UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Rodrigo Barcelos da Silva

AVALIAÇÃO DA AMPLIAÇÃO DO ABAP *TEST DOUBLE FRAMEWORK* (ATDF) NUM DESENVOLVIMENTO REAL DE SOFTWARE

São Leopoldo

2019

UNIVERSIDADE DO VALE DO RIO DOS SINOS – UNISINOS

UNIDADE ACADÊMICA DE PESQUISA E PÓS-GRADUAÇÃO

ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE

Rodrigo Barcelos da Silva

AVALIAÇÃO DA AMPLIAÇÃO DO ABAP *TEST DOUBLE FRAMEWORK* (ATDF) NUM DESENVOLVIMENTO REAL DE SOFTWARE

Trabalho de Conclusão de Curso apresentado como requisito parcial para a obtenção do título de Especialista em Engenharia de Software, pelo curso de Pós-Graduação Lato Sensu em Engenharia de Software da Universidade do Vale do Rio dos Sinos – UNISINOS.
Orientador: Prof. Dra. Josiane Brietzke Porto

São Leopoldo

2019

Avaliação da Ampliação do ABAP *Test Double Framework* (ATDF) num Desenvolvimento Real de Software

Rodrigo Barcelos da Silva[1]

[1]Unidade Acadêmica de Pesquisa e Pós-Graduação – Universidade do Vale do Rio dos Sinos (UNISINOS) - São Leopoldo - RS - Brasil

barcelos.silva.rodrigo@outlook.com

Abstract. *The software unit tests are fundamental to ensure the quality and they can be realized by mean of tool, with the objective of testing program components, as methods or objects class. This study had as objective the enhancement evaluation of a framework, that realizes the mock of objects to non-final classes, created after programming language ABAP, developed in an anterior study involving software tests in a controlled and simulated environment. In a format of action-search it is analyzed if the development realized attended a real scenery of software development. The results show a confidant increase on development team to make refactory methods, tests without need to create scenarios and decrease errors in manual tests.*

Resumo. *Os testes unitários de softwares são fundamentais para garantir a qualidade e podem ser realizados por meio de ferramentas, com o objetivo de testar componentes de programas, como métodos ou classes de objeto. Este artigo teve como objetivo avaliar a ampliação de uma ferramenta, que realiza mock de objetos para classes não finais, criados na linguagem de programação ABAP, desenvolvida em um estudo anterior envolvendo teste de software num ambiente controlado e simulado. Em um formato de pesquisa-ação é analisado se o desenvolvimento realizado atende um cenário real de desenvolvimento de software. Os resultados mostraram um aumento de confiança no time para realizar refactory de métodos, testes sem a necessidade de montar cenários e a diminuição de erros em testes manuais.*

1. Introdução

Na implantação de um *Enterprise Resource Planning* (ERP) que tem como objetivo integrar todos os processos de uma empresa, geralmente, surge a necessidade de criar novos programas ou ampliar existentes para que atenda aos processos específicos de uma dada organização, que não estão presentes no *core* deste sistema.

Por sua vez, no processo de desenvolvimento desse software e de suas customizações, o programador gasta mais tempo depurando do que escrevendo código, devido à dificuldade de encontrar uma falha e, normalmente, resolve em um tempo melhor do que se leva para achá-la (FOWLER, 2004).

Com o surgimento e a adoção de métodos ágeis no desenvolvimento de software há uma valorização em indivíduos e iteração mais que processos e ferramentas, software em funcionamento mais que documentação abrangente, colaboração com o cliente mais que

negociação de contratos e responder a mudanças mais que seguir o plano. (BECK, BEEDLE, *et al.*, 2001).

Em métodos ágeis, desenvolvimentos guiados por teste (do inglês, *Test Driven Development* - TDD) foram iniciados pela metodologia *Extreme Program* (XP), com objetivo de garantir a qualidade dos códigos produzidos. Dessa forma segue-se o modelo Planejamento, Executar, Checagem e Ação (PDCA), com o propósito de garantir a aceitabilidade e a refatoração de trechos de código utilizados pelas equipes de desenvolvimento software. (SANTANA, FERREIRA e ROCHA, 2016).

Estudos anteriores, no contexto de projetos de pequenas e médias empresas, onde desenvolvedores adotaram as práticas de teste unitário, como por exemplo, o TDD, se observam resultados satisfatórios, bem como uma maneira mais produtiva de desenvolver softwares. Em outros experimentos foi visto que essa técnica gera uma melhora na qualidade do código gerado. (SOMMERVILLE, 2011)

Nesse sentido, o projeto chamado nesse estudo de SAP, no qual será realizada a avaliação prática da ampliação do framework (ATDF), feita anteriormente por Filho (2017) adota o Scrum como metodologia de desenvolvimento de software, contendo três fases principais (SOARES, 2004): Pré-planejamento, Desenvolvimento e Pós-planejamento. Neste método de trabalho, as entregas são por interações de uma a quatro semanas, chamadas de sprints, resultando em uma funcionalidade de valor ao usuário (RISING e JANOFF, 2000).

Com a aderência de métodos ágeis nesse determinado projeto, boa parte dos desenvolvimentos são feitos de modo incremental e com isso surgem questionamentos de como garantir a qualidade dos códigos criados e uma entrega de qualidade. Nesse sentido, o teste de software é fundamental para garantir a qualidade, que pode ser feito por meio de um processo de execução de um sistema ou programa para encontrar erros (MYERS, 2012).

Tendo em vista o uso de uma ferramenta de software para realizar testes, os testes unitários de softwares podem ser adotados, com o objetivo de testar componentes de programas, como métodos ou classes de objeto, sendo os tipos mais simples, as funções individuais ou métodos (SOMMERVILLE, 2011).

Dessa forma, na necessidade de se realizar testes do menor componente software, como um método de uma classe, existe a necessidade de se apoiar e utilizar uma ferramenta para realizar *mock* de objetos. Em virtude disto, um estudo anterior permitiu a criação de *mock* de objetos para classes não finais, por meio de uma ampliação do *ABAP Test Double Framework* (ATDF). Nesse estudo anterior, a avaliação foi realizada em apenas um ambiente controlado e simulado, obtendo resultados de um exercício realizado, sendo considerados como limitados para fins de validação prática dessa ampliação. Com isso, um dos trabalhos futuros sugeridos foi a realização de experimento em um ambiente real de desenvolvimento de software (FILHO, 2017).

Essa pesquisa se justifica porque com esta nova funcionalidade do ATDF é possível testar os componentes criados em um contexto real de desenvolvimento de software, para atender os processos da empresa em questão e com isto garantir a qualidade dos seus desenvolvimentos. Sendo assim, pode ser representada e seu desenvolvimento é orientado pela seguinte questão de pesquisa: a ampliação do ATDF realizada em um estudo anterior atende a

um cenário de desenvolvimento real, como o projeto de implementação do ERP SAP de uma empresa no setor de varejo?

Considerando a contribuição que o *framework* ampliado pode trazer aos desenvolvimentos, este trabalho tem como objetivo principal avaliar a aplicação da ampliação do ATDF para automatizar a criação de *mock objects* para classes não finais, em um desenvolvimento real de software de uma solução de ERP na área de varejo. Para alcançar este propósito elenca-se os seguintes objetivos específicos: *(i)* identificar os requisitos para utilização da ampliação no contexto do estudo; *(ii)* capacitar o time de desenvolvimento para usar a ampliação na criação dos testes; *(iii)* analisar os benefícios e as limitações encontradas na utilização do *framework* com a ampliação; *(iv)* analisar comparativamente os resultados obtidos nessa pesquisa com o estudo anterior.

Esta pesquisa se limita a ser realizada dentro do contexto de um time de desenvolvimento da empresa desse estudo, constituído de três desenvolvedores. Além disto, não são previstas grandes alterações nos componentes criados no ambiente real e nem análises comparativas com outras ferramentas de *mock* de objetos existentes.

O artigo está estruturado em seis seções, além dessa introdutória. A segunda seção apresenta o referencial teórico. Na terceira seção é apresentado o método de pesquisa adotado. Na quarta seção apresenta-se a avaliação da ampliação da ferramenta realizada em um processo real de desenvolvimento de software. Na quinta seção é destinada as considerações finais.

2. Referencial Teórico

O conteúdo apresentado nesta seção é resultado de pesquisa bibliográfica realizada para contribuir no entendimento dos principais conceitos utilizados no desenvolvimento deste estudo.

2.1. Teste de software

O teste de software é uma atividade onde desenvolvedor e a equipe de teste tem como objetivo encontrar defeitos no software, com o intuito de garantir a qualidade do produto que está sendo entregue (HIRAMA, 2011).

Segundo Sommerville (2011), os testes de software podem ser classificados em teste de aceitação, teste de sistema, teste de integração e teste de unidade. Considerando de forma decrescente o nível de abstração, quanto mais próximo do código, menos abstrato o nível do teste (FILHO, 2017)

Esta pesquisa está relacionada ao teste de unidade, que tem por objetivo verificar de forma individual um componente de software, que pode ser um método, um procedimento, uma classe completa, um grupo de funções ou classes desde que tenha um tamanho pequeno a moderado (WAZLAWICK, 2013).

Como já mencionado acima, o teste de unidade tem por objetivo testar de forma individual um componente, onde dependendo da tecnologia pode ser usado *frameworks* automatização dos testes. Por exemplo o *JUnit* para o Java (JUNIT, 2002) e o *ABAP Test Double Framework* (ATDF), para o ABAP (MEYANA, 2015), para realizar *mock* de objetos.

Os *mock objetcts* são usados no teste de software e tem o intuito de simular as dependências, desta forma é possível limitar o teste somente em único componente. Estes objetos normalmente implementam a mesma interface de um objeto real, desta forma a unidade que está sendo testada acaba executando uma simulação do objeto no qual ela depende. (SHAIKH MOSTAFA, 2014).

2.2. ABAP Teste e Análises

O ABAP Teste e Análises é um conjunto de ferramentas para garantir a qualidade e a robustez do código ABAP (SAP SE, 2016). Dentro dessas ferramentas existe a análise da cobertura de código ABAP, o ABAP *Test Cockipt* (ATC) e o *ABAP Test Double Framework* (ATDF). A cobertura de código é uma medida usada no teste de software com o objetivo de garantir qualidade. Ela consiste em medir os módulos ou trechos de programas dentro de uma determinada operação na qual está sendo realizado o teste unitário (SAP SE, 2019c).

