnit

The DbUnit[1] framework (Christensen *et al*., 2006) is an extension of JUnit[2] to reuse fixture of tests involving database applications. In this framework, a test class must be defined for each database entity to be tested. The main goal is to reuse the fixtures inserted by a given test class in another test class. To achieve this, each test class must define three methods: (1) An insertion method where the entity represented by the test class is inserted; (2) a deletion method where the entity represented by the test class is removed; and (3) a method that returns the list of entities that the entity being tested depends on, called dependency list. Based on the dependency list returned by the third method, the framework executes, in a recursive order, the insertion test method of each dependency entity. After that, the framework executes, in a recursively inverse order, the deletion test method of each dependency entity.

For example, suppose that there is an entity called teacher and there is an entity called office. Each teacher must have an office. So, in order to test the insertion of a Teacher entity, first it is necessary to insert an Office entity. To accomplish this and to reuse the execution, the DbUnit framework runs the tests relative to these two entities in a specific order. First, the framework runs the test to insert the Office entity. Next, the test to insert the Teacher entity is run. It is worth mentioning that at this point, the framework reuses the entity inserted in the first test. In this way, it is possible to achieve execution reuse because to run the Teacher test it is not necessary to insert an Office entity again. After the recursive running of the insertion tests, the framework runs, in the recursively inverse order, the deletion tests. This is necessary in order to clean the SUT and let it in a consistent state. Thus, first the framework will run the deletion of the Teacher test and, finally, the deletion of the Office test.

The main limitation of this work is that the fixtures created are only accessible through database operations. Fixtures created by test classes of dependency entities cannot be accessed programmatically by the running test. Also, although the framework promotes execution reuse, the running test must not change the state of any dependency entity, because doing so can break other tests that depend on the same dependency entity. Thus, the work breaks the independency principle (Meszaros *et al*., 2003), which advocates that tests must be independent of each other and that a suite of tests should be run in any arbitrary order.

Reusable Fit Specifications

Mugridge and Cunningham (2005) propose the reuse of fixtures in the context of Fit[3]. Fit is an acceptance testing framework in which tests are defined using a tabular format specification (Borg and Kropp, 2011). The work proposes that Fit tests can be connected in a way that one test can serve as a starting point of another one, promoting the reuse of the Fit specifications. Next, from the connected specifications, the authors propose to build a directed graph of tests. Thus, instead of executing each test individually, the authors propose to execute the tests through a graph walking algorithm. The main drawback of the graph approach is the mischaracterization of test as a concise and cohesive unit. This approach leads to comprehension and traceability problems, e.g., it is more difficult to find the source of an error when all tests are run together as a whole.

In our approach the idea is that the setup code of one test class can be used as starting point for another test class. Mugridge and Cunningham (2005) approach differs from our approach because the specification reuse is applied directly between tests. Besides that, when considering the reuse of execution and not only the reuse

[1] http://dbunit.org/
[2] https://junit.org/
[3] http://fit.c2.com/

of setup code or specification, in our work we only reuse execution of test setup that does not change the test fixtures of its test classes.

Picon

The Picon[4] framework (Longo *et al*., 2015) is an extension of JUnit that promotes the reuse of test code of Plain Old Java Objects (POJO). POJO are objects that include a default constructor and getter and setter methods for each attribute. In Picon, the fixtures are defined in an external file through a special notation. The defined fixtures are reused through the declaration of an attribute in the test class. The attribute must have the same name as the desired fixture. For each test, Picon parses the files that contain the fixture definitions, creates the fixture objects and, finally, injects the objects into the running test.

The authors claim that the success of their approach is based on an adequate naming strategy, i.e., programmers should choose clear and meaningful names for the fixtures, otherwise test comprehension will be harmed. In a Test-Driven Development (TDD) project where Picon was used, the authors estimated a reduction of 60% of the test code.

SolUnit

SolUnit[5] (Medeiros *et al*., 2019) is a framework for reducing testing runtime of unit tests for smart contracts. Smart contracts are software programs that are run over a blockchain (Tonelli *et al*., 2018). A blockchain consists of chain of data packages which comprises multiple transactions (Nofer *et al*., 2017). Smart contracts are decentralized and immutable, i.e., once deployed they cannot be changed (Destefanis *et al*., 2018).

SolUnit aims to reduce the time spent to run tests for smart contracts. Because smart contracts need to be deployed on a blockchain to be tested, the time spent to run tests can lead to slow test execution smell. Thus, SolUnit tries to reduce test execution time by reusing smart contract deployment and test setup. To do this, first SolUnit identifies smart contract functions that do not generate new transactions in the blockchain. Tests that only call these functions are identified as well. Next, the framework reuses the contract deployment and test setup execution of the identified tests. The approach was able to reduce test execution time by 30 to 70% considering five open-source projects found on GitHub.

Discussion

As we already mentioned, in our work we focus on promoting both code and execution reuse of test code. In DbUnit and Reusable Fit Specifications, the test code and execution can be reused. However, both approaches break the independency principle. Furthermore, the approach used by DbUnit can only be applied for applications in which the fixtures are persisted in a database. Also, in DbUnit the fixtures cannot be passed between different test classes. The work proposed by Mugridge and Cunningham (2005) differs from our work too. While our work focuses on test code refactoring, Mugridge and Cunningham (2005) propose a strategy for refactoring Fit table specifications. Another difference is that our approach is applied to reuse code between test classes, while Mugridge and Cunningham (2005) propose the reuse of specifications in a test granularity. Picon promotes code reuse of fixtures defined externally to the test classes, but only POJOs can be reused and execution reuse is not provided. SolUnit is designed to reduce tests runtime of smart contracts without breaking the independency principle. The framework automatically identifies test setups that can be reused by other tests. However, SolUnit does not aim to reduce test code duplication.

Next, we present our proposal. It differs from other approaches because it can promote both code and execution reuse while preserving the independency principle and enabling programmatic access to the fixtures.

Estória

In this section we present the Estória[6] framework (da Silva and Vilain, 2016; 2017), an extension of JUnit in which we implemented our proposal. Our main goal is to enable the reuse of implicit setups between different test classes. This idea comes from the basic principle that the implicit setup of one test class can serve as starting point to the implicit setup of another test class. Thus, we define a dependency model between test classes as follows:

- **Definition 1** Given a provider test class and a consumer test class, the dependency relationship between them implies that the implicit setup of the provider will be run before the implicit setup of the consumer

This means that in order to run a given test of the consumer class, Estória will run the implicit setup of the provider before running the implicit setup of the consumer. Estória implements this behavior through the @FixtureSetup annotation. In addition to this annotation, Estória provides a complementary annotation called @Fixture. The @Fixture annotation allows the tests of a consumer class to use the fixtures created in the implicit setup of a provider class.

[4] https://github.com/douglashiura/picon
[5] https://github.com/hmhallan/sol-unit
[6] https://github.com/lucasPereira/estoria

DOI: 10.3844/jcssp.2021.90.111

```
1  public class BankingSystemTest {
2      private BankingSystem bs;
3      @Before public void setup() {
4          bs = new BankingSystem();
5      }
6      @Test public void test() {
7          assertEquals(0, bs.getBanks().size());
8          assertEquals(0, bs.getMints().size())
9      }
10 }
```

Fig. 2: Test class to create a banking system

```
1  @FixtureSetup(BankingSystemTest.class)
2  public class BankTest {
3     @Fixture private BankingSystem bs;
4     private Bank hsbc;
5     @Before public void setup() {
6         hsbc = bs.createBank("HSBC", Currency.GBP);
7     }
8     @Test public void test() {
9         assertEquals("HSBC", hsbc.getName());
10        assertEquals(Currency.GBP, hsbc.getCurrency());
11        assertEquals(0, hsbc.getAccounts().size());
12        assertEquals(1, bs.getBanks().size());
13    }
14 }
```

Fig. 3: Test class to create a bank

Figure 2 and 3 show an example of how to use Estória annotations. In Fig. 3, line 1 indicates that the implicit setup of the test class defined in Fig. 2 will be used. In JUnit, the implicit setup of a test class corresponds to the methods annotated with @Before. Estória is an extension of JUnit. Thus, in Estória, the @Before annotation has the same meaning as in JUnit. Besides that, line 3 in Fig. 3 indicates that the annotated attribute will be dynamically injected by Estória. Thus, in order to run the test in the BankTest class (Fig. 3), Estória executes the following steps: (1) Run the implicit setup of the BankingSystemTest; (2) run the implicit setup of the BankTest; (3) injects the fixtures from the provider class in the annotated attributes of the consumer class; and, finally, (4) run the test. The Estória annotations allow us to achieve code reuse and programmatic access to the fixtures.

Fixture Injection

As shown above, to use a fixture created in the implicit setup of a provider class it is necessary to declare, in a consumer class, an attribute with the same name as the desired fixture presented in the provider class. It is also necessary to annotate the declared attribute with the @Fixture annotation. Although Estória does not need the @Fixture annotation to identify the attributes to be injected, we decided to make this step mandatory because of two reasons: (1) To improve test readability by making clear which fixtures are provided by dependencies; and (2) to minimize the error-prone characteristic of the fixture injection by (a) avoiding fixture injection without the programmer's knowledge and (b) detecting typos in cases where a fixture is not found in the provider classes.

