UNIVERSIDADE DE CAXIAS DO SUL
CENTRO DE COMPUTAÇÃO E TECNOLOGIA DA INFORMAÇÃO
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LUCIANO CAMARGO CRUZ

Desenvolvimento de um *framework* para a realização de testes funcionais no *workflow* Plone

André Luis Martinotto
Orientador

João Luis Tavares da Silva
Coorientador

Caxias do Sul, Dezembro de 2013

"No lugar que eu estou, para mim é o melhor lugar do mundo."
Oswaldo Lara Cruz

AGRADECIMENTOS

À minha mãe, Elba de Fátima Camargo Cruz, por ter me criado com tanta coragem, amor e dedicação. Um exemplo de pessoa a se seguir.

Ao meu pai, Sargento Oswaldo Lara Cruz, por ter me guiado nos meu primeiros passos.

À minha noiva, Drª Fabiana Tonial, pelo amor, carinho, paciência, companheirismo, por ter me ensinado a humildade e por acreditar em mim ao longo desses 10 anos que estamos juntos.

Ao meu irmão, Leandro Camargo Cruz, por ter auxiliado em minha criação.

Ao meu segundo pai, Drº Jenoino Tonial, por ter me ensinado que sempre devemos honrar nosso nome e que com honestidade alcançamos os sonhos.

À minha segunda mãe, Maria Alda Minuscoli Tonial, por me tratar como filho.

Aos meus cunhados, Leonardo Tonial, Leandro Tonial e Milton Tonial, por me tratarem como irmão.

Aos meus padrinhos, Carlos e Carmen, pelo carinho e cuidados.

Aos meus amigos, Leandro Pereira e Lucas Dezordi Lopes, por toda a força e amizade em todos esses anos.

Aos amigos e colegas, João Toss Molon e Matheus Pereira, pela ajuda e amizade ao longo da graduação, no trabalho e fora dele.

Ao meu amigo e orientador, Drº João Luis Tavares da Silva, pela ajuda no desenvolvimento da minha carreira profissional, por sempre estar disposto em me ajudar e pela ajuda no desenvolvimento desse trabalho.

Ao meu amigo, Drº Alexandre Moretto Ribeiro, pela oportunidade profissional, que certamente sem ela não estaria no processo de conclusão dessa graduação.

Ao meu amigo e orientador, Drº André Luis Martinotto, por ter me ensinado, no inicio da graduação, realmente como se programa e pela orientação desse trabalho.

Ao meu amigo, Dorneles Treméa, por ter me ajudado nos meus primeiros passos com Python/Zope/Plone.

À minha tia, Marlene Camargo Bedin, por ter me ajudado em meus primeiros dias em Caxias do Sul.

À família Dambroz, por ter me ajudado com moradia.

Aos meu colegas de trabalhos, entre eles o Felipi Medeiros Macedo, com infinita ajuda em JS.

À comunidade de software livre Python/Zope/Plone, por me ajudar com inúmeras questões.

À toda minha família e a todas as pessoas que, direta ou indiretamente, ajudaram a tornar o curso de graduação e este trabalho possível.

A todos vocês, minha sincera gratidão.

Luciano Camargo Cruz

SUMÁRIO

LISTA DE ACRÔNIMOS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 9

LISTA DE TRECHOS DE CÓDIGO 10

RESUMO 11

ABSTRACT 12

1 INTRODUÇÃO 13
1.1 Organização do texto 14

2 TESTES DE SOFTWARE 15
2.1 Técnicas de Testes de Software 16
2.1.1 Teste Estrutural 17
2.1.2 Teste Funcional 19
2.1.3 Testes Automatizados 21
2.2 Estratégias de Testes 22
2.2.1 Teste de Unidade 22
2.2.2 Teste de Integração 28
2.2.3 Teste de Regressão 28
2.2.4 Teste de Sistema 28
2.2.5 Teste de Desempenho 29
2.3 Considerações Finais 29

3 FUNCIONAMENTO DE WORKFLOWS EM PLONE 30
3.1 Python 30
3.2 *Zope* (*Z Object Publishing Environment*) 31

3.3 Plone 32
3.4 Workflow do Plone 33
3.4.1 Funcionamento do *workflow* em um portal Plone 34
3.4.2 Gerenciando *Workflows* no Plone 37
3.5 Considerações Finais 40

4 WORKFLOW.VALIDATION - FRAMEWORK DE TESTES PARA WORKFLOW PLONE 41
4.1 Caso de uso 42
4.1.1 Cadastro de *workflows* 43
4.1.2 Cadastro de estados 44
4.1.3 Cadastro de papéis 45
4.1.4 Cadastro de permissões 46
4.1.5 Efetuar ligação 47
4.1.6 Exportação do arquivo ZIP com as regras e *scripts* de testes de *workflow* 48
4.1.7 Executar teste 49
4.2 Arquitetura do *framework workflow.validation* 50
4.3 Diagrama do Banco de Dados 51
4.4 Diagrama de Classe 52
4.5 Desenvolvimento da Solução 53
4.6 Cenário de Uso 59
4.7 Considerações Finais 61

5 CONCLUSÃO 63
5.1 Contribuições 63

REFERÊNCIAS 64

LISTA DE ACRÔNIMOS

HTML	*HyperText Markup Language*
XML	*Extensible Markup Language*
Zope	*Z Object Publishing Environment*
ZMI	*Zope Management Interface*
ZPL	*Zope Public License*
CTO	*Zope Corporation*
ZODB	*Zope Object Data Base*
GPL	*General Public License*
CMS	*Content Management System*
ORM	*Object Relational Mapper*
TALES	*Template Attribute Language Expression*

LISTA DE FIGURAS

Figura 2.1: Modelo geral do processo de teste de um software (Adaptado de (SOMMERVILLE, 2003)) 16
Figura 2.2: Grafo de Fluxo referente ao Trecho de Código 2.1 18
Figura 2.3: Fluxograma referente ao Trecho de Código 2.2 20

Figura 3.1: Interface de *ZMI* (*Zope Management Interface*) 31
Figura 3.2: Plone 33
Figura 3.3: Worlflow padão Plone 34
Figura 3.4: Escolha do tipo de conteúdo a ser criado em um portal Plone. . . 35
Figura 3.5: Conteúdo tipo "Pasta" com estado "Privado" do *workflow* como padrão. 35
Figura 3.6: Menu para troca de "Estados". 36
Figura 3.7: Conteúdo com estado "Publicado". 36
Figura 3.8: Exemplo de *workflows* nativos do Plone 37
Figura 3.9: Exemplo de tratamento de estados de um *workflow* 37
Figura 3.10: Conjunto de permissões e os papéis de um *workflow* 38
Figura 3.11: Matriz de transições de um *workflow* 38

Figura 4.1: Diagrama de caso de uso (conforme (LARMAN, 2002)) 42
Figura 4.2: Tela de cadastro de *workflow* 43
Figura 4.3: Tela de cadastro de estados 44
Figura 4.4: Tela de cadastro de papéis 45
Figura 4.5: Tela de cadastro de permissões 46
Figura 4.6: Tela de ligação 47
Figura 4.7: Tela de exportação os arquivo .ZIP 48
Figura 4.8: Tela de execução dos testes de validação 49
Figura 4.9: Diagrama da arquitetura do *framework workflow.validation* 50
Figura 4.10: Diagrama do Banco de Dados 51
Figura 4.11: Diagrama de Classe 52
Figura 4.12: Requisito cadastrado no produto "workflow.validation" 59