Na linguagem *Advanced Business Aplication Programming* (ABAP), que é usada pela SAP para desenvolvimento de aplicações em seus ambientes (SAP SE, 2016), a cobertura do código pode ser obtida em três medidas diferentes *Procedure coverage*, *Statement coverage* e *Branch coverage* (SAP SE, 2019c), a saber:

- *Procedure coverage*: Responsável por medir a proporção de rotinas, métodos e módulos que foram chamados durante o período de medição;
- *Statement coverage*: Responsável por medir a proporção de instruções de códigos fontes executadas durante o período de medição;
- *Branch coverage*: Responsável por medir a proporção de controles de estruturas ambas sendo avaliadas como verdadeiro ou falso durante o período de medição.

O ATC é um conjunto de ferramentas no qual faz verificações estáticas e execução do teste unitário nos objetos desenvolvidos. As empresas adotam a utilização destes artefatos para garantir que os componentes criados pelos programadores estão de acordo com as regras da companhia e não possuem erros de sintaxe. (SAP SE, 2019c). Já, o *ABAP Teste Double Framework* consiste em criar duplas de teste (*mock objects*). Um dos pré-requisitos para utilização dele é o uso de assinaturas públicas, definidas através de uma interface global (HARDY, 2016).

O desenvolvedor utilizando o ATDF não precisa criar classes testes duplas para criar o teste unitário. Com a API, o objeto é criado em tempo de execução de forma automática, possibilitando configurações como valores retorno, exportação, parâmetros de alteração e chamada de exceções ou eventos. O *framework* também possibilita verificar o número de iterações com parâmetros específicos de entrada (MEYANA, 2015).

Conforme Hardy (2016), para um bom *design* é recomendado que toda classe tenha sua assinatura publica, definida por uma interface remetendo ao *design pattern decorator.* Pensado neste princípio, o *ABAP Teste Double Framework* (ATDF) funciona somente com uso de uma interface. A metodologia utilizada consiste em criar uma classe em tempo de execução, implementando uma interface global, que foi utilizada na classe real, adicionando código aos

métodos que possibilitam simular um determinado comportamento, com base nos parâmetros de entrada (HARDY, 2016).

2.3. Ampliação do *ABAP Teste Double Framework* (ATDF)

Conforme mencionado por Hardy (2016), para utilização do *framework* (ATDF) é necessário o uso de uma interface e toda a assinatura publica deve ser definida nela. Com ampliação anterior realizada não há mais esta necessidade, desde que o objeto a seja não final e, além disto, se foi mantida a arquitetura original do ATDF, desta forma não foi anulada nenhuma funcionalidade do desenvolvimento original (FILHO, 2017).

Nessa ampliação foram criados e adaptados alguns objetos utilizados pelo *framework* padrão, os mesmos estão contidos dentro do pacote SABP_UNIT_TEST_DOUBLE. Desta forma foi necessário a criação de um novo pacote chamado ZATDF e adicionado os objetos customizados nele. (FILHO, 2017)

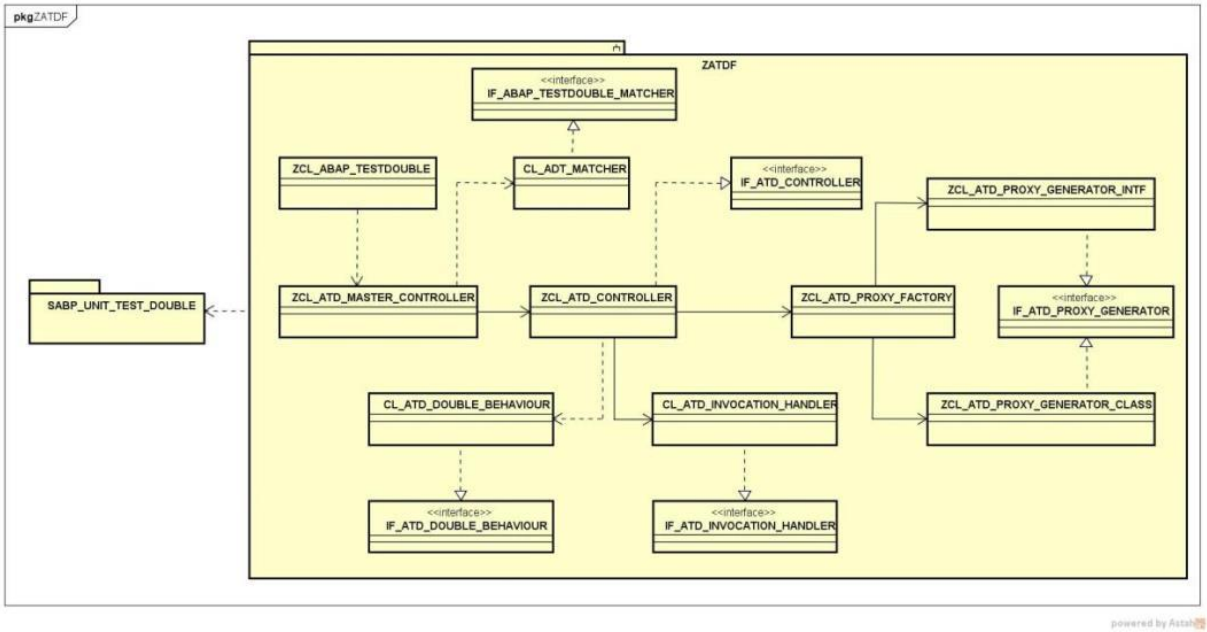


Figura 1 – Diagrama de Classes da Ampliação do ATDF

Fonte: Filho (2017, p. 20)

Na Figura 1 estão os objetos customizados por Filho (2017). Eles são os que iniciam com a letra "Z" e os demais são próprios do *framework* (ATDF), que continuam a ser utilizados na íntegra nessa pesquisa.

2.4. Testes Ágeis

Agile Testing é uma prática que segue os princípios ágeis para criar software com uma melhor qualidade, agrupando técnicas conhecidas que se complementam e evitam não conformidades nas entregas (COLE, 2015). A Figura 2 mostra as práticas de testes relacionadas de acordo com o contexto da aplicação.

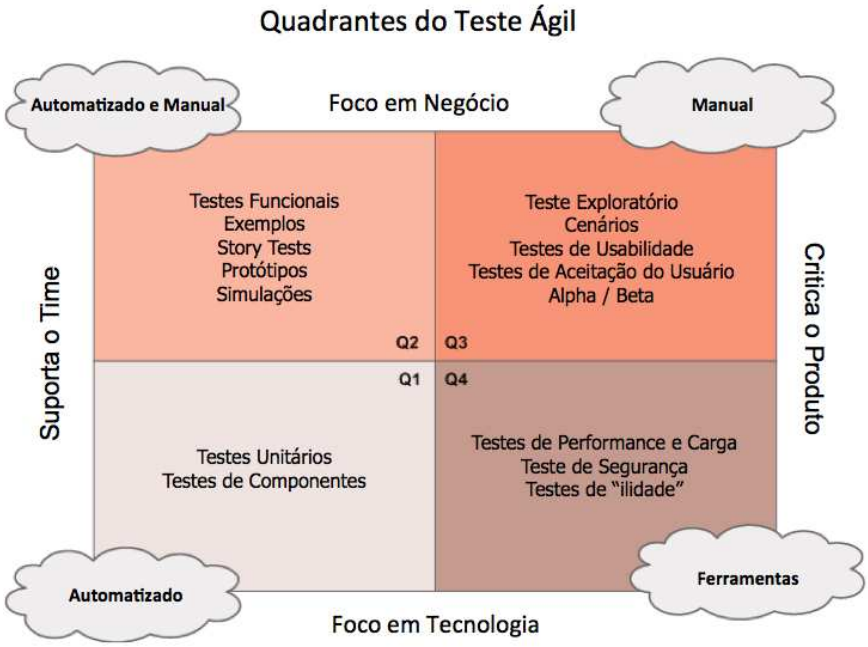


Figura 2 – Quadrantes do Teste Ágil

Fonte: Nogueira (2013)

Na Figura 2, em um dos eixos estão as técnicas que suportam o time e criticam o produto. Já, no outro eixo, os que são voltados ao negócio e tecnologia. Os quadrantes à esquerda, Q1 e Q2 auxiliam o time de desenvolvimento na solução e os testes estão relacionados à especificação de requisitos e à definição da solução (CRISPIN e GREGORY, 2009). Nos quadrantes Q3 e Q4, os testes para garantir que a solução ou produto desenvolvido foi realizado de forma correta, atendendo aos requisitos funcionais e não funcionais (CRISPIN e GREGORY, 2009).

O quadrante Q1 é o mais relevante para esse estudo. Nesse, temos como uma das técnicas principais é o uso do TDD, onde o desenvolvedor cria o teste unitário e depois cria a funcionalidade para satisfazer o mesmo. Essa técnica possibilita o desenvolvedor refletir sobre a modelagem, antes de escrever o código funcional e possibilita obter um código fonte bem testado (BECK, 2002).

Entre os métodos ágeis existem duas conhecidas e relevantes para este trabalho: *Extreme Programming* (XP) e *Scrum*. Entre elas, a que mais enfatiza o uso de testes é a XP, com a técnica de desenvolvimento *Test Driving Development,* que consiste em criar os testes antes do desenvolvimento (WELLS, 2013).

O TDD pode ser combinado com o método *Scrum* num contexto de desenvolvimento de software, com o objetivo de melhorar a qualidade do desenvolvimento dos programas. A aplicação desta prática pode ser adaptada de acordo com a disposição da equipe, desde que traga um melhor conforto ao time e os resultados sejam satisfatórios (SANTANA, FERREIRA e ROCHA, 2016).

2.5. Trabalhos Relacionados

Wilson (2016) realizou um estudo para analisar os efeitos do TDD, comparando com *Test Last Development* (TLD) na qualidade externa e interna no desenvolvimento de software e também a sua produtividade. A metodologia do estudo utiliza uma revisão sistemática da literatura considerando artigos publicados entre 1999 e 2014.