Independence Principle

It is important to highlight that the dependency model does not break the independency principle presented in the Automation Manifesto (Meszaros *et al*., 2003). The independency principle says that tests can be run in any arbitrary order without collateral effects. This principle is justified by the idea that one test execution should not depend on or interfere in a different test execution, e.g., a given test τ should not fail just because the execution of another test put the SUT in an inconsistent state for the test τ. There are two strategies to avoid side effects from previous test executions: (1) In the setup phase, each test should reset the state of the SUT; or (2) in the tear-down phase, each test should undo the modifications made in the SUT. From the perspective of each test, the first strategy is the safest because the test itself guarantees that the SUT will be in a consistent state.

In Estória, the dependency relationship between two classes does not interfere in the execution of their tests.

Instead of that, the relationship between test classes only implies that the implicit setup of the provider class will be run before the implicit setup of the consumer class. This relationship, however, does not interfere in the execution of the tests, e.g., a given test from a provider class could be run after a test from a consumer class, or vice-versa, without any side effect. In other words, in Estória, the dependency relationship between test classes does not imply on a dependency between the tests from the involved classes.

The dependency model and its implications can be better understood through a concept from biology: Commensalism (Beneden, 1876). Commensalism is a relationship between two organisms in which one organism gains benefits from another without benefiting or harming it. It is a commensal behavior that we expect with the dependency relationship between two test classes. More than that, we expect that producer classes do not even need to know the existence of consumer classes.

Transitive Dependencies

In Estória, the dependency between different test classes is a transitive relationship, e.g., if a test class Λ depends on Θ and Θ depends on Γ, then Λ also depends on Γ. Before running a given test, Estória will run, in a recursive inverse order, the implicit setups of the dependencies. Figure 4 illustrates an example of transitive dependencies. The test class presented in Fig. 4 depends on the test class presented in Fig. 3, which in turn, depends on the test class presented in Fig. 2. We say that the test class of Fig. 2 is a transitive dependency of the test class of Fig. 4. Thus, in order to run the test method presented in Fig. 4, Estória will create an Execution Sequence of Implicit Setup and Test Methods (ESISTM) to determine the order to run the methods needed for the test. Figure 5 shows an Unified Modeling Language (UML) activity diagram corresponding to the ESISTM for the test of Fig. 4.

Regarding the created fixtures, after each implicit setup execution, Estória injects the created fixtures in the next dependency implicit setup execution. It is worth mentioning that the fixture injection is also transitive, e.g., the test class in Fig. 4 uses the bs fixture created in the class of Fig. 2. However, the transitivity of fixture injection only works if the provider test class immediately following uses the fixture as well, e.g., if the test class in Fig. 3 does not use the bs fixture, then the test class in Fig. 4 could not use it either.

```
1  @FixtureSetup(BankTest.class)
2  public class AccountTest {
3      @Fixture private BankingSystem bs;
4      @Fixture private Bank hsbc;
5      private Account jane;
6      @Before public void setup() {
7          jane = hsbc.createAccount("Jane Doe");
8      }
9      @Test public void test() {
10         assertEquals("Jane Doe", jane.getName());
11         assertEquals(Money.ZERO, jane.getBalance());
12         assertEquals(1, hsbc.getAccounts().size());
13         assertEquals(1, bs.getBanks().size());
14     }
15 }
```

Fig. 4: Test class to create a account

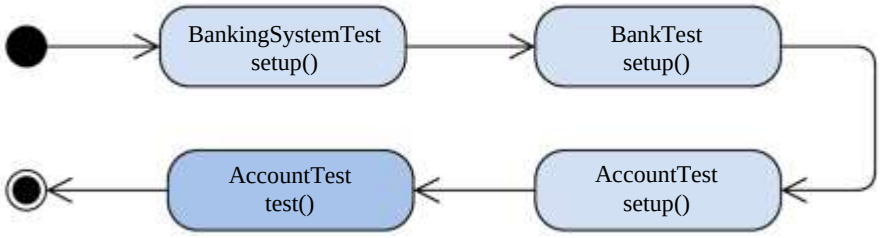

Fig. 5: ESISTM for account test

DOI: 10.3844/jcssp.2021.90.111

Transitive dependencies are especially useful for implementing story tests. Story tests, also known as acceptance tests (Kamalrudin *et al*., 2013), are user tests that are used to determine if a system satisfies the acceptance criteria from the customer perspective (Borg and Kropp, 2011). This type of test is usually implemented in an evolutionary and incremental way (Erdogmus *et al*., 2005). Transitive dependencies make possible implementing tests in the same evolutionary way, i.e., the setup of the test classes that are already implemented can serve as starting point for implementing the next test. The simple observation that story tests can be connected in order to build a bigger picture was the main idea that motivated our work.

Symmetric Dependency and Cyclic Dependency

In graph theory, a symmetric graph is a graph in which any two adjacent vertices v and u are connected by both the edges (v, u) and (u, v) and a cyclic graph is a graph with a closed chain of edges in which the terminal vertex is not distinguished from the initial vertex (Essam and Fisher, 1970). We borrowed these definitions from graph theory and applied it to our work. Thus, in the dependency model, a symmetric dependency occurs when two test classes depend directly on each other and a cyclic dependency occurs when a test class depends on itself through a chain of transitive dependencies. However, we treat symmetric and cyclic dependencies as a modeling error. Estória deals with this type of error simply by throwing a runtime exception.

Multiple Dependencies

The Estória framework also enables multiple dependencies for consumer classes, i.e., a given consumer class may depend on one or more provider classes. This scenario is illustrated through Fig. 4, 6 and 7. In Fig. 4, an account is created in the implicit setup. In Fig. 6, a mint facility to issue money for a given currency is created in the implicit setup. In Fig. 7, the class tests the behavior of an account deposit. In order to test this behavior, the test class in Fig. 7 depends on the implicit setup of two other test classes: The one presented in Fig. 4, in which the account is created and the one shown in Fig. 6, in which a mint is created.

```
1  @FixtureSetup(BankingSystemTest.class)
2  public class MintTest {
3      @Fixture private BankingSystem bs;
4      private Mint royal;
5      @Before public void setup() {
6          royal = bs.createMint("Royal Mint", Currency.GBP);
7      }
8      @Test public void test() {
9          assertEquals("Royal Mint", royal.getName());
10         assertTrue(royal.manufactures(Currency.GBP));
11         assertFalse(royal.manufactures(Currency.USD));
12         assertEquals(1, bs.getMints().size());
13     }
14 }
```

Fig. 6: Test class to create a mint

```
1  @FixtureSetup(AccountTest.class, MintTest.class)
2  public class DepositTest {
3      @Fixture private Account jane;
4      @Fixture private Mint royal;
5      private Money tenPounds;
6      @Before public void setup() {
7          tenPounds = royal.issue(10);
8          jane.deposit(tenPounds);
9      }
10     @Test public void test() {
11         assertEquals(tenPounds, jane.getBalance());
12     }
13 }
```

Fig. 7: Test class to make a deposit

The order in which the provider classes are declared in the @FixtureSetup annotation is a relevant aspect. Firstly, this order determines the execution order of the provider implicit setups, and changes in this order can affect the state of the SUT. Secondly, this order will be used to resolve fixture naming conflicts, i.e., if two provider classes have a fixture with the same name, then Estória will inject the fixture from the first class that appears in the @FixtureSetup annotation. We recommend that programmers avoid this type of conflict because it can harm test comprehension.

Multiple Tests and Multiple Implicit Setups

The test classes of the examples presented from Fig. 2 to 7 contain only one test and one implicit setup method. However, as occurs in JUnit, Estória also enables the existence of multiple tests and multiple implicit setup methods in the same test class. The challenge is to provide these features while providing code reuse between test classes and preserving the independence principle at the same time. To achieve this, Estória executes all the implicit setups of the providers and all the implicit setups of the running test class before each test run.

Figure 8 shows an example of a test class that includes two implicit setups and two tests. To run both tests of the class, Estória will run the following steps for each test: (1) Recursive execution of the setup of the DepositTest provider; (2) execution of the fifteen implicit setup method; (3) execution of the five implicit setup method; and (4) execution of the given test. The order in which the implicit setup methods of the running class are executed is determined by the alphabetical order of the method names. This is the reason why the method fifteen is executed before the method five. We choose to use the alphabetical order because JUnit also uses the same arbitrary criteria.

Graph Models

In order to facilitate the comprehension and to represent the dependency model through a more formal definition, we define the dependency model as a graph model. The directed graph $G = (V, E)$ is defined as follows:

- **Definition 2** $V = \{t \mid t$ is a test class of the system$\}$
- **Definition 3** $E = \{(c, p) \mid c$ is a consumer test class $\wedge$ p is a provider test class$\}$

We call G a Dependency Graph (DG). The V set contains the test classes of the system and the E set contains all the direct dependencies between pairs of test classes. The DG facilitates the validation of the dependency model, e.g., through a breadth-first search algorithm it is possible to detect cyclic dependencies, a type of violation in the dependency model. In addition to the static dependencies between test classes, it is convenient to represent the test classes that are involved in a given test run in order to generate the execution order of setup methods. Thus, from DG, we define a directed subgraph $H = (W, F)$ as follows:

- **Definition 4** $W = \{r \in V \mid r$ is a running class $\vee$ a running class depends on $r\}$
- **Definition 5** $F = \{(c, p) \mid$ c is a consumer test class $\wedge$ p is a provider test class$\}$

We call H an Execution Graph (EG). Unlike DG, EG does not include all the test classes of the system. While DG is a static representation of the dependency model, EG presents a subset of DG containing only the test classes involved, directly or indirectly, with the tests to be run. EG is important to determine the execution order of the implicit setups before the test run. From EG, it is possible to determine the execution order simply through a depth-first search algorithm.

