Revista de Informática Aplicada, Volume 13, Número 2, 2017

A Generalized Functional Testing Approach for Application Services Development

Thiago Nascimento Rodrigues[1]
Regional Election Court of Paraná - TRE/PR
Systems Development Department
Rua João Parolin, 224, 80220-902, Curitiba, PR, Brazil

Resumo

O projeto de arquiteturas de testes flexíveis para a avaliação de sistemas tem desafiado equipes de testadores e desenvolvedores ao longo de distintos processos de desenvolvimento de *software*. Os desafios envolvem arquitetura de testes tanto módulos simples como camadas inteiras de sistemas. Neste cenário, o presente trabalho propõe uma abordagem de testes baseada em testes funcionais. Cada módulo de sistema (*application service*) é encarado como uma caixa-preta e uma abordagem alternativa de projetar casos de testes é apresentada. O conceito de testes funcionais generalizados é analisado sob a perspectiva de casos de testes mais informativos e mais aptos a garantir a qualidade de um dado *application service*. O foco no reuso de instâncias de dados de testes destaca o pontencial do modelo sugerido.

Palavras-chave: testes funcionais, application services, desenvolvimento de sistemas.

Abstract

The design of flexible test architectures for software evaluation has been challenging testers and developers teams throughout distinct software development processes. The challenges involve both a simple software module and a whole software layer. In this scenario, this work proposes a testing approach based on functional tests. Each software module (application service) is seen as a black-box, and an alternative way to design tests case is presented. The concept of generalized functional tests is analyzed under the perspective of tests case more informative and more able to ensure the quality of an application service. The particular focus on the reuse of test data instances highlights the potential of the suggested model.

Keywords: functional tests, application services, software development

[1]Email: nascimenthiago@gmail.com

1 Introduction

The complexity of applications is growing considerably in corporate environments, while designing and developing systems need to satisfy hundreds or thousands of separated requirements. In fact, it is not uncommon that modern applications may be characterized by attributes like persistent data manipulation, concurrent access data, integration with other legacy systems, complex business logic, and so on. Interactions with these kinds of applications tend to be complex, since they typically involve transactions across multiple resources and the coordination of several responses to an action. Encoding the logic of these interactions demands an appropriate code architecture in order to avoid duplication of business logic, and allow efficient maintenance apart from reducing the complexity of systems (STÄBER, 2008).

In this context, the Service Layer Pattern has been proposed as a right way to coordinate complex interactions with the core of monolithic applications or with the service-based systems. According to (FOWLER, 2002), a Service Layer defines an application boundary and its respective set of available operations from the perspective of interfacing client layers. Thereby a Service Layer never exposes details of the internal processes or the business entities (TEAM, 2009).

A common Service Layer building approach for monolithic applications involves providing as a set of thicker classes that implement the application logic directly (FOWLER, 2002). Each such class constitutes an application service which provides a central location to the encapsulated business logic (ALUR; MALKS; CRUPI, 2013). Figure 1 presents how both patterns Service Layer and Application Service are interrelated in a multi-tier application architecture.

From the software quality perspective, application services must be tested as any system component. This paper describes a black-box testing architecture for applications services. Although the focus is on monolithic applications, it is not restricted to it. The remaining of this paper is organized as follow. In Section 2, the classical functional testing approach is described. Section 3 is dedicated to detail the concept of generalized functional tests. In Section 4, an architecture model is presented as a way of implementing the concept described in the previous section. Conclusions and directions for future work are included in the last section.

2 Classical Functional Tests

According to (BARBOSA et al., 2000), a crux of testing activity is the design and the quality evaluation of a particular set of test cases T used for testing a specific product P independently of testing step. Based on this, the input data used by each test case means a critical factor for ensuring the software quality. However, it is impracticable to use all input data domain in order to evaluate functional and operational features of a product under testing. Thereby, a software testing architecture should be flexible enough to explore input data appropriately. At the same time, it must be rigid enough to provide a roadmap toward successful construction of software.

One important testing strategy is black-box, data-driven, or input/output driven testing. The use of this method implies view the program as a black box. The goal is to be completely unconcerned about the internal behavior and structure of the program. Instead, concentrate on finding circumstances in which the program does not behave according to its specification (MYERS; SANDLER, 2004).