Em relação aos resultados de Wilson (2016), 57% dos estudos analisados foram validados por experimento e 32% foram por meio de estudo de caso. Na análise, 76% dos estudos mostraram um aumento na qualidade interna do software e 88% mostraram um aumento significativo na qualidade externa do software. Em relação à produtividade no ambiente acadêmico, o TDD apresentou uma evolução, porém, no cenário industrial ocorreu diminuição, comparado com o TLD. Em um apanhado geral, 44% dos estudos indica um baixo rendimento do TDD em relação ao TLD. Wilson (2016) relatou que o TDD tem maiores benefícios em relação à qualidade do software tanto interna quanto externa, entretanto, no que tange à produtividade, o TLD aparenta ser melhor.

Em outro estudo, Pagotto, Fabri e José (2016) realizaram uma pesquisa sobre a adoção da metodologia Scrum no desenvolvimento de software individual, tendo como objetivo apresentar um processo que possa ser usado por desenvolvedores. Nesse estudo foi criada uma customização da metodologia Scrum, onde foi utilizado o conceito de entrega incremental presente neste processo e práticas do método *Personal Software Process* (PSP), que apresenta como deve ser feito software de qualidade. A validação deste experimento foi realizada primeiramente no ambiente academico e depois, externado para área industrial.

Como resultado teve 55 alunos de Engenharia da Computação e Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná utilizando o Scrum Solo, nos anos de 2012, 2013 e 2014 e obtiveram sucesso no desenvolvimento de seus produtos. Um mapeamento realizado pelos autores mostrou que 8 ex-alunos de um dos cursos mencionados tiveram resultados satisfatorios, na utilização da customização para desenvolvimento de software.

Os trabalhos citados acima se relacionam com este estudo. O primeiro com a pesquisa envolvendo o uso de teste unitário de software e o outro com a utilização da metodologia *Scrum,* que também é presente no ambiente que é realizado o experimento. O Quadro 1 mostra uma análise comparativa entre esses trabalhos relacionados e a presente pesquisa.

Quadro 1 – Análise comparativa entre os trabalhos relacionados

Autor	Tecnologias	Método de pesquisa	Contexto	Objetivo	Resultados
(WILSON BISSI, 2016)	Máquina de busca	Revisão sistematica da literatura	Acadêmico e industrial	Analisar os efeitos do TDD comparando com *Test Last Development* (TLD), na qualidade externa e interna no desenvolvimento de software, bem como a sua produtividade	O TDD no ambiente acadêmico apresentou ser mais efetivo tanto na produtividade quanto na qualidade de software. Entretanto, na área industrial, o TLD obteve uma melhor produtividade.
(PAGOTTO, FABRI e JOSÉ, 2016)	*Astah*	Pesquisa-ação	Acadêmico e Industrial	Descrever o *Scrum Solo* e apresentar sua aplicação no desenvolvimento de parte de um pequeno projeto de software	O processo obteve sucesso em ambos os amientes. Do ponto de vista dos participantes, o Scrum Solo atendeu as expectativas inerentes a gestão de projetos do cliente e do desenvolvedor
(SILVA, 2019)	- ABAP - ABAP *Test Development Framework* - *Framework* ATDF ampliado por Filho (2017)	Pesquisa-ação	Solução de ERP na área de Varejo	Avaliar a ampliação do *Abap Test Double Framework* (ATDF) para automatizar a criação de *mock objects* para classes não finais, em um desenvolvimento real de software	Foi possível avaliar o uso da ferramenta, por meio de objetos testados em parte das funcionalidades da solução de ERP.Os resultados foram satisfatórios em realção ao aumento de confiança do time na realização de *refactory* de código, testes sem necessidade de montar cenários e uma diminuição nos erros em testes manuais

Fonte: Elaborado pelo autor

3. Metodologia da Pesquisa

Tendo em vista a realização de uma avaliação de um sistema é preciso pôr em contraste dois conceitos, a verificação visando se o sistema atende o que foi proposto e a validação para determinar o quão bem o programa resolve o problema (WAINER, 2007).

Na pesquisa anterior fica claro que a ampliação atende ao que foi especificado, porém, há uma limitação quanto ao experimento dentro de um cenário controlado. Devido a isto há a necessidade de aprofundamento em relação à verificação e validação desse desenvolvimento.

3.1. Delineamento da pesquisa

Quanto ao ponto de vista da natureza da pesquisa é classificada como aplicada devido ao envolvimento de uma aplicação prática em um ambiente real de desenvolvimento de software (AZEVEDO, MACHADO e SILVA, 2011). Em relação ao enfoque principal será adotado pesquisa qualitativa na forma de pesquisa-ação, modificando o ambiente que está sendo estudado e realizando iterações com a organização (WAINER, 2007).

O estudo é classificado como exploratório, por se tratar de um aprofundamento na avaliação de um componente de software ampliado em um estudo anterior. Em relação ao tempo pode-se enquadrar como corte-transversal, pois a análise será realizada em um projeto de software que já possui um deadline definido (AZEVEDO, MACHADO e SILVA, 2011).

3.2. Unidade de Análise

A pesquisa ocorreu dentro de uma empresa do mercado de varejo, localizada em Porto Alegre, fundada em 1935. Por decisão da área de governança do projeto, a empresa não terá o nome divulgado nesse estudo. Essa área está realizando a implantação do ERP SAP, onde envolve a troca do sistema legado para um novo. Alguns desenvolvimentos da implantação são realizados por quatro times, que utilizam *Scrum,* totalizando 9.206 colaboradores. A avaliação da ferramenta foi feita dentro de um destes times, que é composto por três desenvolvedores, um *Scrum Master*, um *Product Owner* (PO) e um usuário desse sistema.

Em relação ao time de desenvolvimento envolvido nessa avaliação destaca-se que o escopo abrangeu desenvolvimentos realizados até o final do mês de Janeiro de 2019, sem prorrogação. Devido a estas circunstâncias, a coleta e análise de dados dessa pesquisa seguiu o mesmo prazo.

3.3 Coleta dos dados

Na coleta de dados foram adotadas as técnicas documental, utilizando códigos de funcionalidades do próprio ERP, bem como a de questionário, para obter a percepção dos participantes e a de observação participante, com objetivo de obter dados referente às limitações e erros apresentados pelo *framework* ampliado (LAKATOS, 2009).

Quanto às perguntas do questionário para a presente pesquisa, se adotou as mesmas do estudo anterior (Anexo A), onde o objetivo foi obter o nível de satisfação e a percepção dos participantes em relação ao uso da ampliação do ATDF (FILHO, 2017).

Devido à ferramenta em questão ser desenvolvida dentro do ERP, para a coleta de dados da presente pesquisa foram utilizadas as ferramentas de verificação e de cobertura, disponibilizadas pelo próprio sistema, sendo elas *ABAP Workbanch* e *ABAP Unit Browser* (SAP SE, 2016).

Alguns resultados obtidos por estas ferramentas são o retorno de teste unitário e o teste de cobertura do objeto. Dessa forma, se executado no pacote que contém os desenvolvimentos do projeto possibilitam o mapeamento das classes, que possuem testes unitários associados.

3.3 Análise de dados

A partir dos dados quantitativos, coletados com a utilização das ferramentas do SAP ERP foi feita uma análise matemática e estatística básica (AZEVEDO, MACHADO e SILVA, 2011), que abrangeram cálculos de: quantidade, frequências, medidas, entre outras. (AZEVEDO, MACHADO e SILVA, 2011)

Como resultados, ainda se teve os testes em si e o percentual de cobertura dos objetos. Além de uma busca nos testes unitários, para localizar quais destes testes utilizaram a ampliação do *framework* ATDF, desenvolvida no estudo anterior.

Quanto à análise dos dados qualitativos, oriundos da pesquisa documental e da percepção dos envolvidos, se adotou a técnica de análise de conteúdo (BARDIN, 1995).

4. Desenvolvimento e Análise dos Resultados

Nesta seção é apresentado o resultado da pesquisa-ação realizada em um projeto real de desenvolvimento de software, para atender os objetivos desse estudo. As subseções tratam da analise exploratória e ajustes para uso do *framework,* capacitação do time de desenvolvimento para uso da ampliação, avaliação da ampliação ATDF e análise comparativa com estudo anterior.

4.1. Análise Exploratória e Ajustes para Uso do *Framework*

No início do estudo foi preciso carregar a customização realizada no estudo anterior, no ambiente real do projeto atual e objeto desse estudo. Após isso, foi feita uma análise dos objetos já criados no projeto e foi verificado se os mesmos atendiam o pré-requisito da ferramenta de *mock* de objetos. Tal pré-requisito refere-se à utilização de classes não finais (FILHO, 2017).

Com o objetivo de aprofundar o conhecimento do *framework* ampliado no estudo anterior, na pesquisa adotou-se a estratégia de criação do mesmo exercício realizado em Filho (2017), com a adição de mais variações como por exemplo, o uso do método construtor, mais o uso de classes com métodos estáticos. Além disso foi realizado um teste adicionando o uso de uma classe com o *desing pattern singleton.*

O resultado do exercício realizado juntamente com a análise dos objetos criados no projeto oportunizou a identificação de três pontos de ajustes, na ampliação feita anteriormente. O primeiro foi o uso do método construtor, onde foi preciso alterar o código para que o mesmo não fosse considerado nas redefinições. O segundo foi o tratamento do *alias* ao usar uma interface na classe. E, o terceiro ajuste foi habilitar a realização de *mock* de operações presentes na mesma classe no qual este sendo feito o teste unitário.

Quanto aos ajustes necessários, o primeiro ajuste foi a não redefinição do método *constructor* ao gerar a classe em tempo de execução para criação do *mock objects,* evitando um erro na construção do objeto. A segunda alteração se justifica por não criar operações duplicadas para os métodos das interfaces. Como seus métodos e atributos podem possuir *alias,* o gerador da classe de execução criava duas vezes a mesma operação.