Figure 9 shows the DG (left) of the test classes presented in the examples from Fig. 2 to 8 and the EG (right) considering the execution of the deposit test shown in Fig. 7. The graphs are represented by UML class diagrams. Two extra stereotypes are being used. The <<consumes>> stereotype indicates that a consumer class consumes the implicit setup of a provider class and the <<running>> stereotype indicates the tests to be run.

Singular Execution

Considering the EG shown in Fig. 9 which represents the execution of the deposit test shown in Fig. 7, it is possible to use a depth-first search algorithm to find the ESISTM. However, due to the possibility of multiple dependencies, there is a concern about the strategy to be used when a vertex is visited twice through the depth-first search algorithm. There are two possibilities: (1) The strategy may visit the already-visited vertex again; (2) the strategy may ignore the already-visited vertex. The ESISTM will vary depending on the chosen strategy.

The test class shown in Fig. 7 has two providers. Directly or indirectly, each one of these providers depends on the same test class shown in Fig. 2, the BankSystemTest. Thus, to run the test shown in Fig. 2, two distinct ESISTM may be considered as shown in Fig. 10 and 11. According to the ESISTM shown in Fig. 10, the implicit setup method of the BankSystemTest is run only once. In Fig. 11, the same implicit setup method is run twice, i.e., the BankSystemTest implicit setup method will be run before each direct consumer. In the example of the test of Fig. 7, we do not want to run BankSystemTest implicit setup method twice, otherwise we will create another bs and this is not the goal. The idea is that both consumers use the same bs. Thus, for the given example, we want to ensure that Estória runs

DOI: 10.3844/jcssp.2021.90.111

the BankSystemTest implicit setup method only once. To do this, we introduce the @Singular annotation.

```
1  @FixtureSetup(DepositTest.class)
2  public class WithdrawTest {
3    @Fixture private Account jane;
4    @Fixture private Mint royal;
5    @Fixture private Money tenPounds;
6    private Money fivePounds;
7    private Money fifteenPounds;
8    @Before public void five() {
9      fivePounds = royal.issue(5);
10   }
11   @Before public void fifteen() {
12     fifteenPounds = royal.issue(15);
13   }
14   @Test public void lessThanBalance() {
15     Transaction withdraw = jane.withdraw(fivePounds);
16     assertTrue(withdraw.hasSuccess());
17     assertEquals(fivePounds, jane.getBalance());
18   }
19   @Test public void moreThanBalance() {
20     Transaction withdraw = jane.withdraw(fifteenPounds);
21     assertFalse(withdraw.hasSuccess());
22     assertEquals(tenPounds, jane.getBalance());
23   }
24 }
```

Fig. 8: Test class to withdraw money

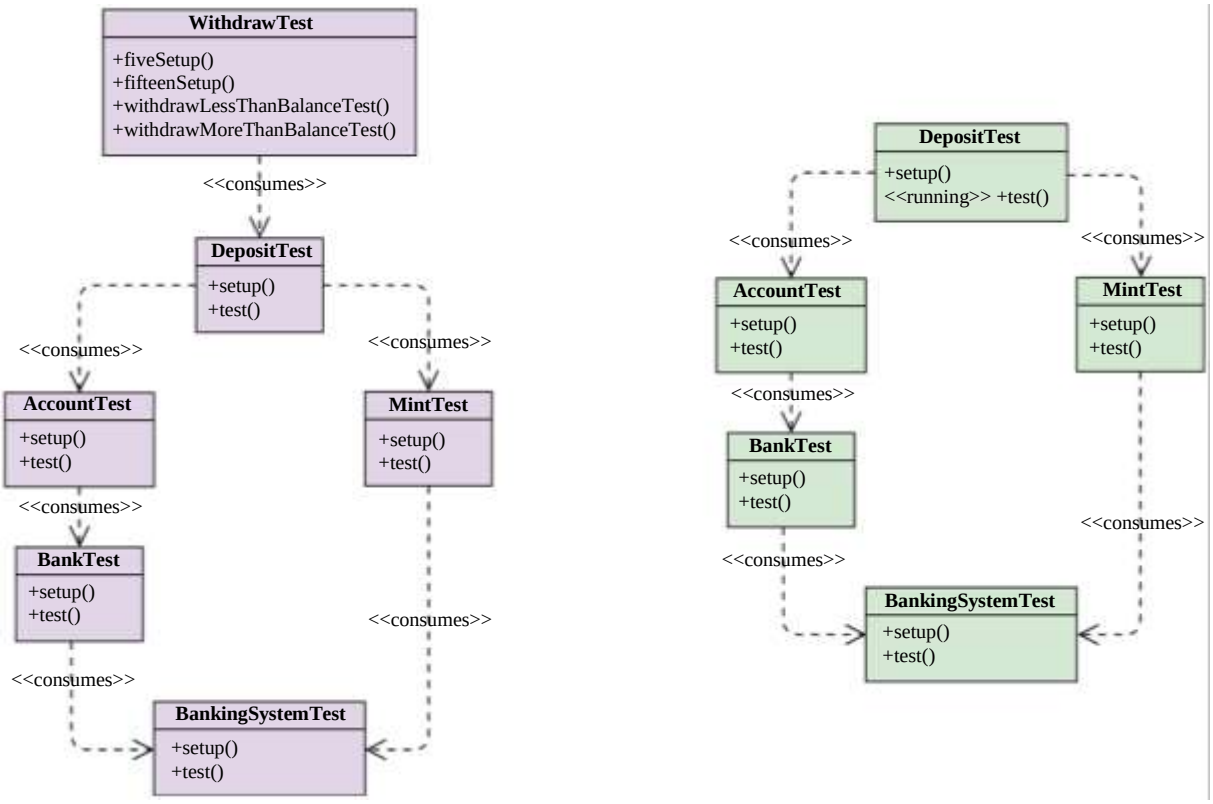

Fig. 9: DG for tests classes and EG for deposit test

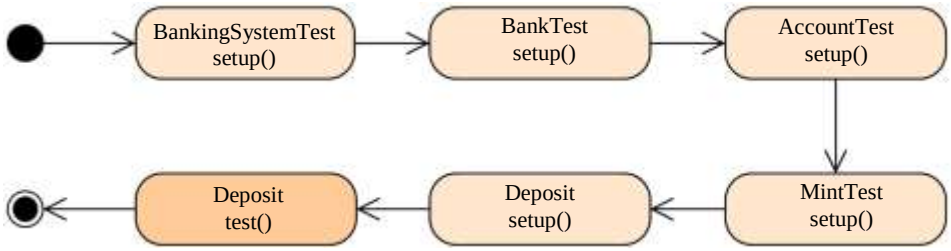

Fig. 10: ESISTM for deposit test with singular provider

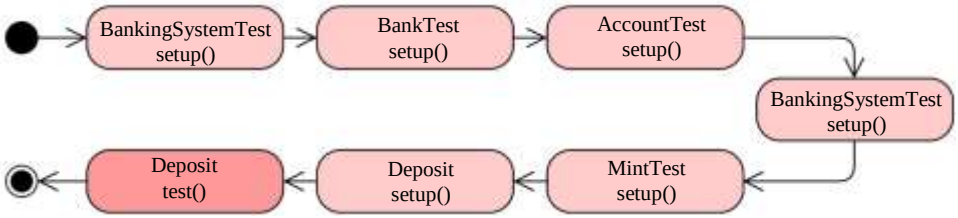

Fig. 11: ESISTM for deposit test with non-singular provider

```
1 @Singular
2 public class BankingSystemTest { ... }
```

Fig. 12: Singular annotation in the test class to create banking system

The @Singular annotation tells Estória that the implicit setup of the annotated test class should not be run before each direct consumer. Instead, the implicit setup method should be run only once, right before the first direct consumer. Thus, to assure that the implicit setup of BankSystemTest is run only once, it is necessary to annotate the test class of Fig. 2 with the @Singular annotation as shown in line 1 of Fig. 12. Estória assumes that implicit setup of classes not annotated with @Singular should be run before each direct consumer. If each direct consumer needs its own bs, then @Singular should not be used.

Execution Reuse

Until now we have shown only aspects involving code reuse. However, Estória has two reuse modes: (1) The one with only code reuse; and (2) the other one with both code and execution reuse. In the mode with only code reuse, the entire chain of implicit setup methods is run before each test. In this mode, the independency principle is never broken, because Estória does not interfere in the ordering of the tests to be run, i.e., the tests can be run in any arbitrary order. From the perspective of the mode with execution reuse, an execution of a chain of implicit setup methods of a given test may be reused for another test. In this context, the independency principle is broken, because Estória must define the ordering of the tests to be run. This section gives an overview of the execution reuse mode of Estória. A more detailed discussion about this subject can be found in previous works (da Silva and Vilain, 2016; 2017).

The first observation necessary to understand the execution reuse of Estória is to distinguish between safe and unsafe tests. We refer to a test as safe when it does not dirty the fixtures that were created in the implicit setup method of the test class nor the fixtures that were created in the implicit setup methods from the dependencies. In this way, another test could use the same fixtures without needing to run the implicit setup methods again. The examples from Fig. 2 to 7 contain only safe tests, because none of these test methods affects the fixtures created in the chain of implicit setups. A counter example is the lessThanBalance test shown in Fig. 8. In line 15, money is withdrawn from an account, changing the balance. After executing this test, it is not possible to reuse the execution of the implicit setup methods by any other test, because a fixture created in the implicit setup, the jane fixture, was affected by this test.