Functional testings are a kind of black-box testing in which the selection of test cases is based on the requirement or design specification of the software entity under testing. Examples of expected results sometimes are called test oracles. They include requirement and design specifications, hand calculated values, and simulated values. So, functional testing emphasizes on the external behavior of the software entity (LUO, 2001). In this paper, applications services are seen as entities that

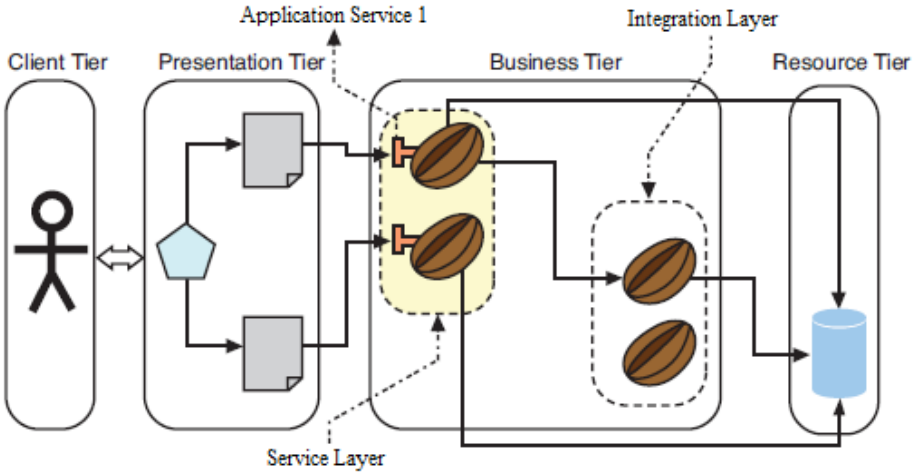

Figure 1: A Multi-Tier Application Architecture

must be submitted to functional testings.

In this way, a service layer S of a product P may be defined as composed by a finite number of applications services α_i, i.e., $S = (\alpha_1, \alpha_2, \cdots, \alpha_n)$. Following this notation, a set of tests T for S may be described as $T = (\tau_1, \tau_2, \cdots, \tau_n)$, where each $\tau_i \in T$ corresponds to a chain of test cases for an application service $\alpha_i \in S$. Since any application service works in response to some input or set of inputs, let $I = (\delta_1, \delta_2, \cdots \delta_m)$ be the type set of inputs data to be used by a whole service layer S. Each subset $(\delta_j, \cdots, \delta_k) \subset I$ corresponds to a data type logical grouping which is related to the set of parameters required by an application service.

Based on this model, an application service $\alpha_i \in S$ can be better defined as

$$\alpha_i : \delta_i \mapsto \epsilon_i$$

where α_i processes input data of the type δ_i and generates output data of the type ϵ_i. Instances of δ_i and ϵ_i are denoted by $\delta_i = (\delta_i^1, \delta_i^2, \cdots, \delta_i^j, \cdots)$ and $\epsilon_i = (\epsilon_i^1, \epsilon_i^2, \cdots, \epsilon_i^j, \cdots)$ respectively. Therefore, a chain τ_i of tests case for α_i can be defined as $\tau_i = (\tau_i^1, \tau_i^2, \cdots, \tau_i^k)$. So, given any data instance δ_i^m of the type δ_i, a test case $\tau_i^j \in \tau_i$ is denoted as follow

$$\tau_i^j(\delta_i^m, \alpha_i) : \alpha_i(\delta_i^m) \mapsto \epsilon_i^m.$$

In this way, a test case $\tau_i^j \in \tau_i$ checks if the logic implemented by the application service α_i processes the input data δ_i^m into the output data ϵ_i^m according to the specified.

This model is a way to describe the traditional functional testings approach for any kind of applications services. Nevertheless, it presents a limitation related to the input data employed to test each application service. Although the $\alpha_i \in S$ invocation involves supplying data instances of the type δ_i, it is not rare that other auxiliary kind of data should be made available so that the α_i application service can work correctly.

By way of example, let $\delta_i' = (\delta_i^1, \delta_i^2)$ be a data instance of the type δ_i to test the α_i application service by means of a set of tests τ_i. Suppose it is necessary to make available data instances of other δ_j and δ_k complementary data types in order to (i) make α_i able to run, and (ii) use τ_i to check the generated output. In this way, if δ_i' is made available, it implies that instances of δ_j and δ_k must also be made available in order to ensure the correct and complete execution of α_i. Let $\delta_j' = (\delta_j^x, \delta_j^v)$ and $\delta_k' = (\delta_k^y, \delta_k^w)$ be instances of δ_j and δ_k respectively. Therefore, a complete set of data instance necessary to the correct and complete tests execution should be as follow

$$((\delta_i^1, \delta_j^x, \delta_k^y), (\delta_i^2, \delta_j^v, \delta_k^w)).$$

It is important to notice that the successful execution of any test case from τ_i implies that only δ_i' data input be used in order to evaluated α_i.

So, data instances of other data types which have been made available, that is δ'_j and δ'_k, are not used for any validation or verification by tests from τ_i.

In this scenario, let $\tau_j \in T$ be a set of tests built for another application service $\alpha_j \in S$. The input data supposed to be used by it comprise two data types: (δ_j, δ_k). In this case, the application service may be described as follow

$$\alpha_j : (\delta_j, \delta_k) \mapsto \epsilon_j,$$

where instances of (δ_j, δ_k) are denoted by $((\delta_j^1, \delta_k^1), (\delta_j^2, \delta_k^2), \cdots)$. Another assumption is that the correct execution of α_j demands that auxiliary data instances of another data type δ_i must be made available, that is, δ_i^x and δ_i^y. Based on all those assumptions, the execution of the set of tests τ_i and τ_j requires that input data instances must be made available according to the configuration described by Figure 2.