Para corrigir estes dois pontos identificados na análise exploratória foi preciso uma alteração na classe *ZCL_ATD_PROXY_GENERATOR_CLASS,* copiada do ATDF *standard,* onde foi preciso modificar o método *generate_class_definition* e *generate_class_implementation* responsáveis por criar o código fonte do objeto *mock*. A Figura 3 ilustra as alterações sublinhadas em vermelho, sendo mantido o padrão de codificação do autor anterior e ocultado trechos não alterados pelo pesquisador.

```
METHOD generate_class_definition.
"... Hide Code
  LOOP AT ls_class_descr-class_descr->methods INTO ls_method.
    IF  ls_method-visibility EQ cl_abap_objectdescr=>public  AND
        ls_method-is_class NE abap_true                      AND
        ls_method-name <> 'CONSTRUCTOR'                      AND
        ls_method-alias_for IS INITIAL.
      CONCATENATE `METHODS: ` ls_method-name ` REDEFINITION.` INTO temp.
      APPEND temp TO rt_def_code.
      CLEAR temp.
    ENDIF.
  ENDLOOP.
"... Hide Code
ENDMETHOD.
```

```
METHOD generate_class_implementation.
"... Hide Code
  LOOP AT it_class_descr INTO ls_class_descr.

    LOOP AT ls_class_descr-class_descr->methods INTO ls_method.
      IF ( ls_method-visibility EQ cl_abap_objectdescr=>public OR
           ls_method-visibility EQ cl_abap_objectdescr=>protected ) AND
           ls_method-is_class NE abap_true AND
           ls_method-name <> 'CONSTRUCTOR' AND
           ls_method-alias_for IS INITIAL.
        CONCATENATE `  method ` ls_method-name  `.`       "#EC NOTEXT
        INTO temp.
        APPEND temp TO rt_impl_code.
        CLEAR temp.
"... Hide Code
     ENDLOOP.
"... Hide Code
  ENDLOOP.
```

Figura 3 – Métodos ajustados para correção de erros

Fonte: Elaborado pelo autor

Através de uma análise em objetos já construídos em outros desenvolvimentos, se encontrou a necessidade de adicionar uma funcionalidade para realização de *mock* de dependências de operações presentes em mesma classe. Assim, realizou-se uma alteração no mesmo objeto citado acima, para incluir a possibilidade de chamar o método real ao invés do *mock,* quando necessário, através do nome do método e o *mock object.*

Nos casos em que o desenvolvedor precise criar um teste unitário, que esteja dentro do cenário acima, o mesmo precisará instanciar a classe como *mock object* e na criação do teste informar qual método será executado como real, conforme Figura 3.

```
METHOD prepare_cartazetes_to_display.
  zcl_abap_testdouble=>configure_call_method_as_real( double = cut method = 'prepare_cartazetes_to_display' ).

************************************************************
* Tabela de Cartazete deve estar vazia
************************************************************
    DATA lt_cartazetes TYPE zttpmr006.
    DATA lt_preco TYPE zttsd_matnr_price.
    DATA(ls_param) = VALUE zsde_param_010( ).

    zcl_abap_testdouble=>configure_call( cut )->returning( mo_zcl_sd_prices ).
    cut->get_zcl_sd_prices( ).

    zcl_abap_testdouble=>configure_call( mo_zcl_sd_prices )->returning( lt_preco ).
    mo_zcl_sd_prices->get_prices_pveac( ).

    DATA(lt_result) = cut->prepare_cartazetes_to_display( is_param = ls_param it_cartazetes = lt_cartazetes ).

    cl_aunit_assert=>assert_initial( act = lt_result msg = 'Cartazete is not empty' ).
ENDMETHOD.
```

Figura 4 – Exemplo de uso do *mock object* para criação do teste unitário como um todo

Fonte: Elaborado pelo autor

Devido ao objeto de teste ser do tipo *mock object* é preciso que seja dito quais métodos serão executados como reais, com exceção dos privados e estáticos. Conforme é possível ver no teste unitário presente na Figura 4, o método que está sendo testado é o *prepare_cartazetes_to_display,* onde utiliza o ZATDF em dois momentos. O primeiro *get_zcl_sd_prices* faz com que o retorne um objeto, que também foi criado utilizando o *framework*. O segundo, *get_prices_pveac,* que retorna uma tabela como resultado. Depois disso tem-se a execução da operação que se deseja testar e a validação do retorno com um componente do ABAP *Unit.*

Analisando o teste acima, a execução da operação é realizada partindo do método da classe *mock,* que foi gerada em tempo de execução herdando a classe real. Com a configuração, a execução do *prepare_cartazetes_to_display* será feita pelo objeto superior que contém o código real. A versão atualizada com as alterações supracitadas está disponibilizada no ambiente de desenvolvimento da empresa objeto desse estudo.

4.2. Capacitação para Uso da Ampliação

Com o objetivo de capacitar o time que participou do experimento foi definido que somente um dos integrantes usaria a ampliação em um primeiro *sprint*. Com o entendimento que a ferramenta está apta para uso, nos próximos desenvolvimentos, os objetos testados seriam apresentados aos demais sujeitos envolvidos, com objetivo de ter exemplos reais do uso da ferramenta.

Após o *sprint* experimental, não foi definido uma regra de criação dos testes unitários e o desenvolvedor ficou livre para escolher se desejava realizar o uso do TDD ou TLD, ou seja,

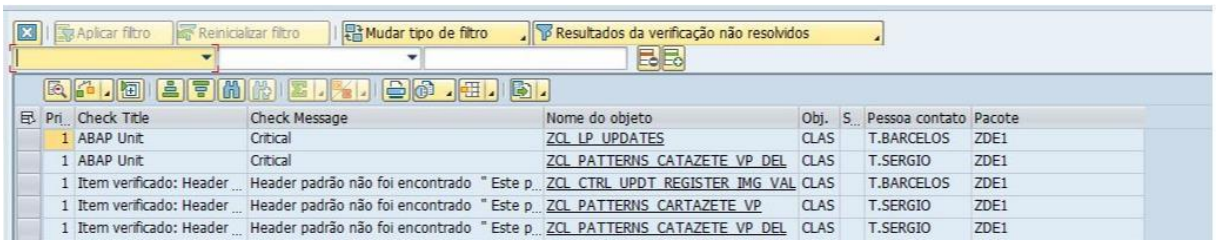
teste antes de do desenvolvimento ou depois. Tendo em vista que a customização manteve as mesmas funcionalidades foi criado um programa ABAP, que contém exemplos de teste unitários do ATDF, com base na publicação de Rossi (2018), que serviu para auxiliar na criação dos testes unitários, o resultado da execução desta ferramenta pode ser vista no apêndice C.

Utilizando o programa supracitado, o pesquisador realizou uma capacitação interna para fins de passagem do conhecimento adquirido na etapa anterior da pesquisa, aos desenvolvedores do time, levando em torno de 45 minutos de duração, em um local reservado para time de *Scrum*, na sede da organização objeto de estudo. Além disso foi passado o *link* do conteúdo criado por Rossi em (2018), para eventuais dúvidas técnicas durante a criação dos testes unitários.

4.3. Avaliação da Ampliação ATDF

O projeto real em que foi realizado esse estudo possui quatro frentes de desenvolvimento para atender processos estratégicos e em uma delas foi adotada como unidade de análise deste estudo. Através de um *Release Backlog* contendo aproximadamente vinte estórias, divididas para dois times de *Scrum* e onde somente um deles foi designado para realização do experimento da pesquisa.

Ambas as equipes de desenvolvimento ficaram com *sprints* de duas semanas. Antes de cada entrega era executado a ferramenta *ABAP Test Cockpit* (ATC) para que os objetos criados e modificados fossem devidamente verificados garantindo os padrões de codificação definidos pelo cliente. No projeto foi configurado que erros de testes unitário fossem considerados de alta prioridade, bloqueando o transporte para outros desenvolvimentos.



Pri...	Check Title	Check Message	Nome do objeto	Obj.	S...	Pessoa contato	Pacote
1	ABAP Unit	Critical	ZCL_LP_UPDATES	CLAS		T.BARCELOS	ZDE1
1	ABAP Unit	Critical	ZCL_PATTERNS_CATAZETE_VP_DEL	CLAS		T.SERGIO	ZDE1
1	Item verificado: Header ...	Header padrão não foi encontrado " Este p...	ZCL_CTRL_UPDT_REGISTER_IMG_VAL	CLAS		T.BARCELOS	ZDE1
1	Item verificado: Header ...	Header padrão não foi encontrado " Este p...	ZCL_PATTERNS_CARTAZETE_VP	CLAS		T.SERGIO	ZDE1
1	Item verificado: Header ...	Header padrão não foi encontrado " Este p...	ZCL_PATTERNS_CATAZETE_VP_DEL	CLAS		T.SERGIO	ZDE1

Figura 5 – Execução da ferramenta (ATC)

Fonte: Elaborado pelo autor

Na Figura 5 é possível analisar que erros de teste unitário são de alta prioridade, conforme conteúdo da coluna "*Check Message*". Nestes casos, o desenvolvedor precisa ajustar o teste contemplando a última alteração, antes de transportar o objeto alterado para outro ambiente. Entre as duas equipes, contando novos objetos e os modificados de *releases* anteriores tem-se um total de 54 aptos para criação de teste unitários, onde em 12 desses foram criados testes unitários.

A Figura 6 mostra um gráfico que ilustra o percentual de objetos atendido e é possível identificar que a quantidade de testes unitários realizados é baixa, considerando a *release* em análise. Porém, para algumas estórias de usuário, esse resultado foi satisfatório.

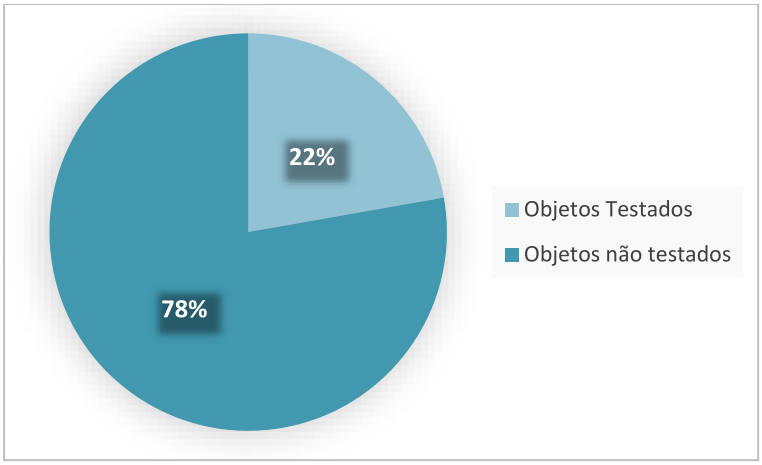


Figura 6 – Percentual de objetos testados

Fonte: Elaborado pelo autor

Diferente do estudo anterior, que coletou métricas com base em quantidade de linhas testadas, neste estudo foram coletados os dados estatísticos entregues pela ferramenta de teste do ABAP. Como já mencionado acima, essa ferramenta possui três medidas diferentes, sendo que para este estudo foi utilizada a *procedure covarage.* Na Tabela 1 é possível ver o resultado dessa medição.