Figura 4.13: Execução dos testes de validação de *workflow* identificando um erro 61
Figura 4.14: Execução dos testes de validação de *workflow* sem identificar erros 61

LISTA DE TABELAS

Tabela 3.1: Definições básicas das ações para cada tipo de papel 36

Tabela 4.1: Cadastro de *workflows* 43
Tabela 4.2: Cadastro de estados 44
Tabela 4.3: Cadastro de papéis 45
Tabela 4.4: Cadastro de permissões 46
Tabela 4.5: Efetuar ligação 47
Tabela 4.6: Exportação do arquivo ZIP com as regras e *scripts* de testes de *workflow* 48
Tabela 4.7: Executar teste 49

LISTA DE TRECHOS DE CÓDIGO

2.1 Função em Python para a localização de um caractere em uma *string* 18
2.2 Exemplo de software otimizado que localiza um caractere em uma *string* 20
2.3 Exemplo do código para automatizar os testes de unidade para a classe CalculadoraSimples. 24
2.4 Exemplo da execução do código de teste utilizando o *framework* Python *unittest*. 24
2.5 Exemplo de uma simples calculadora que converte um número de Hexadecimal em Decimal. 25
2.6 Exemplo da execução do código de teste utilizando o *framework* Python *unittest*, tendo como saída um caso de sucesso. 26
2.7 Exemplo do código para automatizar os testes de unidade para a classe CalculadoraSimples, com o caso de teste para validar letras maiúsculas. 26
2.8 Exemplo da execução do código de teste utilizando o *framework* Python *unittest*. Inserção de um novo caso de teste. Caso com falha. 26
2.9 Exemplo de uma simples calculadora que converte Hexadecimal em Decimal, aceitando letras maiúsculas e minúsculas como parâmetro. . 27
2.10 Exemplo da execução do código de teste utilizando o *framework* Python *unittest*. Com a correção para aceitar letras maiúsculas. . . . 28
4.1 Arquivo de configuração do produto. 53
4.2 Classe Workflow no padrão do *SQLAlchemy*. 53
4.3 Funcionalidades de manipulação dos dados de *workflow*. 54
4.4 Funções que são empacotadas no arquivo ZIP a ser exportado. 55
4.5 Classe que é empacotada no arquivo ZIP a ser exportado responsável pelos testes. 56
4.6 Classe que gera e exporta o arquivo ZIP. 57
4.7 *Script* Python que realiza dos testes. 59

RESUMO

Este trabalho apresenta como principal objetivo o desenvolvimento de um *framework* que permita automatizar os testes funcionais de um *workflow* Plone. O sistema de *workflow* do Plone é uma ferramenta de segurança que executa o controle de acesso aos conteúdos e que também controla um ciclo de vida do conteúdo. Uma limitação encontrada no Plone é a inviabilidade de realização de testes funcionais nas regras de *workflow*, ou seja, a realização de testes que permitam verificar se o software atende os requisitos iniciais. Essa limitação deve-se ao fato do Plone não apresentar recursos que permitam armazenar os requisitos iniciais do *workflow*. Isso dificulta a geração de testes funcionais de forma automática, já que os requisitos iniciais não são conhecidos. Esse trabalho propõe o desenvolvimento de um *framework* que permita automatizar os testes funcionais de um *workflow* Plone. Esse *framework* será responsável por armazenar os requisitos iniciais do *workflow*, efetuar a exportação desses requisitos em um arquivo *XML* (*Extensible Markup Language*) e gerar códigos básicos de testes. Esse arquivo, juntamente com o *workflow* exportado do Plone, poderá ser testado de forma a garantir que o *workflow* desenvolvido no Plone esteja coerente com os requisitos iniciais.

Palavras-chave: Testes, Plone, *Workflow*.

English Abstract

ABSTRACT

This paper aims to present a framework to build automated testing for Plone Workflow. The Plone Workflow system is a security tool that performs access control to objects and which also controls the life cycle of contents. The limitation found in Plone is the impossibility of performing functional tests on the rules of the workflow, i.e., the testing needed to verify whether the software meets the initial requirements. This limitation is due to the fact that Plone does not provide resources to store the initial requirements of the workflow. This hinders the generation of functional tests automatically, since the initial requirements are not known. This work presents a proposal of a framework that will allow to store the initial requirements of a Plone workflow, allowing the export of these requirements using a XML standard format and generating a basic code testing. This pattern, along with the Plone workflow XML representation, can be tested to ensure that the developed Plone workflow is consistent with the initial requirements.

Keywords: Test, Workflow, Plone.

1 INTRODUÇÃO

A dificuldade em se desenvolver um software com baixo percentual de erro é alta, sendo que uma das principais formas para diminuir esse percentual de erros consiste na utilização de testes. Com a utilização de testes é possível validar se o software atende os requisitos, se uma função ou método retorna o valor esperado e se uma ação executa a tarefa corretamente, ou seja, não apresenta "erros" (BERNARDO; KON, 2008; DELAMARO; MALDONADO; JINO, 2007).

Segundo (NETO, 2008), um defeito (*Fault*) é cometido por um indivíduo através do uso incorreto de um processo, método ou ferramenta. Já um erro (*Error*) é a manifestação concreta de um defeito em um artefato computacional, geralmente devido a um desvio de especificação. Por fim, uma falha (*Failure*) é um comportamento operacional do software diferente do esperado pelo usuário. Nesse trabalho vamos considerar o termo "erro" como sendo o objetivo dos testes e o contexto de solução que será desenvolvida.

A realização de testes é uma etapa no desenvolvimento de software que visa identificar se uma determinada tarefa atende aos requisitos. Também é um processo de execução de um programa com a intenção de localizar erros (MYERS, 2004). Esta etapa é de grande importância no desenvolvimento de um software, pois apresenta as falhas aos programadores antes da entrega do software ao cliente.

Na comunidade do desenvolvimento de software livre é de extrema importância a criação de testes automatizados. Esses softwares são, muitas vezes, desenvolvidos por diversas pessoas distribuídas por todo o mundo. Nesse caso a utilização de testes automatizados contribui com a qualidade do software. Uma vez que o desenvolvedor ao efetuar qualquer alteração ou agregar novas funcionalidades pode executar os testes de forma automática garantindo que as novas funcionalidades adicionadas não introduziram erros ao software. Além disso, a existência de testes facilita o entendimento do funcionamento lógico do código.

Nesse contexto destaca-se a utilização de testes na pilha tecnológica formada pela linguagem de programação Python (PYTHON, 2013), o servidor de aplicações Web Zope (ZOPE, 2013) e o gerenciador de conteúdo Plone (PLONE, 2013). A utilização

de testes automatizados no Plone é de grande interesse uma vez que, atualmente, os produtos utilizados no Plone são desenvolvidos por 189 membros distribuídos em diferentes continentes (GITPLONE, 2013).

No Plone, um *workflow* é um processo para gerenciar o acesso e a permissão a um conteúdo em um Website. Este sistema de *workflow* é uma ferramenta de segurança que executa o controle de acesso aos conteúdos e que também controla um ciclo de vida do conteúdo. O *workflow* pode ser representado como um grafo, onde os nodos representam os estados de acessibilidade do conteúdo e as arestas, as transições entre esses estados. O Plone possibilita que o desenvolvedor crie regras de *workflow* que representam as definições de permissão de acesso, os estados de um determinado conteúdo e as transições dos estados (ALBA, 2010; STAHL, 2013).