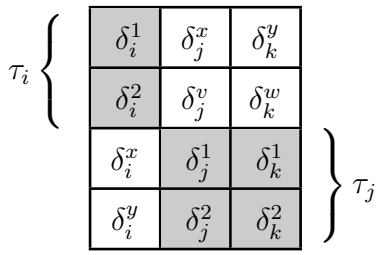

Figure 2: Input data arrangement for τ_i and τ_j set of test cases

At this configuration, the (δ_j^x, δ_k^y) and (δ_j^v, δ_k^w) data chunk will not be used by τ_j set of test cases as well as δ_i^x and δ_i^y will not be processed by τ_i. In other words, test cases for the α_i application service will only use the dataset defined to it, that is, instances of δ_i. Any other data of the type δ_i that had been made available will not be used by tests cases from τ_i. Actually, when some data input are defined as part of a respective set of test cases, this constitutes a restriction over data instances that can be used by each test case. Moreover, as input data are usually coupled with the respective test case code, the code coverage reached by the tests is likewise compromised.

Those limitations have motivated the arising of alternative functional testings approaches. In fact, the functional testing model presented in this work offers a way to improve the use of input data instances by test cases.

3 Generalizing Functional Tests

The proposed functional testing model aims the better usage of input data sample prepared for each test case. Two principles guide this model:

1. To take away the input data definitions from test case code.
2. To allow each test case to access every data instance made available to any application service under testing.

Four main components were defined according to those principles:

- **Initial State:** It is used to define input data instances for the test cases.
- **Execution Context:** It holds on all preconditions necessary to the correct and complete application service execution.
- **Test Scenario:** It is responsible to define all pre-conditions required by each business rule of an application service under testing. All pre-conditions are loaded into the Execution Context, including the data instances defined in the Initial State.
- **Test Repository:** It is a set of generalized test cases which uses all data made available for each test scenario.

Figure 3 presents a sketch model comprising those components. The integration and use of them are centralized in a testing engine which coordinates each test scenario execution. Since the initial state holds on any type of input, no data is defined in test cases. If some additional precondition is demanded by test cases from Test Repository, each test scenario is responsible to provide it. Otherwise, test scenarios should just make available its specific initial state to test repository. The application service invocation could be done by both test engine

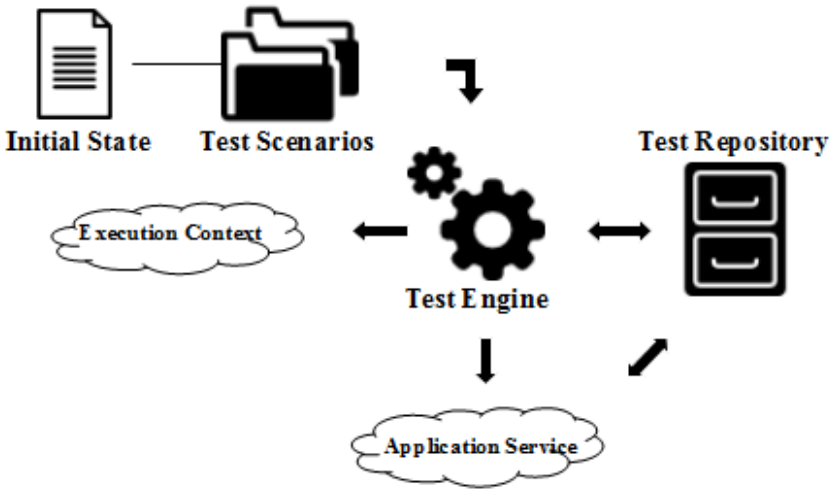

Figure 3: Proposed Components Model

and test repository. This invocation depends on each application service particularity. However, any output from the application service must be returned to test repository since it contains all generalized test cases.

The concept of test scenario applied in this architecture is slightly different from the traditional definition. According to (KANER, 2003a), a test scenario is a test based on a hypothetical story, used to help a person think through a complex problem or system. The story involves a complex use of the program or a complex environment or a complex set of data. This aspect there exists in the proposed architecture since the underlying idea of the scenarios of tests is to check an application service - the complex system - through a complex set of data.

Moreover, a test scenario may be seen as a black box in which test cases are designed to execute scenarios of use cases (BOARD, 2012). So, test scenarios are defined for an individual project or product at different stages such as unit testing, integration testing, interface testing and system testing (LIMAYE, 2009). An application service under testing is the individual product for which test scenarios are defined. However, the proposed model is focused on design test cases specifically in the test repository. Ideally, test scenarios should contain only dataset (Initial States) and no one test case. So, a test scenario C_i for an application service α_i can be defined as follow

$$C_i : (\Delta_i, \Gamma_i, \tau_i) \mapsto \Theta_i,$$

where Δ_i is the initial state (row input data instances) to test an application service α_i, Γ_i comprises any other pre-conditions required by the business rules implemented by α_i, τ_i is related to the set of test cases built to evaluate α_i, and Θ_i is the execution context in which all pre-conditions necessary to the correct and complete execution of the tests are loaded.