Tabela 1 – *Procedure Covarage Unit Test*

Classes	Procedure Covarage s/filtro				Procedure Covarage c/filtro			
	Total	Executado	Não Executado	Percentual	Total	Executado	Não Executado	Percentual
ZCL_CARTAZETE	14	8	6	57.14%	9	8	1	88.98%
ZCL_CTRL_UPDT_REGISTER_IMG_VAL	10	4	6	40%	4	4	0	100%
ZCL_LABEL_PRINT	57	4	53	7.02%	42	4	38	9.52%
ZCL_LP_PRODUCT_TREE	63	6	57	9.52%	56	6	50	10.71%
ZCL_LP_UPDATES	59	36	23	61.02%	56	36	20	64.29%
ZCL_MM_STOCK_ASSORTMENT	12	5	7	41.67%	6	5	1	83.33%
ZCL_NON_APPL_LABELS_REPORT	42	10	32	23.81%	31	10	21	32.26%
ZCL_PATTERNS_CARTAZETE_VP	8	6	2	75.00%	7	6	1	85.71%
ZCL_PEDIDOSCLIENTE	13	12	1	92.31%	12	12	0	100.00%
ZCL_PRICE_TABLE	31	13	18	41.94%	23	13	10	56.52%
ZCL_PVEAC	6	4	2	66.67%	4	4	0	100.00%
ZCL_SPECIAL_DISCOUNT	5	3	2	60.00%	3	3	0	100.00%
Total	320	111	209	48.01%	253	111	142	69.28%

Fonte: Elaborado pelo autor

Ressalta-se que na coleta dos resultados foi preciso adicionar um filtro para desconsiderar métodos não aptos para criação do teste unitário, devido a esta questão existem medidas com e sem filtro, na Tabela 1. Os tipos de operações desconsideradas foram busca de dados em tabelas tanto customizadas quanto *standards*, contendo módulo de funções e com métodos não aptos para criação de *mock* de objetos.

A interpretação do conteúdo mostrado na Tabela 1, é a necessidade de separar essas operações não aptas para criação do teste unitário em outros objetos ou avaliar uma maneira de contemplá-las para evitar resultados falsos positivos na cobertura de código. Com esse resultado fica claro que os objetos testados não foram construídos pensando nos testes unitários. Isso de

certa forma já era esperado, em virtude que no projeto onde foi realizado o estudo não tem a prática de realizar esse tipo de teste

Com intuito de coletar as operações do *mock object* utilizadas foi realizada uma busca considerando um delta das 12 classes, no qual foi realizado o presente estudo. As funcionalidades não utilizadas foram omitidas do resultado, apresentado na Tabela 2.

Tabela 2 – Utilização das operações oferecidas pela ampliação ATDF

Classes	CONFIGURE_CALL	RETURNING	SET_PARAMETER	RAISE_EXCEPTION	TIMES	IGNORE_ALL_PARAMETERS	AND_EXPECT
ZCL_CARTAZETE	5	5	0	0	0	0	0
ZCL_CTRL_UPDT_REGISTER_IMG_VAL	0	0	0	0	0	0	0
ZCL_LABEL_PRINT	4	4	0	0	0	0	0
ZCL_LP_PRODUCT_TREE	0	0	0	0	0	0	0
ZCL_LP_UPDATES	60	44	3	0	0	13	0
ZCL_MM_STOCK_ASSORTMENT	0	0	0	0	0	0	0
ZCL_NON_APPL_LABELS_REPORT	5	4	2	0	0	0	0
ZCL_PATTERNS_CARTAZETE_VP	2	2	0	0	0	0	0
ZCL_PEDIDOSCLIENTE	3	1	1	1	0	0	0
ZCL_PRICE_TABLE	5	5	0	0	3	0	0
ZCL_PVEAC	6	5	0	0	0	0	3
ZCL_SPECIAL_DISCOUNT	7	5	2	0	0	0	0
Total	97	75	8	1	3	13	3

Fonte: Elaborado pelo autor

Conforme pode ser visto na Tabela 2, o método *configure_call* é o mais utilizado, dentro do conjunto de testes unitários para atribuir as configurações para chamada das operações, no qual foi realizado o *mock.* Através dela é acessado as demais funcionalidades como *return*, *set_parameter*, *raise_exception*, *times*, *ignore_all_parameters* e *and_expect*. Na Tabela 1 é possível analisar que há classes que não fizeram o uso do *framework*, na criação do teste unitário a *zcl_ctrl_updt_register_img_val* e a *zcl_mm_stock_assort*, entretanto, as mesmas não possuem uma porcentagem satisfatória de cobertura.

Destaca-se que o resultado da criação de teste unitário nos *sprints* teve um retorno esperado e satisfatório, no que tange a como realizar *refactory* de métodos, testes sem a necessidade de montar cenários e a diminuição de erros em testes manuais. O *framework* ampliado contribuiu para realização dos testes, lidando com as dependências e seu uso permitiu identificar algumas lições aprendidas e pontos de atenção em tempo de desenvolvimento.

Em alguns casos, para que fosse possível a utilização do ATDF foi preciso adequar o código desenvolvido, sendo eles evitar a criação de métodos privados ou estáticos, contendo código de funções *standard,* bem como a seleção de dados.

Na criação de classes utilizando *desing patterns singleton* deve-se considerar a construção do objeto como protegida, ao invés de privada e, quando realizar o uso, procurar utilizar um método *protegido* ou *público* no objeto que vai utiliza-lo. Dessa forma é possível injetar o *mock object* para lidar com as dependências. Além disso, quando declarado o construtor da classe, não se deve deixar os seus atributos de entrada obrigatórios, pois os mesmos não serão atribuídos ao gerar o objeto que vai lidar com as dependências.

4.4. Análise Comparativa com Estudo Anterior

Com o objetivo de ter uma análise comparativa com o estudo anterior, foi aplicado o mesmo questionário, contemplando as 10 perguntas de Filho (2017), com os sujeitos envolvidos na pesquisa.

Ressalta-se que existe uma diferença em relação à quantidade de respondestes, pois nessa pesquisa, o questionário foi aplicado num time de desenvolvimento, com 3 desenvolvedores apenas. Na realizada por Filho (2017) haviam 6 participantes. O preenchimento do questionário foi acompanhado pelo pesquisador, numa sala da organização objeto do estudo, no dia 14 de Fevereiro de 2019. A Tabela 3 mostra os resultados obtidos com a aplicação do questionário.

Tabela 3 – Análise dos resultados do questionário

Nº	**Afirmativa**	Silva (2019)	Filho (2017)
1	A criação de teste unitário além de garantir que o componente a ser testado atende o que foi solicitado, ajuda o desenvolvedor a escrever um código mais simples.	4	5
2	A grande dificuldade para a criação do teste unitário é a manipulação de objetos dependentes, que interferem no componente a ser testado.	5	4
3	**Criar *mock objects* sem uma ferramenta de apoio é uma atividade trabalhosa e quem consome bastante tempo.**	**5**	**5**
4	**A ferramenta criada para automatizar a criação de *mock objects* facilita a criação do teste unitário.**	**4**	**4**
5	A ferramenta criada para automatizar a criação de *mock objects* resulta em um ganho considerável de tempo na criação do teste unitário.	4	5
6	**Em relação a funcionalidade, a ferramenta dispõe de todos os recursos necessários para a criação de mock objects. falhas durante sua utilização**	**3**	**3**
7	Em relação a confiabilidade, a ferramenta não apresenta falhas durante sua utilização.	3	5
8	**Em relação usabilidade, a ferramenta é fácil de usar.**	**4**	**4**
9	Em relação a eficiência de desempenho, a ferramenta tem um tempo de execução adequado.	4	5
10	**Em relação a satisfação, a ferramenta da confiança e conforto para a criação de *mock objects*.**	**4**	**4**

Fonte: Elaborado pelo autor

Comparando os números apurados em relação ao nível de concordância (escala entre 1 e 5) os itens 3, 4, 6, 8 e 10 obtiveram o mesmo resultado da pesquisa realizada por Filho (2017). Com exceção da questão 7, as demais tiveram uma diferença de apenas 1 ponto, sendo que isto pode ser explicado pelos erros encontrados relacionados ao *constructor* e ao *alias* das interfaces nos primeiros testes realizados.

Conforme visto na Tabela 3 fica evidente a necessidade de haver uma ferramenta de apoio para criação de *mock objects* e que o ATDF ampliando no estudo anterior atende a esta necessidade de modo satisfatório.

Em relação aos itens divergentes da Tabela 3, na afirmativa de número 1, a ferramenta ajuda a implementar um código com mais qualidade, entretanto não descarta a revisão por pares realizada a após a criação do código fonte. No item 2, o time percebeu como uma grande dificuldade de realizar o teste unitário de um componente, o fato de lidar com suas dependências, pois conforme visto na Tabela 2, de 12 classes no total, somente 2 classes não utilizaram o *mock objects.*

Na Tabela 3, a afirmativa de número 5 obteve um número inferior a pesquisa anterior, devido aos participantes não terem experiência com a ferramenta e, em algumas vezes, o participante precisou recorrer ao programa de exemplo, gerado para auxiliar na criação dos testes unitários. Por fim, o item 9 apresentou uma pequena divergência, devido ao fato de que em algumas vezes a execução do teste unitário no objeto ser mais lenta, em relação as outras execuções.

5. Conclusão

Através dessa pesquisa foi possível avaliar num contexto real, que o *framework* ATDF ampliado em um estudo anterior atende as necessidades de lidar com as dependências na criação do teste unitário. Entretanto, foi visto que ainda há necessidade de lidar com certas limitações da ferramenta. Como não é possível criar *mock* para operações estáticas e métodos privados, a classe precisa ser não final e a classe não pode ter o seu construtor privado. Devido a essas questões encontradas durante a avaliação realizada, o desenvolvedor precisa criar o seu código pensando no teste, o que requer mudanças de certos vícios ou hábitos na etapa de codificação do processo de desenvolvimento de software.