Uma limitação encontrada no Plone é a inviabilidade de realização de testes funcionais nas regras de *workflow*, ou seja, a realização de testes que permitam verificar se o software atende os requisitos iniciais. Essa limitação deve-se ao fato do Plone não apresentar recursos que permitam armazenar os requisitos iniciais do *workflow*. Isso dificulta a geração de testes funcionais de forma automática, já que os requisitos iniciais não são conhecidos. Assim, esse trabalho propõe-se o desenvolvimento de um *framework* que permita automatizar os testes funcionais de um *workflow* Plone. Esse *framework* será responsável por armazenar os requisitos iniciais do *workflow*, efetuar a exportação desses requisitos em um arquivo *XML* (*Extensible Markup Language*) e gerar códigos básicos de testes. Esse arquivo, juntamente com o *workflow* exportado do Plone, poderá ser testado de forma a garantir que o *workflow* desenvolvido no Plone esteja coerente com os requisitos iniciais.

1.1 Organização do texto

O Capítulo 2 trata de Testes de Software. É abordado uma visão geral sobre testes de software, apresentando algumas definições, as principais técnicas e estratégias de testes.

O Capítulo 3 aborda conceitos como o funcionamento de *workflows* em Plone, as definições básicas, as principais tecnologias utilizadas e o seu funcionamento.

O Capítulo 4 apresenta a solução desenvolvida para o problema desse trabalho. Também é apresentado descrições detalhadas dos casos de uso, definições de diagramas, explicações dos principais trechos de códigos e para validar e experimentar a abordagem proposta, é demostrado um cenário real de uso. O Capítulo 5 refere-se à conclusão do trabalho.

2 TESTES DE SOFTWARE

Os testes de software constituem um processo que apresenta como principal objetivo localizar erros no software. Eles podem ir desde testes de baixo nível, que verificam isoladamente cada funcionalidade do código, a testes de alto nível, que possuem como finalidade validar se o software atende aos requisitos iniciais. O processo de criação de testes é de grande importância no desenvolvimento do software, pois se esses forem bem desenvolvidos, possibilitam a descoberta de erros ainda no processo de desenvolvimento, ou seja, antes da entrega do software ao cliente (DELAMARO; MALDONADO; JINO, 2007; PRESSMAN, 2006; MYERS, 2004).

A descoberta de erros pode ser realizada fazendo uso de inúmeras atividades, que são denominadas de verificação e validação. A verificação corresponde à agregação de várias atividades com a finalidade de assegurar que o software implemente uma respectiva função de forma correta. Já a validação refere-se a um conjunto de atividades realizadas com o propósito de garantir que o software atenda aos requisitos iniciais (DELAMARO; MALDONADO; JINO, 2007; PRESSMAN, 2006; MYERS, 2004).

A utilização das atividades como a verificação e validação contribuem para um ganho na qualidade de um software. Segundo a IEEE 610.12-1990 (IEEE, 1990) pode-se definir qualidade de software, como o grau de um sistema, componente ou processo em satisfazer as especificações dos requisitos e as necessidades ou expectativas do usuário. Já a NBR ISO 9000-2005 (ABNT, 2005) define a qualidade de software como o grau no qual um conjunto de características satisfazem os requisitos (ROCHA; MALDONADO; WEBER, 2001). Embora não seja possível, através de testes, provar que um software esteja totalmente correto, testes criteriosos contribuem para aumentar a qualidade do software.

2.1 Técnicas de Testes de Software

Os testes de software são realizados com a finalidade de localizar erros. Segundo (SOMMERVILLE, 2003):

> "Um teste bem sucedido para a detecção de defeitos é aquele que faz com que o sistema opere incorretamente e, como consequência, expõe um defeito existente. Isso enfatiza um fato importante sobre os testes e demonstra a presença, e não a ausência, de defeitos de programa."

A Figura 2.1 apresenta um modelo geral para o processo para a realização de teste em um software. Esse modelo inicia com a atividade de "Projetar casos de teste", onde definem-se os casos de teste a serem realizados. Já na etapa "Preparar dados de testes" são definidos os parâmetros de entradas, de saída previamente conhecidos e, por fim, a descrição do conteúdo a ser testado. Os dados de testes são os elementos de entrada para a fase seguinte que consiste na execução do teste. A etapa "Executar programa com os dados de teste" corresponde a execução do programa propriamente dito. Como entrada são utilizados os dados definidos na etapa anterior. Por fim, na etapa "Comparar resultados com os casos de teste" o resultado dos testes é comparado com os dados de saída previamente conhecidos. Se esse retorno for igual ao resultado esperado, pode-se assumir que não foram identificados erros neste programa (SOMMERVILLE, 2003).

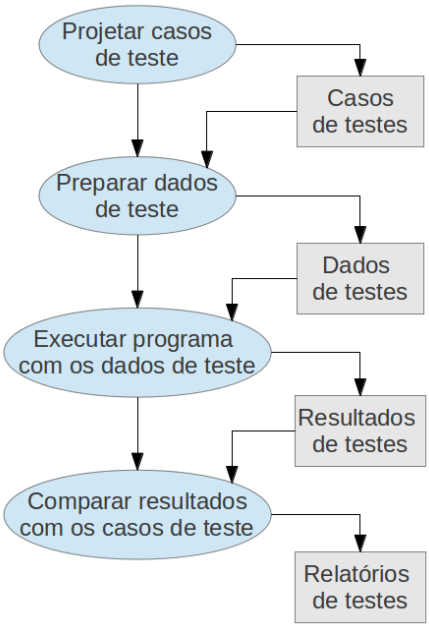

Figura 2.1: Modelo geral do processo de teste de um software (Adaptado de (SOMMERVILLE, 2003))

Existem várias técnicas de testes para softwares, sendo que as mais utilizadas e referenciadas são os testes estruturais, funcionais e mais recentemente os automatizados. Nas próximas seções será apresentada uma descrição dessas técnicas.

2.1.1 Teste Estrutural

Os testes estruturais, também conhecidos como testes de caixa branca, são os testes que abrangem a estrutura interna do software. Nesse tipo de teste as partes do código são testadas de forma isolada, com a finalidade de garantir que todo o código seja executado ao menos uma vez com sucesso. Todas as estruturas de controle do código são cobertos por esta técnica, podendo ser testados os retornos de métodos, bem como códigos estruturados. Esses testes são executados quando o código fonte é finalizado e, além disso, os responsáveis pela execução dos testes necessitam ter um conhecimento estrutural da implementação do software (PRESSMAN, 2006; SOMMERVILLE, 2003).

Fazendo uso da técnica de testes estruturais, o responsável pelo processo de teste pode construir casos de teste direcionados às seguintes atividades (PRESSMAN, 2006):

- Garantir que todos os caminhos independentes de um módulo tenham sido visitados pelo menos uma vez;
- Executar todas as decisões lógicas em seus lados verdadeiro e falso;
- Executar todos os ciclos nos seus limites e dentro de seus intervalos operacionais;
- Executar as estruturas de dados internas de forma a garantir sua validade.