Estória introduces the @Safe and @Unsafe annotations. The annotations are used to indicate safe and unsafe test methods, respectively. Estória assumes that all test methods are unsafe by default. The framework does not reuse the execution of implicit setups after running an unsafe test. To reuse the execution of implicit setups, all tests, except by the last one, must be annotated with the @Safe annotation. Figure 13 shows the test class of Fig. 8 with the @Safe and @Unsafe annotations.

To exemplify the execution reuse mode of Estória, we will consider the tests from Fig. 2 to 8. Figure 14 shows the EG considering the execution of all tests. The tests to be run are marked with the <<running>> stereotype. We also highlight the use of the <<safe>> and <<unsafe>> stereotypes in test methods.

DOI: 10.3844/jcssp.2021.90.111

```
1 @FixtureSetup(DepositTest.class)
2 public class WithdrawTest {
3       @Usafe @Test public void lessThanBalance() { ... }
4       @Safe @Test public void moreThanBalance() { ... }
5 }
```

Fig. 13: Safe and Unsafe annotations in test class to withdraw money

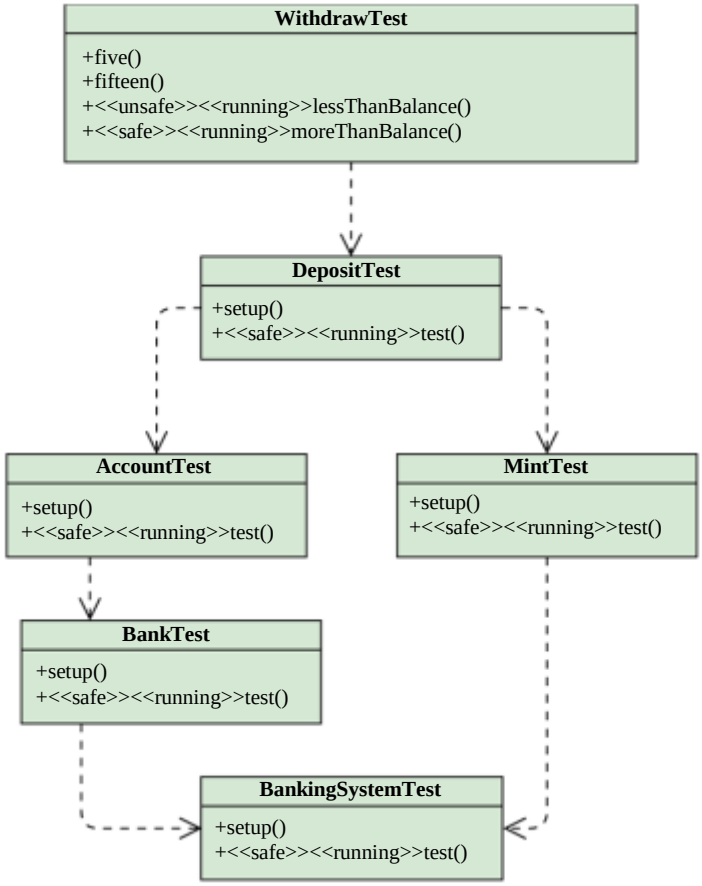

Fig. 14: EG with multiple tests to run

To promote execution reuse, Estória first selects all source vertices in the EG. A source vertex is a vertex without descendants, e.g., in Fig. 14 the WithdrawTest is the only source vertex. For each source vertex, Estória creates the transitive closure of the vertex. A transitive closure is the set of vertices that can be reached from a given vertex, e.g., in Fig. 14 all vertices belong to the transitive closure of the WithdrawTest. The execution reuse in Estória is only possible between tests from the same transitive closure. Estória does not promote execution reuse between tests from distinct transitive closures because they do not share the same dependencies. Estória also does not promote code reuse after executing an unsafe test because unsafe tests change the internal state of the SUT, and this may affect the execution of the next test as well.

To run all the selected tests and to maximize the execution reuse of the implicit setup methods, Estória starts by the biggest transitive closure and creates, from it, the ESISTM according to the reverse topological ordering. Next, the framework starts the execution of the methods according to the ESISTM. This process continues until all implicit setup methods and safe test methods are executed. Unsafe tests are scheduled to be run later, i.e., after the execution of all safe test methods. For each unsafe test, all chain of implicit setup methods must be run again, except for the first unsafe test of a source vertex. As the safe test methods do not change the fixtures of the implicit setup methods, then the first unsafe test can reuse the previous execution.

Figure 15 shows the execution of the EG shown in Fig. 14 considering the reuse mode enabled. Each test

DOI: 10.3844/jcssp.2021.90.111

class and its implicit setup methods serve as starting point for the next test class in the dependency chain. Hence, it is possible to promote the execution reuse of the implicit setup methods. It is worth mentioning that even the lessThanBalance method, an unsafe test, can reuse the execution of the previous implicit setup methods. This is possible because Estória schedules this method as the first one to be run after the last safe method. Thus, the test can reuse the previous implicit setup methods because the previous safe tests do not change the internal state of the SUT. In Fig. 16, the same scenario is shown but without the execution mode enabled. It is important to note that before each test, all implicit setup methods in the dependency chain are executed. While in Fig. 15 the execution of all tests involves 7 implicit setup method executions, in Fig. 16 this number increases to 27.

Estória Algorithms

In this section, we describe the algorithms used by Estória. Algorithm 1 and Algorithm 2 generate ESISTMs for the running tests and Algorithm 3 shows the general algorithm of Estória to run the tests based on the generated ESISTMs.