A ferramenta utilizada neste estudo contribuiu para organização objeto do estudo, adicionando outra maneira de realizar *mock objects*, além do ATDF oferecido no *standard* do ERP. Também é possível lidar com as dependências entre classes, desde que elas estejam aptas a criação do *mock object*, sem a necessidade de criação de uma interface. Além disso, as classes para realização de *mock* são geradas de forma automática, resultando em um menor tempo de desenvolvimento e contribuindo ao time de desenvolvedores.

Com a utilização da ferramenta ampliada por Celso (2017), juntamente com os ajustes realizados pelo pesquisador no presente estudo foi possível aumentar o número de objetos testados e ter um aumento na cobertura de código testado, resultando em um aumento na qualidade do código e na confiança do time, para alterações em componentes já estáveis ou entregues para teste do usuário. Portanto, se considera que o objetivo geral da pesquisa de avaliar o uso da ferramenta num contexto real foi alcançado, porém, cabe a ressalva de que a quantidade de objetos testados não garantiu que fossem testadas todas as funcionalidades do sistema.

Essa é uma limitação encontrada na ferramenta ampliada anteriormente por Celso (2017), avaliada e adaptada na presente pesquisa, podendo ser explorada e evoluída no futuro, no contexto da organização objeto do estudo pelo próprio pesquisador ou por algum desenvolvedor envolvido nessa avaliação e, em outros contextos, por profissionais da

comunidade ABAP. Para tanto, conforme mencionado no item 4.3, é necessário considerar as limitações de seleção de dados, chamada de funções e objetos *standard* na elaboração da arquitetura da solução, sendo importante criar objetos separados para se tornarem aptos a realização do *mock objects.* As alterações realizadas nos objetos criados pelo Celso (2017) estão disponíveis nos Apêndices A, B e na Figura 3, correspondendo a importante contribuição técnica do presente artigo, para a comunidade de desenvolvimento ABAP.

Em relação às áreas de qualidade de software e de engenharia de software, esse estudo contribuiu para o avanço da utilização de técnicas de TDD, onde o desenvolvedor acaba realizando uma reflexão antes da modelagem do código funcional resultando em uma funcionalidade bem testada, encorajamento do time de desenvolvimento na realização de *refactory* de código e maior confiança em manutenções em códigos já entregues.

Em relação aos objetivos específicos, todos foram atendidos como esperado na visão do pesquisador. Considera-se que o objetivo "(i) identificar os requisitos para utilização da ampliação no contexto do estudo" foi alcançado, conforme descrito no item 4.1. Já, o "(ii) capacitar o time de desenvolvimento para usar a ampliação na criação dos testes" pode ser visto no item 4.2. O "(iii) analisar os benefícios e as limitações encontradas na utilização do framework" com a ampliação mostrada no item 4.3. E, por fim, o "(iv) analisar comparativamente os resultados obtidos nessa pesquisa com o estudo anterior" foi atendido conforme descrito o item 4.5, em que o conteúdo da Tabela 3 compara o questionário aplicado na pesquisa do Celso (2017), com os resultados do presente estudo. Ademais, a realização da capacitação do time de desenvolvimento oportunizou detectar a necessidade de uma maior curva de aprendizado e, portanto, prever mais tempo de uso do ATDF, para se ter um maior ganho de experiência e de produtividade.

Quanto aos trabalhos futuros, se sugere um estudo comparativo entre o *framework* ampliado e o *standard* com objetivo de validar qual deles pode contribuir melhor para qualidade dos desenvolvimentos. Outra pesquisa futura identificada seria criar funcionalidades para realizar teste real de seleção de dados, com intuito de aumentar a cobertura do código, em ambiente de desenvolvimento ABAP.

Ainda como trabalhos futuros, conforme nota-se nos resultados desse estudo, que ainda há a necessidade de criar uma maneira de testar as seleções de dados em tabelas e funções, sendo que com isso pode haver um aumento na cobertura dos testes, reduzindo a ocorrência de erros e qualificando o código nestes casos.

Referências

AZEVEDO, D.; MACHADO, L.; SILVA, L. V. D. **Métodos e procedimentos de pesquisa:** do projeto ao relatório final. São Leopoldo: Unisinos , 2011.

BARDIN, L. **Análise de Conteúdo**. Lisboa: Persona, 1995.

BECK, K. et al. Agile Manifesto. **Agile Manifesto**, 2001. Disponivel em: <http://www.agilemanifesto.org>. Acesso em: 24 nov. 2018.

FILHO, C. D. S. AMPLIAÇÃO DO ABAP TEST DOUBLE FRAMEWORK (ATDF). **AMPLIAÇÃO DO ABAP TEST DOUBLE FRAMEWORK (ATDF)**, Porto Alegre, 2017. 52.

FOWLER, M. **Refatoração:** Aperfeiçoamento o Projeto de Código Existente. Porto Alegre: Bookman, 2004, 2004.

GIL, A. C. **Como Elaborar Projetos de Pesquisa**. 6ª. ed. São Paulo : Atlas Ltda., 2017.

HARDY, P. **ABAP to the Future**. 1ª. ed. Boston: Rheinwerk Publishing, 2015.

HARDY, P. **ABAP to the Future**. 2ª. ed. Boston: Rheinwerk Publishing, 2016.

HIRAMA, K. **Engenharia de Sotware:** Qualidade e Produtividade com Tecnologia. Rio de Janeiro: ELSEVIER, 2011.

HRON, M.; OBWEGESER, N. Scrum in practice: an overview of Scrum adaptations, Waikoloa Village, Havaí, EUA, 2018.

JUNIT. **About**, 2002. Disponivel em: <https://junit.org/junit4/>. Acesso em: 07 jan. 2019.

LAKATOS, E. M. M. M. D. A. **Metodologia científica**. 5ª. ed. São Paulo: Atlas, 2009.

MEYANA, P. ABAP test double framework: an introduction. **Blog SAP**, 2015. Disponivel em: <https://blogs.sap.com/2015/01/05/abap-test-double-framework-an-introduction/>. Acesso em: 02 jan. 2019.

MYERS, G. J. **The art of software testing**. 3º. ed. New Jersey: John Wiley & Sons, 2012.

PAGOTTO, T.; FABRI, J. A.; JOSÉ, A. L.́. E. J. A. G. Processo de software para desenvolvimento individual. **Scrum Solo**, Cornélio Procópio, 18 Junho 2016. 6.

RICHARDS, M. **Software Architecture Patterns**. Sebastopol, CA: O'Reilly Media, 2015.

RISING, L.; JANOFF, N. S. The Scrum Software Development Process for Small Teams. **The Scrum Software Development Process for Small Teams**, Agosto 2000. 7.

ROSSI, S. Short examples of CL_ABAP_TESTDOUBLE, 2018. Disponivel em: <https://blogs.sap.com/2018/04/03/short-examples-of-cl_abap_testdouble/>. Acesso em: 05 nov. 2018.

SANTANA, L. F.; FERREIRA, T. A. L.; ROCHA, F. G. **Integração entre Scrum e TDD**, 2016. Disponivel em: <https://eventos.set.edu.br/index.php/sempesq/article/viewFile/4224/1327>. Acesso em: 08 Dezembro 2018.

SAP SE. ABAP Programming Language - Overview. **ABAP Keyword Documentation**, 2016. Disponivel em: <https://help.sap.com/doc/abapdocu_750_index_htm/7.50/en-US/index.htm>. Acesso em: 22 jan. 2019.

SAP SE. Test and Analysis Tools in ABAP. **SAP Documentation**, 2016. Disponivel em: <https://help.sap.com/saphelp_nw70ehp2/helpdata/en/49/268dc67b6716b4e10000000a42189d/frameset.htm>. Acesso em: 29 nov. 2018.

SAP SE. ABAP Testing and Analysis. **ABAP Testing and Analysis**. Disponivel em: <https://www.sap.com/community/topics/abap-testing-analysis.html>. Acesso em: 02 13 2019.

SAP SE. Understanding the Code Coverage Display in the ABAP Unit Browser. **SAP Documentation**. Disponivel em: <https://help.sap.com/doc/saphelp_nw70ehp2/7.02.16/en-US/4a/828b7a3a90387fe10000000a421947/content.htm?no_cache=true>. Acesso em: 22 jan. 2019.

SHAIKH MOSTAFA, X. W. An Empirical Study on the Usage of Mocking Frameworks in Software Testing, Dallas, TX, USA , 2-3 Outubro 2014.

SOARES, M. D. S. Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software, Conselheiro Lafaiete, 2004. 8.

SOARES1, M. D. S. Metodologias Ágeis Extreme Programming. **Metodologias Ágeis** , Conselheiro Lafaiete. 8.

SOMMERVILLE, I. **Engenharia de software**. 9º. ed. São Paulo : Pearson Education do Brasil , 2011.

WAINER, J. **Métodos de pesquisa quantitativa e qualitativa para a Ciência da Computação**, 2007. 42.

WAINER, J. Métodos de pesquisa quantitativa e qualitativa para a Ciência da Computação. **Pesquisa Quantitativa e Qualitativa em Ciência da Computação**, Instituto de Computação – UNICAMP, 2007. 42.

WAZLAWICK, R. S. **Engenharia de Software Conceito e Práticas**. São Paulo: Elsevier, 2013.

WELLS, D. Extreme Programming. **Extreme Programming:** A gentle introduction, 2013. Disponivel em: <http://www.extremeprogramming.org/rules/testfirst.html>. Acesso em: 12 jan. 2019.

WILSON BISSI, A. G. S. S. N. M. C. F. P. E. The effects of test driven development on internal quality, external. **The effects of test driven development**, Curitiba, 2016. 54.

ANEXO A – QUESTIONÁRIO DE AVALIAÇÃO

Este questionário foi utilizado por Filho (2017) para avaliar o uso da ampliação do framework para automatizar a criação de *mock objects*. As afirmativas são respondidas com a utilização de um nível de concordância, de acordo com a seguinte escala:

1- Discordo totalmente

2-

3-

4-

5- Concordo totalmente

1. A criação de teste unitário além de garantir que o componente a ser testado atende o que foi solicitado, ajuda o desenvolvedor a escrever um código mais simples.

1 2 3 4 5

2. A grande dificuldade para a criação do teste unitário é a manipulação de objetos dependentes, que interferem no componente a ser testado.

1 2 3 4 5

3. Criar *mock objects* sem uma ferramenta de apoio é uma atividade trabalhosa e quem consome bastante tempo.

1 2 3 4 5

4. A ferramenta criada para automatizar a criação de *mock objects* facilita a criação do teste unitário.

1 2 3 4 5

5. A ferramenta criada para automatizar a criação de *mock objects* resulta em um ganho considerável de tempo na criação do teste unitário.