A construção dos casos de testes estruturais, em sua grande maioria utiliza um grafo de fluxo ou grafo de programa para representar o fluxo de controle lógico do software. No grafo de fluxo, os nós representam os comandos procedimentais e as arestas, os fluxos de controle (DELAMARO; MALDONADO; JINO, 2007).

O Trecho de Código 2.1 exemplifica o uso da técnica estrutural. Esse Trecho de Código corresponde a uma função que permite localizar um caractere em uma *string* e foi desenvolvido na linguagem de programação Python (PYTHON, 2013). O Trecho de Código foi implementado usando uma lógica mais simplista apenas para exemplificar um fluxo mais completo. Dessa forma, neste exemplo não foram considerados fatores como estrutura e desempenho de código.

A Figura 2.2 é a representação do Trecho de Código 2.1 em forma de um grafo de fluxo. Esse grafo é formado por 7 nós e 8 arestas, sendo que o nó 1 corresponde às linhas 9 e 10, as linhas 11, 12, 13, 14, 15 e 16 são representadas pelos nós 2, 3, 4, 5, 6 e 7, respectivamente. O comando *if* na linha 12 e o comando *break* na linha 14

representam um desvio de execução entre os nós. Se a condição de teste do comando *if* for verdadeira então ocorre um desvio de execução do nó 3 para o nó 4, em caso contrário o desvio de execução ocorre do nó 3 para o nó 7, bem como o comando *break* na linha 14, se executado, causa um desvio de execução do nó 5 para o nó 6 e de maneira semelhante os demais desvios são constituídos.

Trecho de Código 2.1: Função em Python para a localização de um caractere em uma *string*

```
1 #-*- coding: utf-8 -*-
2
3 def get_posicao(palavra, letra):
4     """
5         Encontra uma letra em um palavra.
6         Recebe uma palavra<string> e uma letra<string>.
7         Retorna a posicao da letra na palavra se encontrar, senao retorna -1.
8     """
9     cont = 0
10    posicao = -1
11    while len(palavra) > cont:
12        if palavra[cont] == letra:
13            posicao = cont
14            break
15        cont += 1
16    return posicao
```

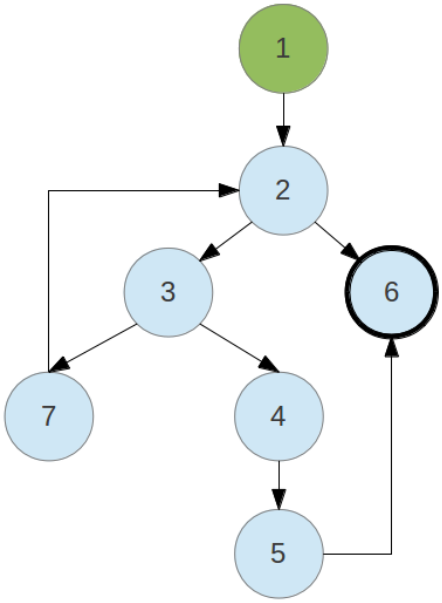

Figura 2.2: Grafo de Fluxo referente ao Trecho de Código 2.1

A definição do grafo de fluxo possibilita estabelecer caminhos que definem a transição entre os nós do grafo. Se forem desenvolvidos testes que executem esses caminhos, todos os comandos do software terão sido executados pelo menos uma vez. O problema é definir o número de caminhos a serem executados. Esse problema pode

ser solucionado utilizando o cálculo da complexidade ciclomática. A complexidade ciclomática é uma medida que fornece a complexidade lógica de um software. Essa medida utilizada no âmbito de testes define o número limite supremo de caminhos a serem percorridos no grafo. Esse número limite determina a quantidade de testes que devem ser desenvolvidos para garantir que todos os comandos sejam executados pelo menos uma vez. A complexidade ciclomática, $v(G)$, é definida para um grafo de fluxo como (PRESSMAN, 2006):

$$v(G) = E - N + 2$$

onde E é o número de arestas e N é o número de nós.

A execução do calculo $v(G)$ para a Figura 2.3 do grafo de fluxo resulta em 3 caminhos, $v(G) = 8 - 7 + 2 = 3$. De fato, os possíveis caminhos são:

- caminho 1: 1,2 e 6
- caminho 2: 1,2,3,7,2 e 6
- caminho 3: 1,2,3,4,5 e 6

Desta forma, uma estratégia de teste de software pode ser definida para cada caminho independente de um programa, sendo que a quantidade de casos de teste é definida pela complexidade ciclomática do programa.

2.1.2 Teste Funcional

Os testes funcionais são baseados nas especificações iniciais do sistema e possuem como finalidade validar se o sistema implementa as funcionalidades que foram definidas na fase dos requisitos. Os testes funcionais também são conhecidos como testes de caixa preta, uma vez que não abordam o funcionamento interno do código. Nesse tipo de testes são informados apenas os dados de entrada para uma rotina de teste, sendo que o resultado é comparado com resultados previamente conhecidos. Segundo (PRESSMAN, 2006), um teste de caixa preta visa verificar as funcionalidades do processo como um todo e não funcionalidades estruturais internas.

A técnica de teste funcional pode ser utilizada nos mais diversos tipos de software. No Trecho de Código 2.2 tem-se uma função que busca a posição de um caractere em uma *string* e, caso o caractere não seja encontrado, a função retorna -1. Pode-se aplicar a técnica funcional nesse software mesmo sem conhecer a estrutura interna do código, sendo que para isso basta conhecer os parâmetros de entrada e os dados de retorno. O fluxograma da Figura 2.3 representa o funcionamento geral do Trecho de Código 2.2. Com o conhecimento do funcionamento geral do software o testador de código pode definir os parâmetros de entrada, efetuar a execução do software de forma manual e analisar os retornos. A partir da análise dos valores de retorno é

possível validar se o software atende aos requisitos iniciais.

Trecho de Código 2.2: Exemplo de software otimizado que localiza um caractere em uma *string*

```
#-*- coding: utf-8 -*-

def get_posicao(palavra, letra):
    """
        Encontra uma letra em um palavra.
        Recebe uma palavra<string> e uma letra<string>.
        Retorna a posicao da letra na palavra se encontrar, senao retorna -1.
    """
    posicao = palavra.find(letra)
    if posicao >= 0:
        return posicao
    else:
        return -1
```