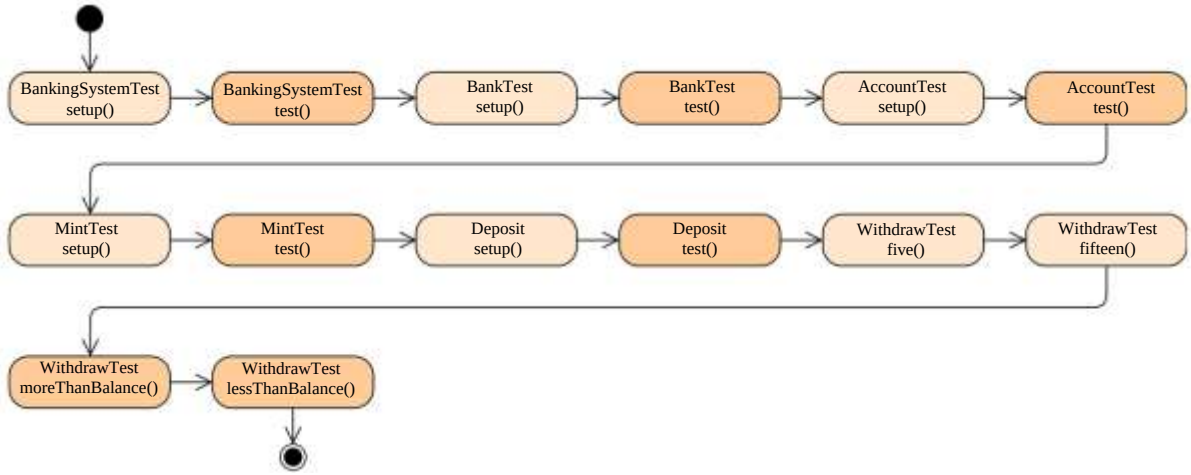

Fig. 15: Flow of execution for all tests with the execution reuse mode

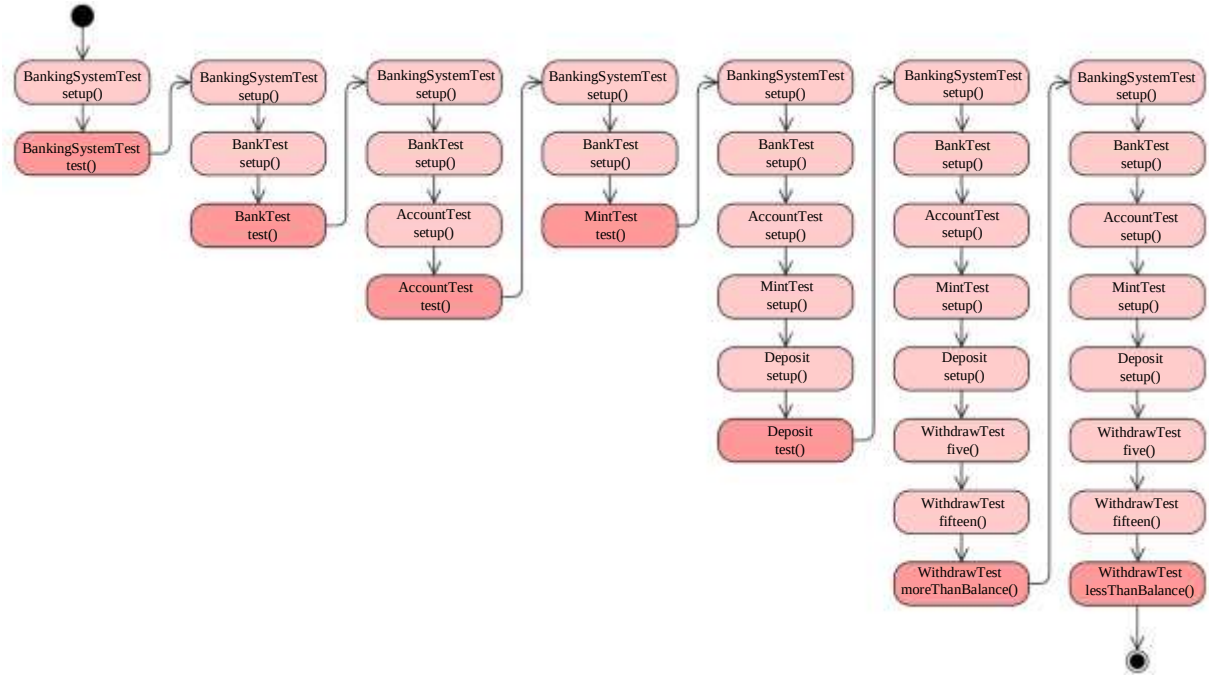

Fig. 16: Flow of execution for all tests without the execution reuse mode

Algorithm 1 shows the algorithm to generate ESISTMs considering the execution without reuse. First, from line 1 to 8, the algorithm creates a set including the tests to run. In line 9, the set of ESISTMs is initialized. Next, from line 10 to 26, the algorithm generates one ESISTM for each test. In line 11, the ESISTM for the current test is initialized. Besides that, a set for controlling the addition of implicit setup methods from singular test classes is initialized in line 12. Next, from line 13 to 15, a stack representing the dependency chain is initialized and the owner test class of the current test is pushed to the stack. In line 16, the current test method to run is pushed into the current ESISTM stack. Then, from line 17 to 24, the algorithm iterates over each test class of the dependency chain and pushes the implicit setup methods into the current ESISTM stack. It is worth mentioning that in line 19 the algorithm checks if the current class in the dependency chain was already added as a singular test class. If it was not, then, in line 20, the algorithm pushes the setup methods of the current class into the ESISTM stack. In line 21, the algorithm pushes the direct dependent classes of the current class in the chain stack. In line 22, the algorithm checks if the current class is singular. If it is, then it is added to the singular test classes set. This is done to avoid adding implicit setups from singular classes again in the same ESISTM. Next, in line 25, the algorithm puts the generated ESISTM in the set of ESISTMs. Finally, in line 27, the algorithm returns the set of ESISTMs to be run.

Algorithm 2 is used to generate the ESISTMs considering the execution with reuse. It has a behavior similar to the Algorithm 1 but has a special treatment for safe and unsafe tests. The first main difference is that the algorithm does not iterate over all tests. Instead of that, the algorithm starts the execution, from line 1 to 6, by initializing and populating the list of source vertex, i.e., the list of classes that are sources in EG. In line 7, the list of sources is ordered according to the length of the transitive closure of each source. This is important for optimizing the execution reuse of Estória. In line 8, the set of ESISTMs is initialized. Next, a set for controlling the already added tests is initialized in line 9. This set is used to avoid adding a given test in more than one ESISTM. Next, from line 10 to 39, the algorithm generates an ESISTM for each source test class. These ESISTMs will contain all safe test methods in the dependency chain (line 20 to 22) and may have the last unsafe test method that is a member of the current source test class (line 24 to 28). This unsafe test method can be added to the ESISTM because it will be the last method to be run. Finally, in line 40, the algorithm returns the set of ESISTMs to be run.

Algorithm 1 contains a ESISTM for each test and Algorithm 2 contains a ESISTM for each source test class. Except for only one unsafe test method in each source test class, the ESISTMs of Algorithm 2 do not contain unsafe test methods. Algorithm 3 uses the other two algorithms in order to generate the ESISTMs and run the tests. The algorithm starts by creating the sets of ESISTMs from line 1 to 6. If reuse mode is enabled, then the Algorithm 2 is used (line 3), otherwise, it is not used, and an empty set is initialized (line 5). In line 7, a test report is initialized. In line 8, a set of already executed test is initialized. Because the ESISTMs without reuse have tests that also appear in the ESISTMs with reuse, then this set is used for avoiding running a given test twice. From line 9 to 11 the algorithm creates a list of ESISTMs. This list is ordered, first the ESISTMs with code reuse and next the ESISTMs without code reuse. Next, from line 12 to 24, the algorithm runs the ESISTMs. In line 14, it is verified if the first test was not executed yet. This can be the case of tests that appear both in the ESISTMs with reuse and in the ESISTMs without reuse. Thus, these tests are run only in the ESISTM with reuse. From line 15 to 22, the algorithm runs the methods of the given ESISTM until it has no more methods to run. The result of the method execution is stored in line 17. If the executed method is a test (line 18), then the algorithm puts the result in the report (line 19) and puts the test method in the set of executed tests (line 20). Finally, in line 25, the algorithm returns the test report.

Algorithm 1: Generation of ESISTMs without execution reuse.

```
   Input: H = (W, F) such that H is an EG
   Output: The set of ESISTMs without execution reuse
1  testsToRun ← ∅
2  foreach r∈W do
3      foreach test ∈ r do
4          if isRunning(test) then
5              put(testsToRun, test);
6          end
7      end
8  end
9  esistmsWithoutExecutionReuse ← ∅;
10 foreach test 2 testsToRun do
11     esistm ← ∅;
12     singularsAdded ← ∅;
13     chain ← ∅;
14     ownerTestClass ← getTestClass(test);
15     push(chain, ownerTestClasse);
16     push(esistm, test);
17     repeat
18         class ← pop(chain);
19         if class ∉ singularsAdded then
20             foreach setup ∈ class do push(esistm,
               setup);
21             foreach dependent ∈ class do
                 push(chain, dependent);
22             if isSingular(class) then
                 put(singularsAdded, class);
```

```
23          end
24      until chain ≠ ∅;
25      put(esistmsWithoutExecutionReuse, esistm);
26  end
27  return esistmsWithoutExecutionReuse
```

Algorithm 2: Generation of ESISTMs with execution reuse.

```
    Input: H = (W, F) such that H is an EG
    Output: The set of ESISTMs with execution reuse
1   sources ← ∅;
2   foreach r∈W do
3       if isSource(H, r) then
4           add(sources, r);
5       end
6   end
7   sources ←
      sortByBiggestTransitiveClosure(sources);
8   esistmsWithExecutionReuse ← ∅;
9   testsAdded ← ∅;
10  foreach source ∈ sources do
11      unsafeTestAddedLast ← false;
12      esistm ← ∅;
13      singularsAdded ← ∅;
14      chain ← ∅;
15      push(chain, source);
16      repeat
17          class ← pop(chain);
18          foreach test ← class do
19              if isRunning(test) test ∉ testsAdded
                  then
20                  if isSafe(test) then
21                      push(esistm, test);
22                      put(testsAdded, test);
23                  else
24                      if source = class ∧¬
                          unsafeTestAddedLast then
25                          unshift(esistm, test);
26                          put(testsAdded, test);
27                          unsafeTestAddedLast ←
                              true;
28                      end
29                  end
30              end
31          end
32          if class ∉ singularsAdded then
33              foreach setup ∈ class do push(esistm,
                  setup);
34              foreach dependent ∈ class do
                  push(chain, dependent);
35              if isSingular(class) then
                  put(singularsAdded, class);
36          end
37      until chain ≠ ∅;
38      put(esistmsWithExecutionReuse, esistm);
39  end
40  return esistmsWithExecutionReuse
```

Algorithm 3: Execution of Estória.

```
    Input: H = (W, F) such that H is an EG
    Input: a reuse flag indicating if execution reuse
    mode is or not enabled
    Output: a test report
1   esistmsWithoutReuse ←
      generateEsistmsWithoutExecutionReuse;
2   if reuse then
3       esistmsWithReuse ←
          generateEsistmsWithExecutionReuse;
4   else
5       esistmsWithReuse ← ∅;
6   end
7   report ← ∅;
8   executed ← ∅;
9   allEsistms ← ∅;
10  add(allEsistms, esistmsWithReuse);
11  add(allEsistms, esistmsWithoutReuse);
12  foreach esistm ∈ allEsistms do
13      firstTest first(esistm);
14      if firstTest ∉ executed then
15          repeat
16              method ← pop(esistm);
17              result ← run(method);
18              if isTest(method) then
19                  put(report, result);
20                  put(executed, method);
21              end
22          until esistm ≠ ∅;
23      end
24  end
25  return report
```

Evaluation

In this section, we show the evaluation of Estória through three different perspectives: (1) Code reuse; (2) execution reuse; and (3) usage. In the first and second perspectives, we investigate the efficiency of Estória in promoting code and execution reuse, respectively. In the third perspective, we investigate the usage of Estória through an experiment in which Estória is used by programmers of a software company. Next, we present the experiments and our findings in each experiment as well.

Code Reuse Experiment

In this section we show an experiment conducted to investigate the efficiency of Estória in promoting code reuse. To conduct the experiment, we used an event

scheduling system that was developed as part of a project of a Computer Science graduate curse of the Universidade Federal de Santa Catarina (UFSC). To develop this system, the students used a set of agile practices, including software testing.

First, we selected a set containing 24 tests grouped in 4 test classes. We named this set as control group. The tests of the control group were written by graduate students enrolled in the course. In this project, the students created unit tests for an event scheduling system. The tests of the control group were written using classical fixture setup strategies, i.e., inline setup, implicit setup and delegated setup. Next, after the end of the course, we manually created the experimental group by rewriting the tests of the control group using the dependency setup strategy available in Estória. The experimental group was composed by the same 24 tests, but we redistributed them in 14 test classes. The tests of both control and experimental groups were written in Java. The control group tests were written for JUnit, while the experimental group tests were written for Estória.