1 2 3 4 5

6. Em relação a funcionalidade, a ferramenta dispõe de todos os recursos necessários para a criação de *mock objects*.

1 2 3 4 5

7. Em relação a confiabilidade, a ferramenta não apresenta falhas durante sua utilização.

1 2 3 4 5

8. Em relação usabilidade, a ferramenta é fácil de usar.

1 2 3 4 5

9. Em relação a eficiência de desempenho, a ferramenta tem um tempo de execução adequado.

1 2 3 4 5

10.Em relação a satisfação, a ferramenta da confiança e conforto para a criação de *mock objects*.

1 2 3 4 5

APÊNDICE A – Alteração na classe ZCL_ATD_PROXY_GENERATOR_CLASS

```
CLASS zcl_atd_proxy_generator_class DEFINITION
  PUBLIC
  FINAL
  CREATE PRIVATE

  GLOBAL FRIENDS zcl_atd_proxy_factory .

PRIVATE SECTION.

  METHODS add_parameters
    IMPORTING
      !is_methdescr TYPE abap_methdescr
      !iv_parmkind TYPE abap_parmkind
    RETURNING
      VALUE(rt_result) TYPE string_table .
  METHODS generate_call_super_method
    IMPORTING
      !is_methdescr TYPE abap_methdescr
    RETURNING
      VALUE(rt_result) TYPE string_table .
  METHODS get_only_method_name
    IMPORTING
      !is_method TYPE abap_methdescr
    RETURNING
      VALUE(rv_result) TYPE abap_methname .
  METHODS generate_class_definition
    IMPORTING
      !it_class_descr TYPE tt_class_descr
      !iv_class_name TYPE abap_classname
    RETURNING
      VALUE(rt_def_code) TYPE string_table
    RAISING
      cx_atd_proxy_exception .
```

```
CLASS zcl_atd_proxy_generator_class IMPLEMENTATION.


  METHOD add_parameters.
    DATA lv_header TYPE abap_bool.
    LOOP AT is_methdescr-parameters INTO DATA(ls_parameters)
      WHERE parm_kind = iv_parmkind.
      IF lv_header = abap_false.
        CASE iv_parmkind.
          WHEN cl_abap_objectdescr=>importing.
            APPEND `EXPORTING` TO rt_result.
          WHEN cl_abap_objectdescr=>exporting.
            APPEND `IMPORTING` TO rt_result.
          WHEN cl_abap_objectdescr=>changing.
            APPEND `CHANGING` TO rt_result.
          WHEN cl_abap_objectdescr=>returning.
            APPEND `RECEIVING` TO rt_result.
          WHEN OTHERS.
            EXIT.
        ENDCASE.
        lv_header = abap_true.
      ENDIF.


      APPEND ls_parameters-name && ` = ` && ls_parameters-name  TO rt_result.
    ENDLOOP.
  ENDMETHOD.


  METHOD generate_call_super_method.
    APPEND ` CALL METHOD super->` &&  is_methdescr-name TO rt_result.
    APPEND LINES OF me->add_parameters( is_methdescr = is_methdescr iv_parmkind =
cl_abap_objectdescr=>importing ) TO rt_result.
    APPEND LINES OF me->add_parameters( is_methdescr = is_methdescr iv_parmkind =
cl_abap_objectdescr=>exporting ) TO rt_result.
    APPEND LINES OF me->add_parameters( is_methdescr = is_methdescr iv_parmkind =
cl_abap_objectdescr=>changing )  TO rt_result.
```

```
    APPEND LINES OF me->add_parameters( is_methdescr = is_methdescr iv_parmkind =
cl_abap_objectdescr=>returning ) TO rt_result.
    APPEND `.` TO rt_result.
  ENDMETHOD.


  METHOD generate_class_definition.


    APPEND `   public section.` TO rt_def_code.            "#EC NOTEXT


*   Configure call method
    APPEND `  `  TO rt_def_code.
    APPEND `TYPES: BEGIN OF _udt_conf_method_type,` TO rt_def_code.
    LOOP AT ls_class_descr-class_descr->methods INTO ls_method
      WHERE visibility NE cl_abap_objectdescr=>private AND is_class NE abap_true AND name
<> 'CONSTRUCTOR' AND is_abstract = abap_false AND alias_for IS INITIAL.


      APPEND ` BEGIN OF ` && me->get_only_method_name( ls_method ) && `,` TO
rt_def_code.
      APPEND ` as_real TYPE abap_bool,` TO rt_def_code.
      APPEND ` END OF ` && me->get_only_method_name( ls_method ) && `,` TO rt_def_code.
    ENDLOOP.
    APPEND ` END OF _udt_conf_method_type. `  TO rt_def_code.
    APPEND ` DATA _udt_conf_method TYPE _udt_conf_method_type.  `  TO rt_def_code.
    APPEND `  `  TO rt_def_code.
ENDMETHOD.


  METHOD get_only_method_name.
    IF is_method-is_interface = abap_false.
      rv_result = is_method-name.
    ELSE.
      SPLIT is_method-name AT '~' INTO DATA(lv_interface) rv_result.
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

APÊNDICE B – Alteração na classe ZCL_ABAP_TESTDOUBLE

```
CLASS ZCL_ABAP_TESTDOUBLE DEFINITION
  PUBLIC
  FINAL
  CREATE PRIVATE
  FOR TESTING.

  CLASS-METHODS configure_call_method_as_real
    IMPORTING
      !double TYPE REF TO object
      !method TYPE abap_methname .

  CLASS-METHODS configure_call_method_as_mock
    IMPORTING
      !double TYPE REF TO object
      !method TYPE abap_methname .
ENDCLASS.

CLASS zcl_abap_testdouble IMPLEMENTATION.
  METHOD configure_call_method_as_mock.
    DATA lv_men TYPE string.
    FIELD-SYMBOLS <fs_any> TYPE any.

    lv_men = `DOUBLE->_udt_conf_method-` && method && `-as_real`.
    ASSIGN (lv_men) TO <fs_any>.

    " Is not assigned probably the method is a PRIVATE
    IF <fs_any> IS ASSIGNED.
      <fs_any> = abap_false.
    ENDIF.
  ENDMETHOD.

  METHOD configure_call_method_as_real.
```

```
    DATA lv_men TYPE string.
    FIELD-SYMBOLS <fs_any> TYPE any.

    lv_men = `DOUBLE->_udt_conf_method-` && method && `-as_real`.
    ASSIGN (lv_men) TO <fs_any>.

    " Is not assigned probably the method is a PRIVATE
    IF <fs_any> IS ASSIGNED.
    <fs_any> = abap_true.
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