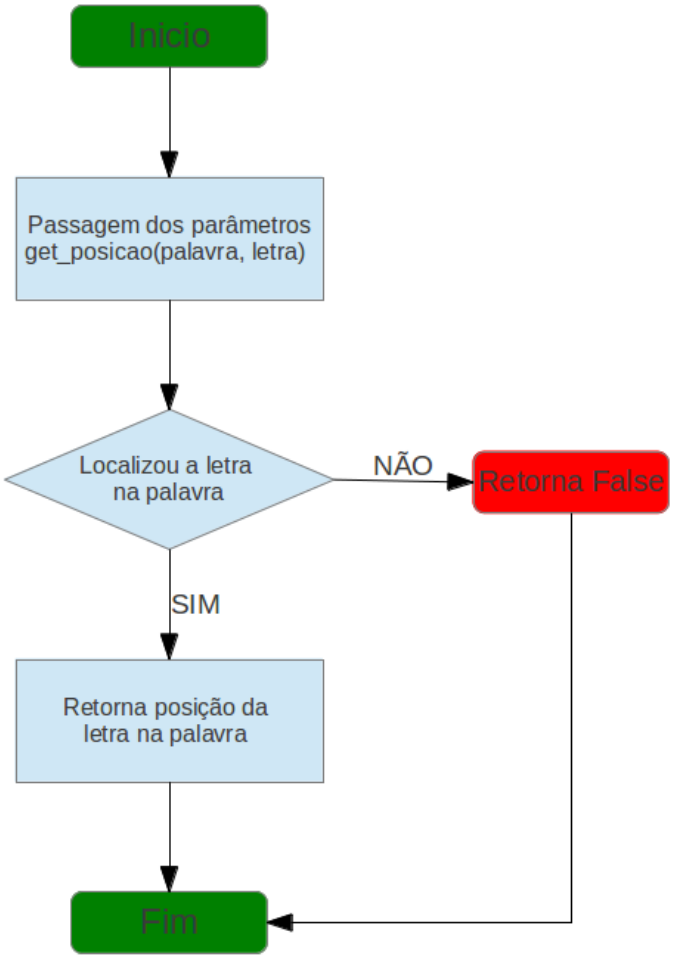

Figura 2.3: Fluxograma referente ao Trecho de Código 2.2

2.1.3 Testes Automatizados

Os testes automatizados são códigos que testam as funcionalidades do sistema de forma automática, podendo ser disparados inúmeras vezes no processo de desenvolvimento do sistema. A execução dos testes automatizados pode ser agendada em servidores específicos ou ainda quando os computadores estão ociosos. A utilização dessa prática agrega ganhos futuros, pois após a criação dos testes os desenvolvedores reduzem o tempo necessário para a realização de testes manuais e, consequentemente, aumentam o tempo disponível para o desenvolvimento de novas tarefas (FEWSTER; GRAHAM, 1999).

A utilização de testes automatizados acarreta em ganho de tempo e qualidade ao software, quando comparada com a realização de testes manuais. Um caso de teste pode ser executado manualmente de forma rápida, mas torna-se mais trabalhoso quando executado manualmente inúmeras vezes. Além disso, a tarefa de comparar resultados de um grande volume de dados é extremamente exaustiva para um humano, aumentando a possibilidade de ser efetuada com erros.

Nesse contexto, o emprego da técnica de automatização de testes resultará na diminuição do tempo de execução do processo de testes, proporcionando um aumento da produtividade. Além disso, essa técnica pode ser empregada para testes de desempenho e carga de software, as quais são complexas de serem testadas de forma manual em um ambiente de desenvolvimento (BERNARDO; KON, 2008; FEWSTER; GRAHAM, 1999).

2.1.3.1 Ferramentas para automatização de testes

A automatização de testes é proporcionada por ferramentas ou *frameworks* que possibilitam a execução dos testes de forma automatizada. Essas ferramentas possibilitam que o desenvolvedor crie casos de testes com a finalidade de validar os códigos desenvolvidos. São inúmeras as ferramentas para automatização de testes, sendo que cada linguagem de programação possui uma ou mais ferramentas. Segundo o *Software QA and Testing Resource Center*[1] existem mais de 500 ferramentas sobre testes e qualidade de software. A seguir são apresentados, apenas em caráter de exemplificação, três *frameworks* que são utilizados para automatização de testes em linguagens de programação distintas:

- O *JUnit* é um *framework* que possibilita a criação de códigos com a finalidade de automatizar testes para a linguagem de programação Java. Foi criado por Kent Beck e Erich Gamma baseado no *framework* de testes do *Smalltalk*. Mais informações sobre o *JUnit* podem ser encontradas em: *http://www.junit.org*.
- O *Unittest* (UNITTEST, 2013), também chamado de *PyUnit*, é um

[1] http://www.softwareqatest.com/qatweb1.html

framework para testes unitários baseado no *JUnit*, escrito para linguagem de programação *Python*. Esse foi iniciado por Steve Purcell, sendo introduzido na versão do *Python* 2.1, tornando-se biblioteca padrão desde então. Na Seção 2.2.1.1 é exemplificado o uso desse *framework*, uma vez que esse será o *framework* utilizado no desenvolvimento da solução proposta neste trabalho.

- O *PHPUnit* é uma ferramenta para automatizar testes de unidade na linguagem de programação *PHP*. Esse foi criado por Sebastian Bergmann, baseado no *JUnit*. Mais informações sobre esse podem ser encontradas em: *http://phpunit.de*.

2.2 Estratégias de Testes

Há inúmeras estratégias que podem ser aplicadas para testes de software, sendo que essas estratégias podem ser aplicadas em diferentes tipos de testes. Por exemplo, essas podem ser utilizadas para validar funcionalidades específicas, como um trecho de código, interfaces, integrações de processos, teste de carga, entre outros. A seguir serão descritas algumas estratégias de testes, dando ênfase na estratégia de teste de unidade, pois essa será a abordagem utilizada para o desenvolvimento desse trabalho.

2.2.1 Teste de Unidade

O teste de unidade ou teste unitário é usado na localização de erros dentro do limite interno de funções, métodos e pequenos trechos de códigos. Pode-se utilizar o teste de unidade para verificar se os métodos ou funções retornam os resultados corretos, se cada fluxo de controle está funcionando como esperado e, por fim, se a lógica de um trecho de código está coerente com os requisitos iniciais (TALES, 2009; PRESSMAN, 2006).

O teste de unidade facilita a organização do processo de teste uma vez que prioriza pequenas unidades ou módulos do programa. À medida que erros são detectados, eles podem facilmente ser corrigidos em escala menor, facilitando também a tarefa de depuração. Além disso, os testes de unidade podem ser executados simultaneamente em vários módulos do programa.

A ideia básica do teste unitário é examinar o comportamento do código sob condições variadas, baseado em dados de entrada e nos resultados esperados. As "unidades" testadas geralmente se referem a pequenos módulos de códigos como, por exemplo, funções, procedimentos, métodos ou classes. É importante destacar que o teste de unidade verifica apenas o método ou classe focada pelo programador, independente de recursos externos como, por exemplo, sistema de arquivos, banco de dados, rede, etc.

A criação dos testes de unidade pode ser efetuada antes ou após o início da fase

de desenvolvimento. A diferença entre essas duas direções é que ao se desenvolver os testes de unidade antes da codificação, o desenvolvedor focalizará mais na compreensão do problema e, consequentemente, estará ampliando a análise do problema (TALES, 2009). Assim, uma boa compreensão do problema e uma base adequada de testes de unidade, levará o desenvolvedor a programar somente o necessário evitando códigos desnecessários e podendo diminuir, em muitas vezes, a complexidade do código. A abordagem em se desenvolver os testes de unidade no final da codificação também é válida, pois permite a reexecução dos testes, garantindo que novas alterações não provoquem erros.

As abordagens ágeis para o desenvolvimento de software adotam que os testes de unidade sejam desenvolvidos antes da codificação e sejam automatizados. A automatização dos testes de unidade é proporcionada por ferramentas que facilitam a reexecução dos testes. Essas ferramentas possibilitam a execução dos testes a qualquer momento no processo de codificação, permitindo que o programador execute esses testes inúmeras vezes.

2.2.1.1 Exemplo da utilização dos testes de unidade

Nessa seção será exemplificado a utilização dos testes de unidade de forma automatizada usando o *framework* Python *Unittest* (UNITTEST, 2013). Nesse exemplo é apresentado o desenvolvimento de um software, cujo objetivo é converter valores da base hexadecimal para a base decimal. O desenvolvimento será conduzido pelo método ágil onde primeiramente tem-se a codificação dos casos de testes.