Next, we collected the following measurements from both groups: (1) Number of code lines of test and helper classes; (2) sum of the number of repeated lines, excluding assertions; (3) sum of distinct repeated lines, excluding assertions; (4) sum of the number of repeated lines, including assertions; and (5) sum of distinct repetitions, including assertions. In the measurements 2, 3, 4 and 5 the following symbols were not counted as repetitions: @Test, @Before and @Fixture annotations, identical method declarations and block delimiter symbols.

Figure 17 shows the collected measurements for control and experimental groups. The experimental group presented a considerable reduction in the number of duplicate lines. The control group reached a total of 126 duplicate lines, excluding assertions, while the number of duplicate lines was reduced to 66 in the experimental group. Considering the total number of test code lines of each group, 40.91% of the lines of the control group corresponded to duplicate lines, while in the experimental group it was 20.37%. Comparing the control group with the experimental group, the latter had a reduction of 47.62% of the duplicate lines considering absolute values. It is interesting to note that the tests of the experimental group, i.e., the tests written for Estória, led to a considerable increase in test classes. While the control group had 4 test classes, the experimental group had 14 test classes. This behavior is expected because in Estória the reuse of fixture setups is strongly affected by the distribution of test classes. Estória motivates the use of the test class per fixture strategy (Meszaros, 2007). In the usage experiment we evaluate better the impact of increasing the number of test classes.

It is worth mentioning that the control group was written by four graduate students, including the first author of this work and the experimental group was written only by the first author of this work. Thus, we can cite as a treat to the validity of this experiment an implicit bias involved. The other students were not involved in the written of the experimental group because the scope of the course did not include Estória. To write the experimental group we preserved the tests behavior. We did not change any assertion of test setup code. The modifications made were only structural changes, i.e., we only moved test code for another classes and methods in order to apply the dependency setup of Estória. The sources used in the experiment can be found on GitHub[7].

Execution Reuse Experiment

To evaluate the execution reuse capabilities of Estória we conducted another experiment. The goal of the experiment was to identify potential differences in the time needed to run tests with and without the execution reuse mode of Estória. In the experiment, we extracted 32 acceptance tests of a system for course assessment developed to the Brazilian Ministry of Education and Culture (MEC). As these tests correspond to a feature that was being developed along with the realization of the experiment, it was easier to rewrite the tests from JUnit to Estória. Furthermore, we chose to use acceptance tests for Graphical User Interface (GUI) instead of unit tests because the typical slower execution of GUI tests facilitates the human perception of potential differences in execution times.

Once the tests were converted to Estória, we run them twice, with the execution reuse mode enabled and with the execution reuse mode disabled. In Fig. 18 we show the test report of the execution without reuse and in Fig. 19 we show the test report of the execution with reuse. While the execution of the tests without reuse took 1089 seconds to be completed, the execution with reuse took 131 seconds. This represents a reduction of approximately 8 times the execution time when the execution reuse mode is enabled. The sources used in the experiment can be found on GitHub[8].

Usage Experiment

To evaluate Estória usage by different participants, we conducted an experiment divided into four phases: (1) Preparation, (2) learning, (3) usage and (3) assessment. In the preparation phase, we gave the participants an ad hoc application, the Banking System, developed specifically for the experiment. We showed the participants the system requirements and a class diagram of the system as well. Next, in the learning phase, we explained to the participants four fixture setup strategies: Inline setup, implicit setup, delegated setup and the dependency setup strategy available in Estória. The goal was to establish a common base between all participants because some participants were more

[7] https://github.com/lucasPereira/alocacaoDeHorarios/tree/testesestoria
[8] https://github.com/lucasPereira/saas-teste-estoria

experienced in test development than others. In the usage phase, we asked each participant to finish the implementation of two tests for each fixture setup strategy learned in the previous phase. In total, each one of the 13 participants had to finish the implementation of 8 tests. In this phase, for each fixture setup strategy we collected the time needed to finish the implementation and verified if the strategy was implemented properly. Finally, in the last phase, the assessment, we asked each participant to answer a questionnaire.