APÊNDICE C – Programa utilizado na capacitação do time de desenvolvimento

```
REPORT zatdf_short_examples.

CLASS lcl_cut DEFINITION  FINAL  CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_atd_demo.
    METHODS constructor  IMPORTING atd_aunit TYPE REF TO zif_atd_demo.

  PRIVATE SECTION.
    DATA: mr_atd_aunit TYPE REF TO zif_atd_demo.
ENDCLASS.

CLASS lcx_cut DEFINITION INHERITING FROM cx_no_check.
ENDCLASS.

CLASS ltc_atd_aunit DEFINITION INHERITING FROM cl_aunit_assert  FINAL  FOR
TESTING DURATION SHORT  RISK LEVEL HARMLESS.

  PRIVATE SECTION.
    METHODS:
      setup,
      raise_exception_good FOR TESTING RAISING cx_static_check,
      raise_exception_good2 FOR TESTING RAISING cx_static_check,
      exporting_good FOR TESTING RAISING cx_static_check,
      returning_good FOR TESTING RAISING cx_static_check,
      exporting_twice_good FOR TESTING RAISING cx_static_check,
      returning_twice_good FOR TESTING RAISING cx_static_check,
      times_good FOR TESTING RAISING cx_static_check,
      times_good2 FOR TESTING RAISING cx_static_check,
      importing_required_good FOR TESTING RAISING cx_static_check,
      importing_required_good2 FOR TESTING RAISING cx_static_check,
      importing_optional_good FOR TESTING RAISING cx_static_check,
```

```
      importing_optional_good2 FOR TESTING RAISING cx_static_check,
      importing_optional_good3 FOR TESTING RAISING cx_static_check,
      changing_good FOR TESTING RAISING cx_static_check,
      ignore_parameter_good FOR TESTING RAISING cx_static_check,
      ignore_all_parameters_good FOR TESTING RAISING cx_static_check,

    DATA: test_double TYPE REF TO zif_atd_demo,
          cut         TYPE REF TO lcl_cut,
          result      TYPE i.

ENDCLASS.

CLASS lcl_cut IMPLEMENTATION.
  METHOD constructor.
    mr_atd_aunit = atd_aunit.
  ENDMETHOD.

  METHOD zif_atd_demo~exporting.
    mr_atd_aunit->exporting(  IMPORTING result = result ).
  ENDMETHOD.

  METHOD zif_atd_demo~demo_raise_exception.
    mr_atd_aunit->demo_raise_exception( ).
  ENDMETHOD.

  METHOD zif_atd_demo~demo_raise_exception2.
    result = mr_atd_aunit->demo_raise_exception2( whatever ).
  ENDMETHOD.

  METHOD zif_atd_demo~demo_returning.
    result = mr_atd_aunit->demo_returning( ).
  ENDMETHOD.

  METHOD zif_atd_demo~importing_optional.
```

```
    IF whatever IS NOT SUPPLIED.
      result = mr_atd_aunit->importing_optional( ).
    ELSE.
      result = mr_atd_aunit->importing_optional( whatever ).
    ENDIF.
  ENDMETHOD.

  METHOD zif_atd_demo~importing_required.
    result = mr_atd_aunit->importing_required( whatever ).
  ENDMETHOD.

  METHOD zif_atd_demo~changing.
    mr_atd_aunit->changing( CHANGING whatever = whatever ).
  ENDMETHOD.

  METHOD zif_atd_demo~demo_raise_event.
    mr_atd_aunit->demo_raise_event( ).
  ENDMETHOD.

  METHOD zif_atd_demo~several_cases.
    result = mr_atd_aunit->several_cases( whatever = whatever ).
  ENDMETHOD.

  METHOD zif_atd_demo~decision_table.
    result = mr_atd_aunit->decision_table( p1 = p1 p2 = p2 ).
  ENDMETHOD.

  METHOD zif_atd_demo~exporting_returning.
    returning = mr_atd_aunit->exporting_returning( IMPORTING exporting = exporting ).
  ENDMETHOD.
ENDCLASS.

CLASS ltc_atd_aunit IMPLEMENTATION.
```

```
METHOD setup.
  " create test double object
  test_double ?= cl_abap_testdouble=>create( 'zif_atd_demo' ).
  " injecting the test double into the object to be tested (Code Under Test)
  CREATE OBJECT cut EXPORTING atd_aunit = test_double.
ENDMETHOD.

METHOD raise_exception_good.
  " GOOD EXAMPLE
  " When the method DEMO_RAISE_EXCEPTION is called
  " Then an exception should be raised
  DATA: lx_exp TYPE REF TO lcx_cut,
        lx_act TYPE REF TO lcx_cut.

  CREATE OBJECT lx_exp.
  cl_abap_testdouble=>configure_call( test_double )->raise_exception( lx_exp ).
  test_double->demo_raise_exception( ).

  TRY.
      cut->zif_atd_demo~demo_raise_exception( ).
    CATCH lcx_cut INTO lx_act.
  ENDTRY.

  assert_equals( exp = lx_exp act = lx_act ).
ENDMETHOD.

METHOD raise_exception_good2.
  " GOOD EXAMPLE
  " When the method DEMO_RAISE_EXCEPTION2 is called with parameter 20
  " Then an exception should be raised
  "
  " When the method DEMO_RAISE_EXCEPTION2 is called with parameter 30
  " Then an exception should not be raised and the returned value should be 88
  DATA: lx_exp TYPE REF TO lcx_cut,
```

```
        lx_act TYPE REF TO lcx_cut.

    CREATE OBJECT lx_exp.
    cl_abap_testdouble=>configure_call( test_double )->raise_exception( lx_exp ).
    test_double->demo_raise_exception2( 20 ).

    cl_abap_testdouble=>configure_call( test_double )->returning( 88 ).
    test_double->demo_raise_exception2( 30 ).

    TRY.
       cut->zif_atd_demo~demo_raise_exception2( 20 ).
     CATCH lcx_cut INTO lx_act.
    ENDTRY.
    assert_equals( exp = lx_exp act = lx_act ).

    result = cut->zif_atd_demo~demo_raise_exception2( 30 ).
    assert_equals( exp = 88 act = result ).
  ENDMETHOD.

METHOD exporting_good.
    " GOOD EXAMPLE
    " When the method EXPORTING is called
    " Then it should return the exporting parameter RESULT with value 20
    cl_abap_testdouble=>configure_call( test_double )->set_parameter( name = 'RESULT' value =
20 ).
    test_double->exporting( ).

    cut->zif_atd_demo~exporting( IMPORTING result = result ).
    assert_equals( exp = 20 act = result ).
  ENDMETHOD.

  METHOD returning_good.
    " GOOD EXAMPLE
    " When the method DEMO_RETURNING is called
```

```
    " Then it should return the returning parameter RESULT with value 97
    cl_abap_testdouble=>configure_call( test_double )->returning( 97 ).
    test_double->demo_returning( ).

    result = cut->zif_atd_demo~demo_returning( ).

    assert_equals( exp = 97 act = result ).
  ENDMETHOD.

  METHOD exporting_twice_good.
    " GOOD EXAMPLE
    " When the method EXPORTING is called multiple times
    " Then it should always return the exporting parameter RESULT with value 20
    cl_abap_testdouble=>configure_call( test_double )->set_parameter( name = 'RESULT' value =
20 ).
    test_double->exporting( ).

    cut->zif_atd_demo~exporting( IMPORTING result = result ).
    cut->zif_atd_demo~exporting( IMPORTING result = result ).

    assert_equals( exp = 20 act = result ).
  ENDMETHOD.

  METHOD returning_twice_good.
    " GOOD EXAMPLE
    " When the method DEMO_RETURNING is called multiple times
    " Then it should return the returning parameter RESULT with value 97
    cl_abap_testdouble=>configure_call( test_double )->returning( 97 ).
    test_double->demo_returning( ).

    result = cut->zif_atd_demo~demo_returning( ).
    result = cut->zif_atd_demo~demo_returning( ).

    assert_equals( exp = 97 act = result ).
```

```
ENDMETHOD.

METHOD times_good.
  " GOOD EXAMPLE
  " When the method DEMO_RETURNING is called the first time
  " Then it should return the returning parameter RESULT with value 97
  "
  " When the method DEMO_RETURNING is called the next times
  " Then it should return the returning parameter RESULT with value 0
  cl_abap_testdouble=>configure_call( test_double
      )->returning( 97 ).
  test_double->demo_returning( ).
  cl_abap_testdouble=>configure_call( test_double
      )->returning( 0 ).
  test_double->demo_returning( ).

  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 97 act = result ).

  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 0 act = result ).

  " from now on (max times consumed), the last configuration is used
  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 0 act = result ).
ENDMETHOD.

METHOD times_good2.
  " GOOD EXAMPLE
  " When the method DEMO_RETURNING is called the two first times
  " Then it should return the returning parameter RESULT with value 97
  "
  " When the method DEMO_RETURNING is called the next times
  " Then it should return the returning parameter RESULT with value 0
```

```
  cl_abap_testdouble=>configure_call( test_double
      )->returning( 97
      )->times( 2 ).
  test_double->demo_returning( ).
  cl_abap_testdouble=>configure_call( test_double
      )->returning( 0 ).
  test_double->demo_returning( ).

  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 97 act = result ).

  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 97 act = result ).

  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 0 act = result ).

  " from now on (max times consumed), the last configuration is used
  result = cut->zif_atd_demo~demo_returning( ).
  assert_equals( exp = 0 act = result ).
ENDMETHOD.

METHOD importing_required_good.
  " GOOD EXAMPLE
  " When the method IMPORTING_REQUIRED is called with parameter 5555
  " Then it should return 97
  cl_abap_testdouble=>configure_call( test_double )->returning( 97 ).
  test_double->importing_required( 5555 ).

  result = cut->zif_atd_demo~importing_required( 5555 ).

  assert_equals( exp = 97 act = result ).
ENDMETHOD.
```

```
  METHOD importing_required_good2.
    " GOOD EXAMPLE
    " When the method IMPORTING_REQUIRED is called with any parameter value
    " Then it should return 97
    " (note: this test method is the duplicate of IGNORE_PARAMETER_GOOD)
    cl_abap_testdouble=>configure_call( test_double
        )->ignore_parameter( 'WHATEVER' )->returning( 97 ).
    test_double->importing_required( whatever = 5554 ). " any value, it's ignored

    result = cut->zif_atd_demo~importing_required( 5555 ).

    assert_equals( exp = 97 act = result ).
  ENDMETHOD.

  METHOD importing_optional_good.
    " GOOD EXAMPLE
    " When the method IMPORTING_OPTIONAL is called with the optional parameter equal to
any value,
    " Then it should return 97
    cl_abap_testdouble=>configure_call( test_double
        )->ignore_parameter( 'WHATEVER' )->returning( 97 ).
    test_double->importing_optional( ).

    result = cut->zif_atd_demo~importing_optional( whatever = 5555 ).

    assert_equals( exp = 97 act = result ).
  ENDMETHOD.

  METHOD importing_optional_good2.
    " GOOD EXAMPLE
    " When the method IMPORTING_OPTIONAL is called without the optional parameter
    " Then it should return 97
    cl_abap_testdouble=>configure_call( test_double )->returning( 97 ).
    test_double->importing_optional( ).
```

```
  result = cut->zif_atd_demo~importing_optional( ).

  assert_equals( exp = 97 act = result ).
ENDMETHOD.

METHOD importing_optional_good3.
  " GOOD EXAMPLE
  " When the method IMPORTING_OPTIONAL is called with the optional parameter 5555
  " Then it should return 97
  cl_abap_testdouble=>configure_call( test_double )->returning( 97 ).
  test_double->importing_optional( 5555 ).

  result = cut->zif_atd_demo~importing_optional( 5555 ).

  assert_equals( exp = 97 act = result ).
ENDMETHOD.

METHOD changing_good.
  " GOOD EXAMPLE
  " When the method CHANGING is called with the changing parameter equal to 0
  " Then it should be changed to 40
  "
  " When the method CHANGING is called with the changing parameter equal to 50
  " Then it should be changed to 110
  DATA: whatever TYPE i.

  cl_abap_testdouble=>configure_call( test_double
      )->set_parameter( name = 'WHATEVER' value = 40 ).
  whatever = 0.
  test_double->changing( CHANGING whatever = whatever ).

  cl_abap_testdouble=>configure_call( test_double
      )->set_parameter( name = 'WHATEVER' value = 110 ).
```

```
  whatever = 50.
  test_double->changing( CHANGING whatever = whatever ).

  whatever = 0.
  cut->zif_atd_demo~changing( CHANGING whatever = whatever ).
  whatever = whatever + 10.
  cut->zif_atd_demo~changing( CHANGING whatever = whatever ).

  assert_equals( exp = 110 act = whatever ).
ENDMETHOD.

METHOD ignore_parameter_good.
  " GOOD EXAMPLE
  " When the method IMPORTING_REQUIRED is called with any parameter value
  " Then it should return 97
  cl_abap_testdouble=>configure_call( test_double
      )->ignore_parameter( 'WHATEVER' )->returning( 97 ).
  test_double->importing_required( 1234 ). " <=== important call & dummy value !

  result = cut->zif_atd_demo~importing_required( 5555 ).

  assert_equals( exp = 97 act = result ).
ENDMETHOD.

METHOD ignore_all_parameters_good.
  " GOOD EXAMPLE
  " When the method IMPORTING_REQUIRED is called with any parameter value
  " Then it should return 97
  cl_abap_testdouble=>configure_call( test_double
      )->ignore_all_parameters( )->returning( 97 ).
  test_double->importing_required( 5554 ). " (dummy value)

  result = cut->zif_atd_demo~importing_required( 5555 ).
```

```
    assert_equals( exp = 97 act = result ).
  ENDMETHOD.

ENDCLASS.
```

APÊNDICE D – Resultado da execução do programa utilizado na capacitação do time de desenvolvimento