No Trecho de Código 2.3 tem-se o código referente aos casos de teste, que permitirão a validação da classe *CalculadoraSimples()*. No método *setUp()* é feito o instanciamento da classe *CalculadoraSimples()*. O método *test_conversao()* tem como finalidade validar os retornos do método *converte_hex_para_dec()*, que é responsável pela conversão dos valores passados por parâmetros, de hexadecimal para decimal. A validação do código à ser testado é feita através da execução de métodos da classe *unittest.TestCase()*. Nesse caso é utilizado o método *assertEqual()* que recebe como primeiro parâmetro o método a ser testado e como segundo parâmetro o resultado de retorno esperado da execução do código. A execução do método *assertEqual()* resulta em sucesso se o resultado da execução da função (primeiro parâmetro) é igual ao resultado esperado (segundo parâmetro), caso contrário é apresentado um erro. É utilizado ainda o método *assertFalse()*, também da classe *unittest.TestCase()*, que tem como finalidade verificar um caso onde a execução do código resulta Falso. A passagem dos parâmetros é semelhante aos do método *assertEqual()*.

Trecho de Código 2.3: Exemplo do código para automatizar os testes de unidade para a classe CalculadoraSimples.

```
1
2 #-*- coding: utf-8 -*-
3 import unittest
4 from exemplo_unitario import *
5
6 class TestUnitarios(unittest.TestCase):
7
8     def setUp(self):
9         self.calculadora = CalculadoraSimples()
10
11     def test_conversao(self):
12         self.assertEqual(self.calculadora.converte_hex_in_dec("0123456789"), [0,
               1, 2, 3, 4, 5, 6, 7, 8, 9])
13         self.assertEqual(self.calculadora.converte_hex_para_dec("abcdef"), [10,
              11, 12, 13, 14, 15])
14         self.assertEqual(self.calculadora.converte_hex_para_dec("0123456789
              abcdef"), [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
15         self.assertFalse(self.calculadora.converte_hex_para_dec("xyz"), [])
16
17 if __name__ == '__main__':
18     unittest.main()
```

Na primeira execução do Trecho de Código 2.3 é esperado que todos os testes falhem, como pode ser observado na linha 14 do Trecho de Código 2.4, que apresenta a saída "*FAILED (failures=1)*". Isso ocorre pois ainda não foi desenvolvido o método que é responsável pela conversão dos valores da classe *CalculadoraSimples()*.

Trecho de Código 2.4: Exemplo da execução do código de teste utilizando o *framework* Python *unittest*.

```
1 luciano@luciano-XPS-L421X:~/Tcc_Luciano/scripts$ python exemplo_unitario_tests.
      py
2 F
3 ======================================================================
4 FAIL: test_conversao (__main__.TestUnitarios)
5 ----------------------------------------------------------------------
6 Traceback (most recent call last):
7   File "exemplo_unitario_tests.py", line 11, in test_conversao
8     self.assertEqual(self.calculadora.converte_hex_para_dec("0123456789"), [0,
          1, 2, 3, 4, 5, 6, 7, 8, 9])
9 AssertionError: None != [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
10
11 ----------------------------------------------------------------------
12 Ran 1 test in 0.002s
13
14 FAILED (failures=1)
```

No Trecho de Código 2.5, tem-se o código da classe *CalculadoraSimples()* e do método *converte_hex_para_dec()*. Esse método faz a conversão de valores hexadecimal em decimal, recebendo como parâmetro uma *string* e retornando uma lista com as conversões ou uma lista vazia caso ocorra algum erro.

A partir da finalização do Trecho de Código 2.5, pode-se reexecutar os testes do Trecho de Código 2.3. No Trecho de Código 2.6 tem-se o resultado dos testes, onde pode-se observar que todos os testes resultaram em sucesso. A partir de uma análise mais detalhada do código observa-se que um caso de teste foi esquecido, ou seja, faltou testar a execução do método *converte_hex_para_dec()* utilizando-se letras maiúsculas como parâmetros de entrada. Como o desenvolvimento é dirigido a testes tem-se (TALES, 2009):

1. Criação de novos casos de teste
2. Execução dos casos de teste, sendo esperado que esses falhem
3. Efetuar as devidas correções no código
4. Reexecução dos testes e verificar se os mesmos retornam sucesso

Trecho de Código 2.5: Exemplo de uma simples calculadora que converte um número de Hexadecimal em Decimal.

```
1 #-*- coding: utf-8 -*-
2
3 class CalculadoraSimples(object):
4
5     def converte_hex_para_dec(self, dados_hex):
6         """
7             Efetua conversoes de Hexadecimal em Decimal
8             Retorna uma lista<list> com as convercoes, senao retorna uma lista
                  vazia[]
9         """
10        numeros = ['0','1','2','3','4','5','6','7','8','9']
11        retorno = []
12        for dado in dados_hex:
13            if dado in numeros:
14                retorno.append(int(dado))
15            elif dado == "a":
16                retorno.append(10)
17            elif dado == "b":
18                retorno.append(11)
19            elif dado == "c":
20                retorno.append(12)
21            elif dado == "d":
22                retorno.append(13)
23            elif dado == "e":
24                retorno.append(14)
25            elif dado == "f":
26                retorno.append(15)
27            else:
28                return []
29        return retorno
```

Trecho de Código 2.6: Exemplo da execução do código de teste utilizando o *framework* Python *unittest*, tendo como saída um caso de sucesso.

```
1 luciano@luciano-XPS-L421X:~/Tcc_Luciano/scripts$ python exemplo_unitario_tests.
      py
2 .
3 ----------------------------------------------------------------------
4 Ran 1 test in 0.000s
5
6 OK
```

O Trecho de Código 2.7 ilustra o caso de teste para validar se o método *converte_hex_para_dec()* aceita letras maiúsculas. Após a nova execução dos testes, é esperado uma falha, como é mostrado no Trecho de Código 2.8.

Trecho de Código 2.7: Exemplo do código para automatizar os testes de unidade para a classe CalculadoraSimples, com o caso de teste para validar letras maiúsculas.

```
1
2 #-*- coding: utf-8 -*-
3 import unittest
4 from exemplo_unitario import *
5
6 class TestUnitarios(unittest.TestCase):
7
8     def setUp(self):
9         self.calculadora = CalculadoraSimples()
10
11     def test_conversao(self):
12         self.assertEqual(self.calculadora.converte_hex_para_dec("0123456789"),
               [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
13         self.assertEqual(self.calculadora.converte_hex_para_dec("abcdef"), [10,
               11, 12, 13, 14, 15])
14         self.assertEqual(self.calculadora.converte_hex_para_dec("0123456789
               abcdef"), [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
15         self.assertFalse(self.calculadora.converte_hex_para_dec("xyz"), [])
16         self.assertEqual(self.calculadora.converte_hex_para_dec("ABCDEF"), [10,
               11, 12, 13, 14, 15])
17
18 if __name__ == '__main__':
19     unittest.main()
```