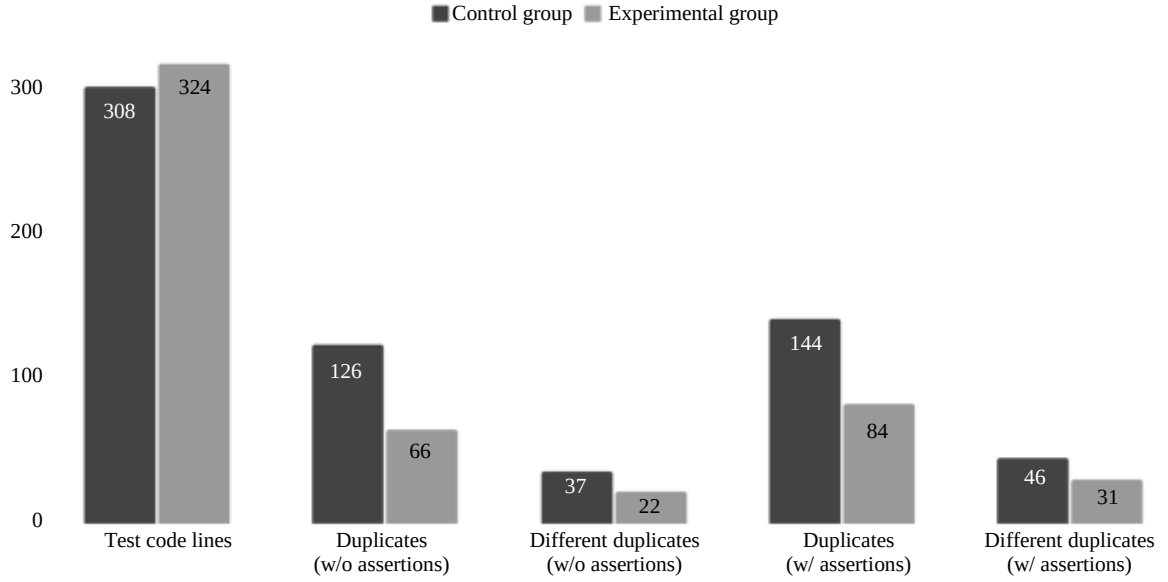

Fig. 17: Measurements of the code reuse experiment

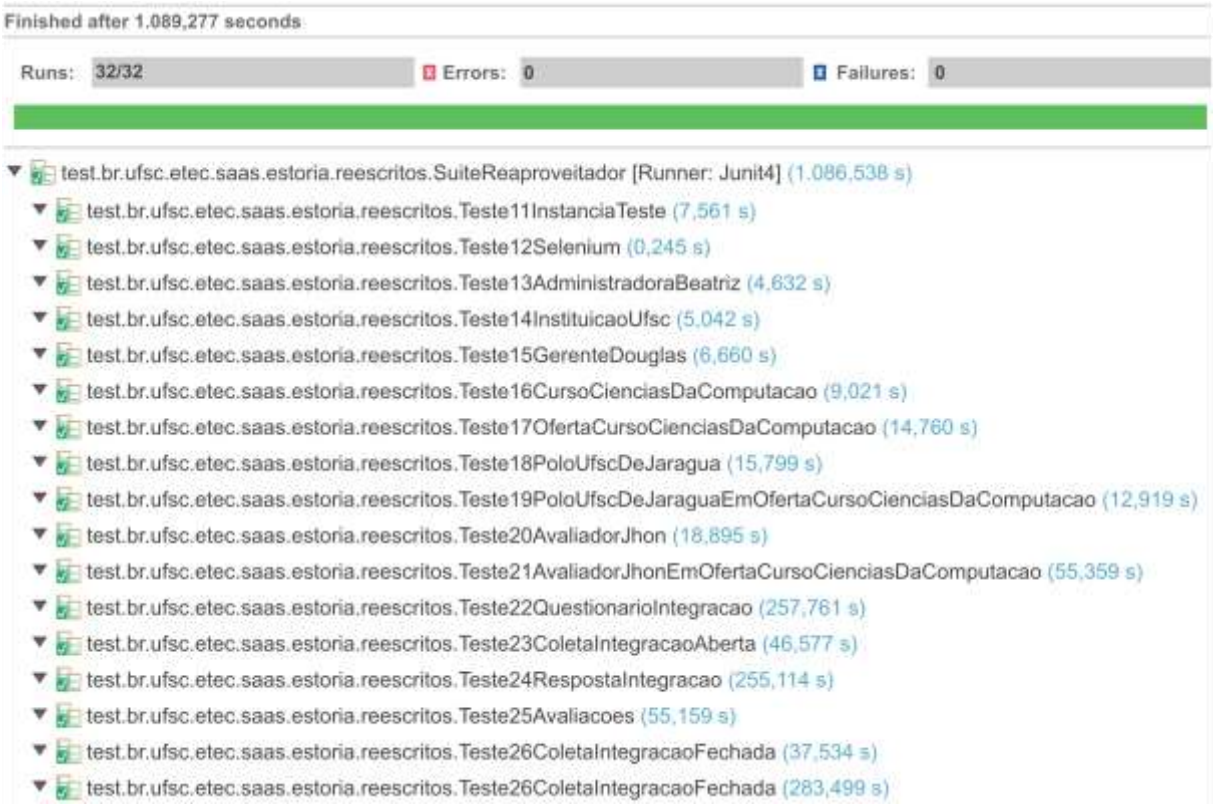

Fig. 18: Test report of the execution experiment without reuse

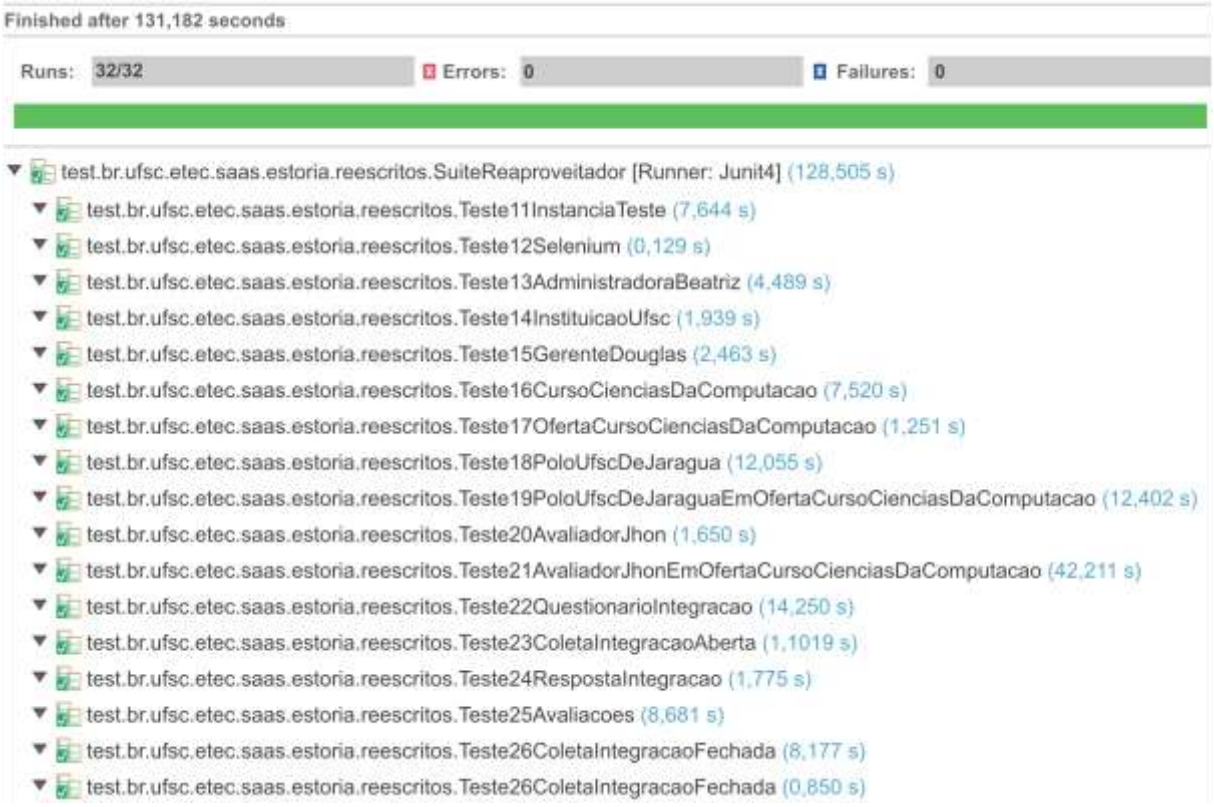

Fig. 19: Test report of the execution experiment with reuse

Table 1: Usage phase results of case of study

Strategy	Average	Std. Deviation	Shortest	Largest	Incorrect
Inline setup	13.9	11.7	3	40	3
Implicit setup	9.1	5.3	5	20	4
Delegated setup	9.3	5.5	5	20	5
Dependency setup	10.6	9.6	5	34	4

Table 1 summarizes the experiment results from the usage phase. The table shows the average time needed by all the 13 participants to finish the implementation considering each fixture setup strategy. Each participant implemented 2 tests for each strategy. Standard deviation, shortest time and largest time to complete the task are also shown in the table. These metrics are only relative to the tests implemented properly. In addition, the total number of tests not properly implemented considering each strategy is shown in the last column of the table, e.g., in inline setup, 3 out of 26 tests were implemented incorrectly. While the implicit setup strategy had the best average time compared to the others, the inline setup had the worst results.

In the beginning of the assessment phase, we tried to identify the experience of the participants regarding the following topics: Software development, software testing, Java language and fixture setup strategies. The main goal of the initial part of the questionnaire was to trace a profile of the participants. The results presented in Table 2 show a quite heterogeneous environment regarding the experience of the participants. In total, 13 employees of a technology company participated in the experiment. All participants reported having at least some experience with software development. Regarding software testing, the experience of the participants was considerably low, 3 participants reported not having any experience and no participant reported having high experience with software testing. Regarding experience with the Java language, the language used in the experiment, a high number of participants declared themselves without Java experience. In total, 4 participants reported not having any experience with Java at all.

Table 3 shows the participants' experience with fixture setup strategies before the experiment. We can observe that 7 of 13 participants declared they knew the inline setup strategy; 4 participants knew the implicit setup and only 2 participants knew the delegated setup strategy.

DOI: 10.3844/jcssp.2021.90.111

Table 2: Participants experience results of case of study

Topic	No experience	Low	Medium	High
Software development	0	5	4	4
Software testing	3	5	5	0
Java language	4	3	4	2

Table 3: Participants experience with fixture setup strategies results of case of study

Strategy	Knew before the experiment	Did not know before the experiment
Inline setup	7	6
Implicit setup	4	9
Delegated setup	2	11
Dependency setup	0	13

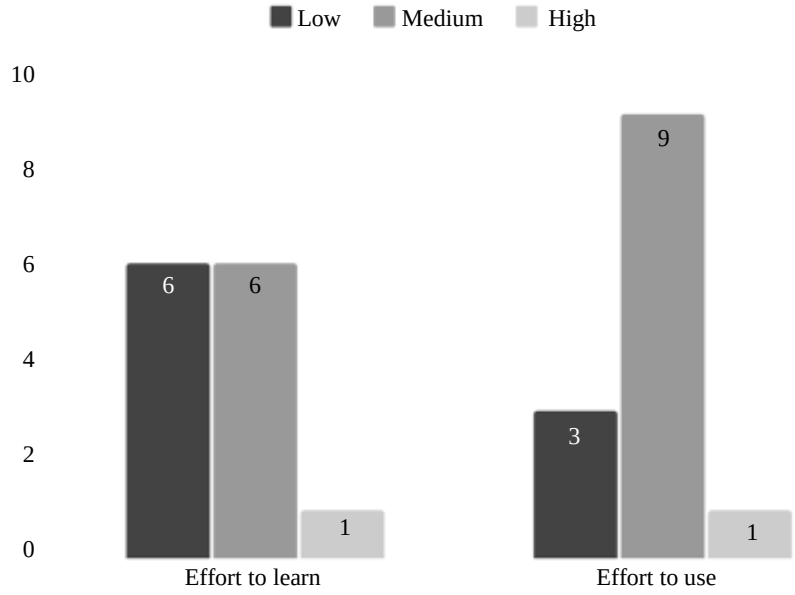

Fig. 20: Effort to learn and use results of the study of case

Next, we asked each participant to measure the effort needed to learn and to use the dependency setup available in Estória. We also asked them to evaluate how useful they considered the dependency strategy to be. Figure 20 shows the results considering the effort needed to learn and to use the dependency setup. Regarding the effort to learn, 1 participant considered that the effort was high, while the other 12 participants were equally divided into low and medium effort. The results of the effort to use were similar, but with fewer low-effort evaluations and more medium-effort evaluations. Moreover, we asked the participants to answer how useful they considered the dependency setup. In total, 10 participants considered it useful and 3 considered it very useful. No participant considered the dependency setup strategy to be not useful at all.

In the last question of the questionnaire, each participant could write comments about the fixture setup strategy. Only two participants answered this question. One participant said that the proposal is "interesting and easy to use". The other participant argued that Estória facilitates the removal of redundancies in test code, but he added that redundancies sometimes are useful to identify potential problems in test code. The participant said that "it is easy to write some field wrong, but it is harder to make the same mistake again" and that "redundancy facilitates the identification of this type of error".

Analysis of the Experiments

In this section, we presented three experiments to evaluate Estória. The first two experiments investigated the reuse capabilities of the framework and the last one investigated the adherence to Estória considering the use by programmers from industry. Our goal was to establish a foundation to evaluate the effectiveness and efficiency of Estória. Regarding the effectiveness, we wanted to know if Estória is suitable to promote code and execution reuse. Regarding the efficiency, we wanted to know whether Estória is simple and fast to use. While

reuse experiments cover the evaluation in terms of the effectiveness, the usage experiment covers the evaluation in terms of the efficiency.

According to the results of the experiments, Estória seems to be effective considering the used projects. In the code reuse experiment, the framework promoted a reduction of 47.62% of the duplicate lines, while in the execution reuse experiment, the framework achieved a reduction of approximately 8 times the execution time. It worth mentioning some treats to validity regarding these two experiments. In the code reuse experiment the first author of this work participate of the implementation of the control group tests and was responsible for writing the experimental group tests. Thus, there is an intrinsically bias involved. The execution reuse experiment has the same treat to validity because the first author of this work also wrote the tests of this experiment. Besides that, regarding the external treat to validity, i.e., the capacity to generalize the results for other projects, we can cite that we used only one project for each experiment. This reduced the internal treat to validity because we performed the experiments in a more controlled environment but reduced the capacity to generalize the results.

Estória also has been shown to be efficient, since in the usage experiment no significant difference between Estória and classical fixture setup strategies, such as implicit and delegated setup, was observed. Furthermore, the framework was assessed by programmers from industry and it was considered useful by most of them. In total, 10 participants considered it useful and 3 considered it very useful. No participant considered Estória to be not useful at all. It worth mentioning that the participants do not knew Estória before the experiment and do not have any relationship with the authors of this work. However, we also need to point out that only 13 programmers participated of the experiment. In order to generalize the results, we need to perform the experiment with more participants.