Another new concept arises from the proposed model. It has mentioned that Test Repository holds *Generalized Functional Tests* (GFT). They can be understood as agnostics tests regarding data inputs. In fact, each generalized test does not know which instance of data it will use in order to evaluate an application service. Instead, it knows just the data type to use in the test activity. In this way, a generalized test consumes all data chunk that corresponds to the type understandable by it. All data chunk is made available by each initial state from each test scenario.

Furthermore, it is important to notice where test cases are defined. In this model, there are two places where to put each one of them: in a specific test scenario or in the test repository. Defining a test case in a specific test scenario corresponds to the classical approach. On the

other hand, to put a test case in the test repository means it was promoted to a generalized test. An ideal situation would be to promote all test cases to generalized tests.

Basing on those components and definitions, let $C_i = \{C_i^1, C_i^2, \ldots, C_i^n\}$ a set of test scenarios implemented for an application service α_i. Using the proposed architecture, for each test scenario $C_i^j \in C_i$, the following steps are executed:

1. The data instances defined in the initial context are loaded into the execution context by the test scenario C_i^j. Hence, the input data become available to test repository.

2. Any other required preconditions are set up by C_i^j and loaded into the execution context.

3. Each test case $\tau_i^j \in \tau_i$ from test repository is executed.

This flow makes possible a new test scenario C_i^{n+1} may be included in C_i. In this case, all instance of data made available for it, will be used by all generalized test cases from test repository. Indeed, if C_i^{n+1} contents some test case that may be promoted to the test repository, they will reciprocally use the data sample made available for all others test scenarios.

Following these definitions, given a set of data $\delta_i' \subset \delta_i$ and a chain of tests case τ_i, a test case $\tau_i^j \in \tau_i$ for an application service α_i may be described as

$$\tau_i^j(\delta_i^j, \alpha_i) : \alpha_i(\delta_i^j) \mapsto \epsilon_i^j, \delta_i^j \in \delta_i'.$$

In this model, each test case can only use data from δ_i' set. In fact, the whole set of data used for testing α_i is defined as part of a test case. Moreover, the execution of each specific test case remains restricted to use only specifics sample of data $\delta_i^j \in \delta_i'$. Actually, no other data instances of the same δ_i^j type will be used by the test case because it was not defined as part of the respective test case. Based on this limitation, a test case can be converted into a generalized test case by taking away from it any parameter which limits its execution scope. A test case redesigned according to this principle can be defined as follow

$$\tau_i^j(\delta_i^j, \alpha_i) : \alpha_i(\delta_i^j) \mapsto \epsilon_i^j, \forall \delta_i^j \in \delta_i.$$

In this rewritten test case, it is important to notice that input parameters of τ_i^j are no more restricted to a specific subset $\delta_i' \subset \delta_i$. Actually, this redesigned test makes possible any δ_i^j belonging to a data source $(\delta_1, \delta_2, \cdots \delta_i, \cdots, \delta_m)$ can be used as input data. Thereafter, the τ_i^j test case will use any available instance of δ_i to testing α_i application service.

4 Implementing GFT

As aforementioned, the new proposed model for functional testings is based on the generalized test cases concept. This kind of test case is not restricted to a specific set of input data. Actually, a generalized test case knows only the data type used by the application service submitted to test. It does not know any particular data instance.

Although generalized test cases incorporate this new design issue, they should remain incorporating the requirements for good test cases as suggested by (KANER, 2003b). In this way, testers or programmer developers must continue narrowing their focus to build generalized test cases more reliable, more useful for troubleshooting, and more informative. However, given that a functional test may be promoted to a generalized test case, it is necessary to enhance it. Such enhancement involves the implementation of two additional aspects:

1. **Plain Logic:** The idea behind the Plain Logic concept is the re-implementation of the business logic under testing. In fact, if an application service under testing implements a business rule R_i, a generalized test case for it must contain a $\tilde{R}_i$ as a re-implementation of R_i. However, the main difference between R_i and $\tilde{R}_i$ is the way of building it. In the application service, R_i should be implemented according to an elaborated architecture, which

probably involves some design patterns, software frameworks, and any other good techniques to support the development activity. Nevertheless, the $\tilde{R}_i$ implementation in a generalized test case must not use any additional resource (software or technique). Actually, $\tilde{R}_i$ is considered as plain logic if its respective code corresponds to a simply way to implement it, that is, it is easy to evaluate. In this way, although the generalized test case logic is more complex than a classical test case (almost no logic), the evaluation of it is simplified by the way of to implements the plain logic as an enhancement feature.