Trecho de Código 2.8: Exemplo da execução do código de teste utilizando o *framework* Python *unittest*. Inserção de um novo caso de teste. Caso com falha.

```
1 luciano@luciano-XPS-L421X:~/Tcc_Luciano/scripts$ python exemplo_unitario_tests.
      py
2 F
3 ======================================================================
4 FAIL: test_conversao (__main__.TestUnitarios)
5 ----------------------------------------------------------------------
6 Traceback (most recent call last):
7   File "exemplo_unitario_tests.py", line 15, in test_conversao
8     self.assertEqual(self.calculadora.converte_hex_para_dec("ABCDEF"), [10, 11,
          12, 13, 14, 15])
9 AssertionError: Lists differ: [] != [10, 11, 12, 13, 14, 15]
```

```
10
11 Second list contains 6 additional elements.
12 First extra element 0:
13 10
14
15 - []
16 + [10, 11, 12, 13, 14, 15]
17
18 ----------------------------------------------------------------------
19 Ran 1 test in 0.001s
20
21 FAILED (failures=1)
```

No Trecho de Código 2.9, tem-se o método *converte_hex_ para_dec()* alterado (linha 12) de forma que seja aceito letras maiúsculas. Novamente, são executados os testes do Trecho de Código 2.7, sendo que após a execução verifica-se que os testes retornaram sucesso (2.10). Com a execução dos testes bem sucedida, o processo de desenvolvimento do software de exemplo pode ser considerado concluído.

Trecho de Código 2.9: Exemplo de uma simples calculadora que converte Hexadecimal em Decimal, aceitando letras maiúsculas e minúsculas como parâmetro.

```
1  #-*- coding: utf-8 -*-
2
3  class CalculadoraSimples(object):
4
5      def converte_hex_para_dec(self, dados_hex):
6          """
7              Efetua conversoes de Hexadecimal em Decimal
8              Retorna uma lista<list> com as convercoes, senao retorna uma lista
                   vazia[]
9          """
10         numeros = ['0','1','2','3','4','5','6','7','8','9']
11         retorno = []
12         dados_hex = dados_hex.lower()
13         for dado in dados_hex:
14             if dado in numeros:
15                 retorno.append(int(dado))
16             elif dado == "a":
17                 retorno.append(10)
18             elif dado == "b":
19                 retorno.append(11)
20             elif dado == "c":
21                 retorno.append(12)
22             elif dado == "d":
23                 retorno.append(13)
24             elif dado == "e":
25                 retorno.append(14)
26             elif dado == "f":
27                 retorno.append(15)
28             else:
29                 return []
30         return retorno
```

Trecho de Código 2.10: Exemplo da execução do código de teste utilizando o *framework* Python *unittest*. Com a correção para aceitar letras maiúsculas.

```
1
2 luciano@luciano-XPS-L421X:~/Tcc_Luciano/scripts$ python exemplo_unitario_tests.
      py
3 .
4 ----------------------------------------------------------------------
5 Ran 1 test in 0.000s
6
7 OK
```

2.2.2 Teste de Integração

Os testes de integração são utilizados para validar a integração de subsistemas e são aplicados após os testes de unidade. O uso dessa técnica na integração de grandes sistemas não é muito eficaz, pela dificuldade em efetuar correções caso sejam encontrados erros.

Para facilitar a localização de erros na integração é usado uma abordagem de integração incremental, que consiste em fazer a integração entre pequenos módulos, ou seja, para a integração entre três módulos, inicialmente integra-se dois módulos. Posteriormente, são executados os testes, sendo que, caso esses sejam executados com sucesso o terceiro módulo é integrado. Após a integração do terceiro módulo executa-se novamente as rotinas de testes de integração (PRESSMAN, 2006).

2.2.3 Teste de Regressão

Esta estratégia auxilia na localização de erros quando agregam-se mudanças ou correções de erros ao software, pois essas ações podem ocasionar a introdução de erros em funcionalidades já validadas. Se forem localizados erros nesta tarefa, é dito que sistema regrediu e inicia-se a fase de correção dos erros antes da liberação do sistema (DELAMARO; MALDONADO; JINO, 2007).

A aplicação dessa estratégia consiste em reexecutar os casos de testes ao final do desenvolvimento das alterações para garantir que as novas mudanças não ocasionaram erros ao software. Essa estratégia pode ser executada de forma manual ou utilizando-se ferramentas automatizadas que facilitam o processo de teste (SOMMERVILLE, 2003).

2.2.4 Teste de Sistema

Este teste constitui o processo mais difícil, uma vez que é aplicado a todo o software. Segundo (MYERS, 2004), os casos de testes para esta técnica devem ser baseados nos objetivos do software, juntamente com a documentação do usuário. Essa abordagem de união entre os objetivos do software mais a documentação do usuário, aproxima-se de uma execução real do sistema.

O teste de sistema apresenta como principal objetivo demonstrar que o sistema implementa todos os requisitos iniciais. Este teste é executado simulando um usuário final, sendo que o ambiente de teste é o mesmo que o usuário utilizaria em seu ambiente de trabalho, aumentando assim a probabilidade dos erros serem descobertos.

2.2.5 Teste de Desempenho

Os testes de desempenho são testes que avaliam a performance do sistema, podendo ser executados ao longo de todo o desenvolvimento do software. Este tipo de teste pode ser empregado nos módulos do software à medida que eles vão sendo integrados e também ao final do desenvolvimento, como um todo. Nesse tipo de teste, podem ser testados por exemplo, o número de requisições que determinado módulo suporta, o uso de recursos de hardware, o tempo de uma consulta em um banco de dados, entre outros (PRESSMAN, 2006).

2.3 Considerações Finais

Este capítulo apresentou uma visão geral sobre testes de software. Foram descritas as principais técnicas de testes de software, apresentando como objeto de estudo as técnicas estruturais, funcionais e automatizados. Também foram apresentadas as principais estratégias de teste, tais como, teste de unidade, de integração, de regressão, de sistema e por fim teste de desempenho.

Na técnica de teste estrutural (caixa branca) é necessário que o desenvolvedor tenha conhecimento da estrutura interna do código, por outro lado, a técnica funcional (caixa preta) não aborda o funcionamento interno do código, verificando apenas se o sistema desenvolvido implementa as funcionalidades que foram definidas nos requisitos.

Como foi definido anteriormente, o teste de unidade tem como principal objetivo localizar erros em pequenos trechos de códigos. O teste de integração visa validar a integração entre subsistemas. O teste de regressão é utilizado quando efetua-se alterações ou correções de erros ao software. O teste de sistema é aplicado na finalização do software, tendo por objetivo validar se as funcionalidades descritas nos requisitos iniciais foram de fato implementadas. Por fim, no teste desempenho a finalidade é avaliar a performance global de todo o software.

Levando em consideração que o problema deste trabalho é definido em termos da validação das especificações a partir dos requisitos iniciais e baseado nos estudos das características dos testes, a técnica de teste funcional aliada à estratégia de teste unitário, são os mais adequadas para a solução do problema.

3 FUNCIONAMENTO DE WORKFLOWS EM PLONE

Neste capítulo são apresentados os conceitos fundamentais sobre o sistema de *workflow* do Plone e as principais tecnologias envolvidas no desenvolvimento de portais Web usando Plone. Os principais conceitos, o funcionamento e a gestão de um *workflow* Plone são abordados neste capítulo para compreensão geral.