Conclusion and Future Works

In this study we presented a model called dependency model, which defines a new relationship between test classes. The dependency model establishes a consumer/provider relationship in which a consumer test class may have one or more provider test classes. This relationship implies that the consumer class will use the implicit setup of the provider classes. From the dependency model, we developed a new fixture setup strategy, namely the dependency setup and implemented the strategy in Estória, a JUnit extension framework. Estória enables the use of dependency setup through two different modes: One that promotes code reuse and another that promotes both code and execution reuse.

Unlike other fixture setup strategies, dependency setup enables the reuse of code between test classes without affecting the structure of the involved classes. Thus, it is possible to use the implicit setup of a test class as starting point for the implicit setup of another test class. This enables the creation of a chain of reuse, reducing the test code duplication. Furthermore, the dependency setup strategy was implemented in a framework with a built-in mechanism to promote execution reuse. It is important to note that test frameworks typically do not provide execution reuse built-in mechanisms.

Initial experiments showed that the dependency strategy leads to a reduction of 47.62% of the duplicate lines of the test code. In addition, execution reuse mode of Estória allowed us to reduce the execution time by a ratio of 8. We also conducted an experiment to evaluate the usage of Estória by different participants of a technology company. Out of 13 participants, 10 considered the strategy to be useful and 3 considered the strategy very useful.

We also compared our model with other models for reusing code or execution of tests. Estória and DbUnit (Christensen *et al*., 2006) models were similar regarding the approach, i.e., both approaches promote code and execution reuse through definition of dependencies between test classes. However, unlike DbUnit, Estória is application independent. Also, fixture objects in Estória can be shared across test classes, which it is not possible in DbUnit. The model proposed by Mugridge and Cunningham (2005) was compared with the Estória model as well. Both approaches promote code and execution reuse, but the work of Mugridge and Cunningham (2005) is applied to Fit specifications while our approach is applied to test code. Besides that, in our model tests can be run individually, while in Mugridge and Cunningham (2005) the tests are connected together and are run as a whole, loosing traceability. Comparing Estória model with Picon (Longo *et al*., 2015), we observed that both models promote code reuse. However, Picon cannot be used to promote execution reuse. Also, Picon can only be applied to promote reuse of POJOs. The most similar model to Estória that was found in literature was SolUnit (Medeiros *et al*., 2019). Regarding execution reuse, SolUnit model has one advantage when compared with Estória model: SolUnit automatically identifies test setups that can be reused by other tests and in Estória this is done manually by developers. However, Estória is application independent while SolUnit can be applied only to smart contract tests. Besides that, Estória also promotes code reuse, while SolUnit does not.

To summarize our contribution to the research in the testing area, we presented a generic model to promote both code and execution reuse between test classes. We also developed a testing framework (available at https://github.com/lucasPereira/estoria) that implements this model as a new fixture setup strategy. This testing framework has a built-in mechanism that promotes both

code and execution reuse. As future work, we intend to investigate the possibility of automatic identification of safe tests. In this way, Estória could automatically infer if a test is safe or not. Currently, the test developer must explicitly mark a test as safe to promote execution reuse. We also expect to use Estória in real projects.

Author's Contributions

Lucas Pereira da Silva: Proposed and defined the dependency model, implemented Estória, carried out the experiments and wrote the paper.

Patrícia Vilain: Proposed and defined the dependency model, carried out the experiments and wrote the paper.

Ethics

The authors confirms that this article has not been published in any other journal. The corresponding author confirms that all the authors have read and approved the manuscript and there are no ethical issues involved.

References

Adamoli, A., Zaparanuks, D., Jovic, M., & Hauswirth, M. (2011). Automated GUI performance testing. Software Quality Journal, 19(4), 801-839.

Agrawal, H., Horgan, J. R., Krauser, E. W., & London, S. A. (1993, September). Incremental regression testing. In 1993 Conference on Software Maintenance (pp. 348-357). IEEE.

Alvestad, K. (2007). Domain Specific Languages for Executable Specifications (Master's thesis, Institutt for datateknikk og informasjonsvitenskap).

Beneden, P. J. (1876). Animal parasites and messmates. HS King & Company.

Berner, S., Weber, R., & Keller, R. K. (2005, May). Observations and lessons learned from automated testing. In Proceedings of the 27th international conference on Software engineering (pp. 571-579).

Bertolino, A. (2007, May). Software testing research: Achievements, challenges, dreams. In Future of Software Engineering (FOSE'07) (pp. 85-103). IEEE.

Borg, R., & Kropp, M. (2011, May). Automated acceptance test refactoring. In Proceedings of the 4th Workshop on Refactoring Tools (pp. 15-21).

Bourque, P., Fairley, R. E., & Society, I. C. (2014). Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0. IEEE Computer Society Press, Washington, DC, USA.

Christensen, C. A., Gundersborg, S., De Linde, K., & Torp, K. (2006, December). A unit-test framework for database applications. In 2006 10th International Database Engineering and Applications Symposium (IDEAS'06) (pp. 11-20). IEEE.

da Silva, L. P., & Vilain, P. (2016, June). Execution and code reuse between test classes. In 2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA) (pp. 99-106). IEEE.

da Silva, L. P., & Vilain, P. (2017). Reuse of Fixture Setup between Test Classes. In SEKE (pp. 224-229).

Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., & Horowitz, B. M. (1999, May). Model-based testing in practice. In Proceedings of the 21st international conference on Software engineering (pp. 285-294).

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., & Hierons, R. (2018, March). Smart contracts vulnerabilities: a call for blockchain software engineering?. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 19-25). IEEE.

Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. IEEE Transactions on software Engineering, 31(3), 226-237.

Essam, J. W., & Fisher, M. E. (1970). Some basic definitions in graph theory. Reviews of Modern Physics, 42(2), 271.

Fowler, M. (2018). Refactoring: improving the design of existing code. Addison-Wesley Professional.

Freeman, S., & Pryce, N. (2009). Growing object-oriented software, guided by tests. Pearson Education.

Garousi, V., & Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. Journal of systems and software, 138, 52-81.

Greiler, M., Van Deursen, A., & Storey, M. A. (2013a, March). Automated detection of test fixture strategies and smells. In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (pp. 322-331). IEEE.

Greiler, M., Zaidman, A., Van Deursen, A., & Storey, M. A. (2013b, May). Strategies for avoiding text fixture smells during software evolution. In 2013 10th Working Conference on Mining Software Repositories (MSR) (pp. 387-396). IEEE.

Haugset, B., & Hanssen, G. K. (2008, August). Automated acceptance testing: A literature review and an industrial case study. In Agile 2008 Conference (pp. 27-38). IEEE.

Kamalrudin, M., Sidek, S., Aiza, M. N., & Robinson, M. (2013). Automated acceptance testing tools evaluation in Agile software development. Sci. Int, (4), 1053-1058.

Kumar, D., & Mishra, K. K. (2016). The impacts of test automation on software's cost, quality and time to market. Procedia Computer Science, 79, 8-15.

Lambiase, S., Cupito, A., Pecorelli, F., De Lucia, A., & Palomba, F. (2020, July). Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In Proceedings of the 28th International Conference on Program Comprehension (pp. 441-445).

Longo, D. H., Wilges, B., Vilain, P., & Cislaghi, R. (2015). Fixture Setup through Object Notation for Implicit Test Fixtures. Journal of Computer Science, 11(6), 794.

Mattsson, M., Bosch, J., & Fayad, M. E. (1999). Framework integration problems, causes, solutions. Communications of the ACM, 42(10), 80-87.

Medeiros, H., Vilain, P., Mylopoulos, J., & Jacobsen, H. A. (2019, November). Solunit: A framework for reducing execution time of smart contract unit tests. In Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (pp. 264-273).

Meszaros, G. (2007). xUnit test patterns: Refactoring test code. Pearson Education.

Meszaros, G., Smith, S. M., & Andrea, J. (2003, August). The test automation manifesto. In Conference on Extreme Programming and Agile Methods (pp. 73-81). Springer, Berlin, Heidelberg.

Mugridge, R., & Cunningham, W. (2005, June). Agile test composition. In International Conference on Extreme Programming and Agile Processes in Software Engineering (pp. 137-144). Springer, Berlin, Heidelberg.

Nofer, M., Gomber, P., Hinz, O., & Schiereck, D. (2017). Blockchain. Business & Information Systems Engineering, 59(3), 183-187.

Ramler, R., & Wolfmaier, K. (2006, May). Economic perspectives in test automation: balancing automated and manual testing with opportunity cost. In Proceedings of the 2006 international workshop on Automation of software test (pp. 85-91).

Sobernig, S., & Zdun, U. (2010, July). Inversion-of-control layer. In Proceedings of the 15th European Conference on Pattern Languages of Programs (pp. 1-22).

Sweet, R. E. (1985). The Mesa programming environment. ACM SIGPLAN Notices, 20(7), 216-229.

Tiwari, R., & Goel, N. (2013). Reuse: reducing test effort. ACM SIGSOFT Software Engineering Notes, 38(2), 1-11.

Tonelli, R., Destefanis, G., Marchesi, M., & Ortu, M. (2018). Smart contracts software metrics: a first study. arXiv preprint arXiv:1802.01517.

Tsai, W. T., Saimi, A., Yu, L., & Paul, R. (2003, November). Scenario-based object-oriented testing framework. In Third International Conference on Quality Software, 2003. Proceedings. (pp. 410-417). IEEE.