2. **Expected Calculated Values:** According to (SAINI; RAI, 2013), an expected result is a final outcome which is defined based on requirements specifications for the test execution. In fact, in traditional test cases, expected values are explicitly *defined* in the code and compared against the result received after applying the test data to software. However, for the generalized test cases, this concept is extended to expected calculated values. Instead of setting the expected value, it is *calculated* by plain logic. In this way, a test case assertion involves a comparison between two calculated values: one calculated by the application service (execution invoked by the test case) and another one calculated by the test case itself (via plain logic).

Based on these enhancement features, the generic code structure of a generalized test case differs significantly from the traditional test case. For instance, the Algorithm 1 presents a common way to implement test cases. It is important to highlights some intrinsic characteristics:

- Test data instances defined in test case code. (Line 1)
- Expected values set up in the test case code. They are used to compare against the outcome from the software execution - the α_i application service. (Line 2)
- There is no logic implemented in test case code.

Algorithm 1 Traditional Test Case

Input: Application Service α_i
Output: Success or Fail

1: Build a *testDataT* as a data instance of Type T;
2: Build a *expectedValueT* as an expected value of T Type;
3: *actualResultT* $\leftarrow \alpha_i(testDataT)$;
4: **if** Assertion comparing (*expectedValueT*, *actualResultT*) is OK **then**
5: Return SUCCESS;
6: **else**
7: Return FAIL;
8: **end if**

On the other hand, test case promotion to a generalized functional testing implies several modifications in the code structure as may be seen in the Algorithm 2.

Algorithm 2 Generalized Test Case

Input: Application Service α_i, Data Source δ_i
Output: Success or Fail

1: Get a *testDataSetT* composed by all data instance of Type T from δ_i;
2: **for each** *testDataT* $\in$ *testDataSetT* **do**
3: Calculate *expectedValueT* via plain logic;
4: *actualResultT* $\leftarrow \alpha_i(testDataT)$;
5: **if** Assertion comparing (*expectedValueT*, *actualResultT*) is NOT OK **then**
6: Return FAIL;
7: **end if**
8: **end for**
9: Return SUCCESS;

Essentially, the differences between the two approaches are related to the two aforementioned improvements:

- Test data instances no more are built in test case code. They are obtained from a provided data source. It is important to notice that any data instance of the specific type (T) will be used as input data test (line 1).

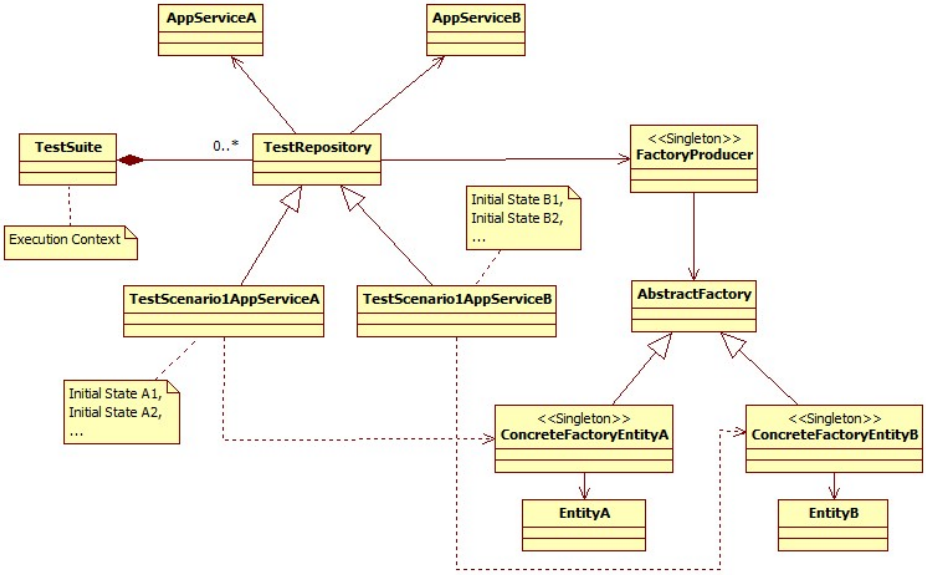

Figure 4: Generalized Functional Testings Architecture

- The expected values used in assertion comparison are not defined. Instead of this, they are calculated via a plain logic implementation (line 3).

- Instead of just one assertion comparison, there are several comparisons according to the number of data instances obtained from the data source (line 5).

Naturally, it is common to have application services that implement business rules which are applied to the entire set of data instances rather than to instance by instance as showed by Algorithm 2. An example involves calculating some measurement based on an analysis of all data instances made available. In this case, the Algorithm 2 must be slightly modified. The assertion comparison (Line 5) should be moved to outside the loop. Additionally, the *actualResult* should accumulate the measurement of interest analyzing each test data instance (Line 4). The application service will be invoked just once and the generated output should be compared against the calculated expected value (*actualResult*) outside the loop.

Based on these test improvements, there are several ways to design an architecture which supports generalized test cases. Figure 4 describes a suggestion of class diagram for this functional testing approach. It constitutes an implementation of the model presented by Figure 3.

It is important to identify how main components of the model are represented on this architecture:

- **Initial State:** Each one is associated with the appropriate Test Scenario. In practice, an initial state can be a SQL file, a class method or any other structure which must contain all test data definitions.

- **Test Engine** and **Execution Context:** The *TestSuite* class is the connector with the Test Engine. It is responsible to execute the whole architecture. Another responsibility of it is to ensure the prerequisites for the correct execution of all test cases (Execution Context).

- **Test Scenarios:** In the figure, there are two Test Scenarios - *TestScenarioAppServiceA* and *TestScenarioAppServiceB*. Each one is responsible to establish the appropriated conditions for its specific set of test cases. Moreover, any no generalized test case should be defined in the respective Test Scenario.

- **Test Repository:** It is implemented by the *TestRespository* class. All the generalized test cases are defined in this class.

Another important feature of this architecture is related to the data source of test instances. As there is no data defined in test cases, data instances are provided by concrete factories. Each test scenario uses a specific factory in order to make available data instances to Test Repository. A strategy used to coordinate the construction of factories is applying the Abstract Factory design pattern which provides an interface for creating families of related or dependent objects without specifying their concrete classes (FREEMAN et al., 2004). There is no more of one instance per concrete factory. This characterizes the use of the Singleton design pattern.

4.1 A Concrete Example

Let R_1 and R_2 be business rules implemented by an application service Ω such that

R_1: receives the Employer Identification Number (CNPJ, in its Portuguese acronym), with no validation digit, and it returns the same number added to the respective calculated validation digit.

R_2: receives as parameters the amount of admitted and layoff employees from a company, and it returns the respective employee balance (the difference between admitted ones and layoff ones). A negative balance implies that a zero value is returned.

The application service Ω should extract data from a source database, and it must store the output data into another target database. Suppose those databases are described by two very simplified schemes as presented by Tables 1 and 2.

Table 1: Source Database

cnpj_no_vd	admitted	layoff

For the R_1 and R_2 rules of this example, the set $\partial = \{\alpha, \beta\}$ is such that

$$\alpha = \{\text{cnpj\_no\_vd}\}$$

Table 2: Target Database

cnpj_vd	employee_balance

and

$$\beta = \{\text{num\_admitted, num\_layoff}\}$$

Let T_1 and T_2 be the set of test cases for R_1 and R_2 respectively. It is important to point out that, according to the approach presented in this work, R_1 and R_2 are faced as system functionalities. Thereby T_1 and T_2 constitute functional tests which use the Ω application service as a black-box.

Assuming null values are not permitted for any attribute of the scheme, so

$$\begin{aligned} \alpha' &= \{863832180001, 364657490001\} \text{ and} \\ \beta' &= \{(10,5),(10,10),(5,10)\}. \end{aligned}$$

As T_1 and T_2 must be executed, it is necessary that instances of ∂ should be fully loaded. Therefore, it will be loaded the following data instances to α' and to β' respectively

$$\begin{aligned} \partial_1^1 &= \{863832180001, 2, 0\}; \\ \partial_1^2 &= \{364657490001, 0, 1\}; \end{aligned}$$

$$\begin{aligned} \partial_2^1 &= \{407994500001, 10, 5\}; \\ \partial_2^2 &= \{125942160001, 10, 10\}; \\ \partial_2^3 &= \{573964610001, 5, 10\} \end{aligned}$$

The instances of $\partial' \subset \partial$ must be stored as described by Tables 3 and 4.

Table 3: Input Data to T_1

cnpj_no_vd	admitted	layoff
863832180001	2	0
364657490001	0	1

Table 4: Input Data to T_2

cnpj_no_vd	admitted	layoff
407994500001	**10**	**5**
125942160001	**10**	**10**
573964610001	**5**	**10**

The data highlighted in bold corresponds to data that will be used by the respective tests.

Other data are only loaded as required by the scheme. In this way, if T_1 would can use the input data loaded to T_2, and vice versa, then both tests will be favored by a higher input data diversity. Hence, R_1 and R_2 would be more widely tested and the Ω application service quality would be even more improved.

Aiming to reach the aforementioned effect, the tests must be projected in such a way that any data loaded into ∂ can be used by any other rule of R every time is possible. In the context of this example and considering the diagram presented in Figure 3, two scenarios should be created for testing Ω. In this way, Scenario 1 must refer the *Initial State* in order to load the source database. This initial state can be implemented as a SQL file, and its content can be described by queries as follow

```
INSERT INTO table_source_db
(cnpj_no_vd , admitted , layoff)
VALUES (863832180001, 2, 0);
INSERT INTO table_source_db
(cnpj_no_vd , admitted , layoff)
VALUES (364657490001, 0, 1);
```

As the aim is to generalize the tests and do not to restrict them, each test case must be able to read all CNPJ made available in the source database, apply the respective business rule, and verify if the output value is in the data set loaded into the target database. In other words, the test case T_1 related to the rule R_1 must implement the following steps

1. Read all CNPJ from the source database.
2. For each read CNPJ:
   (a) Invoke the validation digit calculus.
   (b) Build the CNPJ added by the validation digit.
   (c) Test if the expected CNPJ is in the target database.
3. Test if the amount of CNPJ in the source database corresponds to the amount of CNPJ in the target database.

The test case is implemented as an extension of *TestRepository* class. Because of this, the execution of any scenario will imply that all test cases be executed for all data set in the source database. It is important to point out that this feature was made possible because no specific CNPJ was used as parameter. In other words, this test case was generalized.

In an analogous way, the test case T_2 must be implemented. As soon as the test building be concluded, all CNPJ loaded to T_2 will also be used by T_1. Then, the data instances used to test R_1 will be expanded. Inversely, the data related to the amount of admitted and layoff persons made available by T_1 will also be used by T_2 in order to execute the tests to R_2.

4.1.1 Implementation Details

The implementation of this concrete example was made employing the Java programming language, version 1.8.0_31. The complete source code is available on a GitHub repository (RODRIGUES, 2017). Moreover, as the JUnit platform became the de facto standard framework for developing unit tests in Java (MASSOL; HUSTED, 2003), it was employed as Test Engine, and the Test Suite was defined as a JUnit Runner[2]. Relying on this platform, both T_1 and T_2 were constructed as two distinct Test Scenarios, namely $TestScenarioValidationDigit$ and $TestScenarioEmployeeBalance$, respectively.

The databases were emulated through two classes: $SourceDatabase$ and $TargetDatabase$. Because of this, the Initial State associated to each Test Scenario was built providing instances of $\partial' \subset \partial$ by means of Java primitive wrapper classes[3]

[2]According to (APPEL, 2015), the purpose of a JUnit Runner is to compose several test cases and/or other suites into a single entity that is processable by JUnit. The composition is accomplished by means of the $@SuiteClasses$ annotation, which is used to specify a list of test cases or nested suites.

[3]Each Java primitive data type has a class dedicated to it. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class.

rather than SQL files. So, the input data for the Test Scenario T_1 was made available by the class $InitialStateValidationDigit$ as follow

```
this.inputData =
    new HashMap<String, List<Long>>();
this.inputData.put("863832180001",
    Arrays.asList(2L, 0L));
this.inputData.put("364657490001",
    Arrays.asList(0L, 1L));
```

Two test cases were generalized and incorporated to the Test Repository. The first one checks if the balance calculated by the Ω application service is correct. The second one evaluates the validation digit generated by Ω. In this way, the steps described in the previous section and which must be executed by T_1 were implemented as follow

```
@Test
public void testCNPJValidationDigit(){

  // Read all CNPJ from the source DB
  Map<String, List<Long>> data =
      SourceDatabase.getInstance().
      getAllCNPJData();

  // For each read CNPJ
  for (String sCnpj : data.keySet()) {
    CNPJ cnpj = CNPJFactory.
        getInstance().
        build(sCnpj, data.get(sCnpj));

    // Validation digit calculus
    this.employerService.
        generateBalance(cnpj);

    // Plain Logic of Validation Digit
    String expectedCNPJ =
        addCNPJValidationDigit(
            cnpj.getValueNoDV());

    // Test if the expected CNPJ is
    // in the target database
    assertTrue(TargetDatabase.
        existsCNPJonBalanceTable(
            expectedCNPJ));
  }

  // Test the amount of CNPJ in the
  // source/target databases
    assertEquals(data.size(),
        TargetDatabase.
        selectAllCNPJ().size());
}
```

4.2 A Real Case Report

According to the Ministry of Labor (MTE, in its Portuguese acronym) (MTE, 2017), the General Record of Employed and Unemployed Persons (CAGED, in its Portuguese acronym) constitutes a permanent record of employed and unemployed persons supervised by Brazilian worker laws. This general record is used as a resource for the preparation of studies, researches, projects, and programs related to the job market and government decisions. Every Brazilian company must send CAGED files to the authorities pointing out each new admission or layoff. All these files are received, processed according to a large number of business rules, and the output is stored in a database. The aim is to consolidate the processed information in order to feed a strategic Data Warehouse.

The CAGED files processing cycle constitutes a project comprising the analysis of about 300,000 files per month. This workload generates a data volume of approximately 2 gigabytes. The amount and the complexity of the business rules involved in this process demanded the development of a specific software solution. Since a very tailored architecture for tests was necessary, the model presented in this work was adopted.

At the end of the software development, the effort dedicated to the tests activity resulted in:

- 51 implemented test scenarios
- 1,196 generalized functional tests

This is the test infrastructure which supports the quality of the CAGED files processing for all Brazilian companies.

5 Conclusions and Future Work

The presented model proposes an alternative approach for testing projects of applications services. Its essential characteristic is to improve the quality of the test activity. In this way, its adoption may lead to relevant benefits like Data Completeness and Code Coverage. The data completeness comes from the

data instances used by generalized test cases. Actually, every generalized test case uses the whole dataset made available by a provided data source. Moreover, as each data instance may be used by any generalized test case, there is a wider coverage of the code under testing. Actually, some unexpected behaviors can be detected through the use of data instances not initially prepared for a specific test case. Then, there is a high data-reuse which in turn promotes a better code coverage.

Another important benefit from this model is related to the improvement of test cases. As expected values are calculated rather than defined, it is necessary to implement some business logic in the test case. However, as business rules should be re-implemented as plain logic, they are easier to evaluate and hence, the test case becomes more informative. Furthermore, this simpler logic implementation facilitates the test code checking. As a result, there is a better quality assurance.

Naturally, there are other commons advantages of using this model. The test automation is simplified since the model is focused on functional testings and an application service can be tested as a black-box. Another benefit is the regression obtained as a straightforward result from the generic structure of test cases. All of these features mean a relevant support to several software (application service) troubleshooting tasks.

As an additional aspect, the model presented by this work aims to grant to a testing project an iterative and incremental design approach. In fact, as each application service under testing is sliced in test scenarios, each slice can be seen as an isolated testable piece. So, as soon as a new feature be incorporated into the application service under testing, a new test scenario (new slice) may be designed for it and its generalized test case can be added to Test Repository. In the same way, when there is a new application service in the Service Layer, all same structures may be incrementally generated in a new testing iteration.

Finally, it is relevant to mention that this paper proposes an architecture specification for the generalized functional testings model. However, its adoption as design for a testing project implies that the whole structure presented in Figure 4 must be implemented and customized for the project reality. Therefore, it is a future work to build the proposed testing architecture as a framework which may be just coupled into a project. This will facilitate the use of the presented approach and will preserve the focus on testing build activity.

References

ALUR, D.; MALKS, D.; CRUPI, J. *Core J2EE Patterns: Best Practices and Design Strategies*. 2nd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013. ISBN 0133807460, 9780133807462.

APPEL, F. *Testing with Junit*. [S.l.]: Packt Publishing, 2015. ISBN 1782166602, 9781782166603.

BARBOSA, E. F. et al. *Introdução ao Teste de Software*. João Pessoa, PB: [s.n.], 2000. 330–378 p. Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software (SBES 2000).

BOARD, I. S. T. Q. *Standard glossary of terms used in Software Testing*. 2012.

FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. 133-137 p. ISBN 0321127420.

FREEMAN, E. et al. *Head First Design Patterns*. [S.l.]: O' Reilly & Associates, Inc., 2004. ISBN 0596007124.

KANER, C. *An Introduction to Scenario Testing*. 150 W. University Blvd. Melbourne, FL 32901, USA, 2003.

KANER, C. What is a good test case? In: *Software Testing Analysid & Review Conference - STAR East*. Orlando, FL, USA: [s.n.], 2003.

LIMAYE, M. G. *Software Testing*. [S.l.]: Tata McGraw-Hill Education, 2009. ISBN 978-0-07-013990-9.

LUO, L. *Software Testing Techniques Technology Maturation and Research Strategy*. Pittsburgh, PA 15232, USA, 2001.

MASSOL, V.; HUSTED, T. *JUnit in Action*. Greenwich, CT, USA: Manning Publications Co., 2003. ISBN 1930110995.

MTE. *Cadastro Geral de Empregados e Desempregados [General Record of Employed and Unemployed Persons]*. 2017. <http://trabalho.gov.br/trabalhador-caged>, Accessed: 21-09-2017.

MYERS, G. J.; SANDLER, C. *The Art of Software Testing*. [S.l.]: John Wiley & Sons, 2004. ISBN 0471469122.

RODRIGUES, T. N. *tnas/gft-ria-sample: gft-ria-2017*. 2017. Disponível em: <https://doi.org/10.5281/zenodo.1095136>.

SAINI, G.; RAI, K. Software testing techniques for test cases generation. *International Journal of Advanced Research in Computer Science and Software Engineering*, v. 3, n. 9, p. 261–265, September 2013. Full text available.

STÄBER, F. *Service layer components for decentralized applications.* 1-182 p. Tese (Doutorado) — Clausthal University of Technology, 2008. Http://d-nb.info/992573637. Disponível em: <http://www.gbv.de/dms/clausthal/E_DISS/2009/db109198.pdf>.

TEAM, M. P. . P. *Microsoft Application Architecture Guide (Patterns & Practices)*. Microsoft Press, 2009. ISBN 9780735627109. Disponível em: <http://www.amazon.com/gp/product/073562710X>.