3.1 Python

O Python é uma linguagem de programação de alto nível, dinâmica, fortemente tipada, totalmente orientada a objetos, interativa, interpretada, multiplataforma e de software livre. O seu desenvolvimento foi iniciado por Guido van Rossum, em 1990, tendo como principal finalidade o ensino de programação. O seu criador nomeou a linguagem de Python, por ser um grande fã do seriado de comédia *Monty Python's Flying Circus*.

Atualmente, a linguagem Python é mantida pela *Python Software Foundation*, sendo muito utilizada por grandes companhias, no desenvolvimento de softwares em diversos segmentos, tais como, aplicações Web, *Desktop*, *mobile*, games, entre outros. (ROSSUM, 2005; BORGES, 2010; LUTZ; ASCHER, 2007; VENNERS, 2013).

A linguagem de programação Python tem uma vasta biblioteca padrão, denominada, na filosofia Python de "baterias inclusas". Essas bibliotecas disponibilizam funcionalidades que permitem operar sobre estruturas de dados nativas, tais como listas, dicionários, tuplas e inúmeras outras. Além disso, o Python possibilita a instalação de novos módulos ou bibliotecas que acrescentam novas funcionalidades.

A documentação da linguagem Python é ampla, sendo que um grande percentual dela é distribuída de forma gratuita. Além disso, a comunidade de desenvolvedores em Python é muito numerosa, sendo que no Brasil a comunidade Python disponibilizou várias matérias sobre aprendizagem em Python (PYTHONBRASIL, 2013).

3.2 *Zope* (*Z Object Publishing Environment*)

O *Zope* é um servidor de aplicações Web escrito quase que totalmente na linguagem Python, sendo que alguns módulos específicos foram desenvolvidos na linguagem C. O desenvolvimento do Zope foi iniciado em 1996 por Jim Fulton e, atualmente, é mantido pela *CTO* (*Zope Corporation*). O Zope é software livre registrado sob uma licença *ZPL* (*Zope Public License*), podendo ser instalado em diversas plataformas tais como Unix, GNU/Linux, entre outras.

No Zope, o conteúdo a ser publicado na Web é obtido a partir de objetos escritos em Python que são armazenados em um banco de dados chamado *ZODB* (*Zope Object Data Base*). Esses objetos podem ser criados ou gerenciados via um interface Web, chamada *ZMI* (*Zope Management Interface*), que é apresentada na Figura 3.1.

A interface *ZMI* é o ambiente de gestão do Zope, através dela é possível efetuar configurações no Zope, manipular objetos Zope e configurar aplicações Web. Destaca-se que o Zope não pode ser utilizado para armazenar conteúdos estáticos, ou seja, esse não permite a disponibilização de arquivos que estão no sistema de arquivos do servidor. Ao invés disso, faz a geração de um arquivo *HTML* (*HyperText Markup Language*) com o respectivo conteúdo armazenado no banco de dados *ZODB*, sendo que esse *HTML* pode ser interpretado por qualquer navegador Web (ZOPE, 2013; COOPER, 2004).

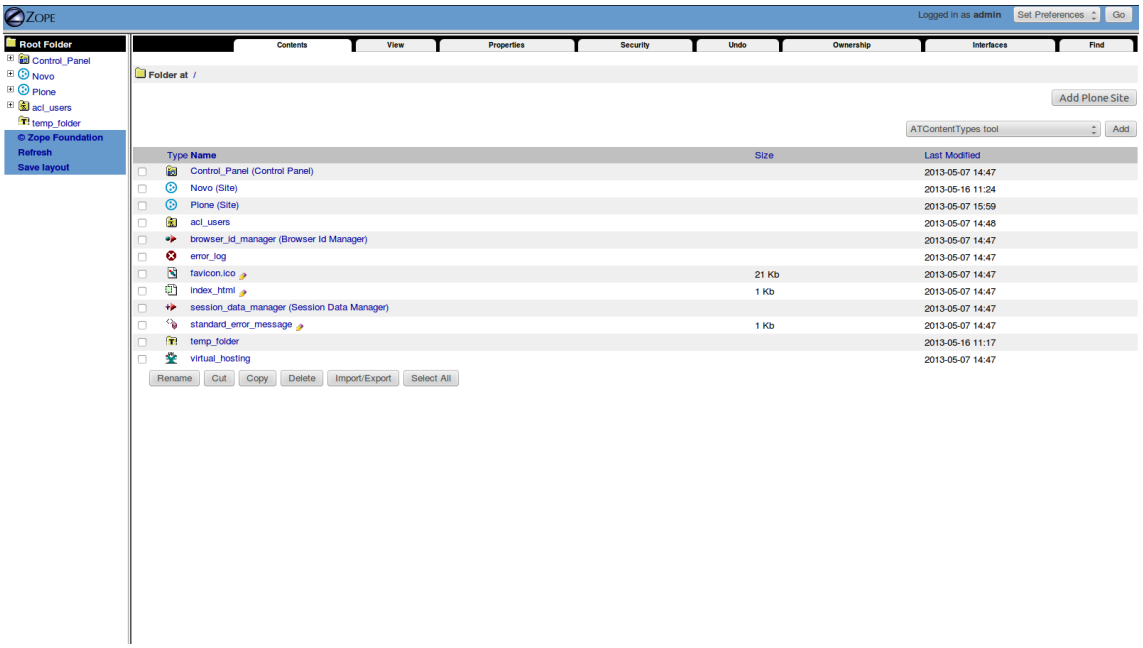

Figura 3.1: Interface de *ZMI*

3.3 Plone

O Plone é um Sistema de Gerenciamento de Conteúdo *CMS* (*Content Management System*), que é executado como uma aplicação no servidor ZOPE. Dessa forma, é possível a criação de diferentes portais Plone em uma única instância ZOPE, sendo que cada portal Plone tem suas próprias funcionalidades.

O desenvolvimento do Plone foi iniciado por Alexander Limi e Alan Runyan em 2000, sendo que, atualmente, a *Plone Foundation* detém os direitos sobre o código fonte, que é disponibilizado livremente sob a licença *GPL* (*General Public License*).

O Plone é utilizado para criação de portais Web, sistemas Web ou como ferramenta de trabalho colaborativo. Esse possui vários tipos de conteúdos nativos, tais como um sistema de publicação de notícias, *upload* de arquivos, imagens, páginas, eventos, links, pastas e coleções. Também há nativamente no Plone funcionalidades, tais como cadastro de usuários, regra de permissões, *workflows* para controle de acesso à conteúdo, robusto sistema de busca, suporte a multilínguas, entre outros. Um ponto importante é que o Plone é mantido por voluntários ao redor do mundo, tendo uma vasta documentação, listas de discussões e uma conferência anual (ASPELI, 2007; PLONE, 2013).

O Plone é desenvolvido para ser acessado via Web, sendo compatível com os principais navegadores. Um exemplo de um portal Plone padrão é mostrado na Figura 3.2. O Plone possibilita a criação de produtos que acrescentam novas funcionalidades à um portal Plone. Esses produtos, são portáveis, podendo ser instalados em qualquer portal Plone, para tanto, é necessário respeitar a versão do Plone para qual foi desenvolvido. Entre os produtos disponibilizados pela *Plone Foundation*, destaca-se produtos para integração com redes sociais, fórum de discussão, sistema de enquete, novos temas para *layout*, entre vários outros. Esses produtos são distribuídos de forma gratuita e podem ser acessados no site do Plone (PRODUTOS, 2013).

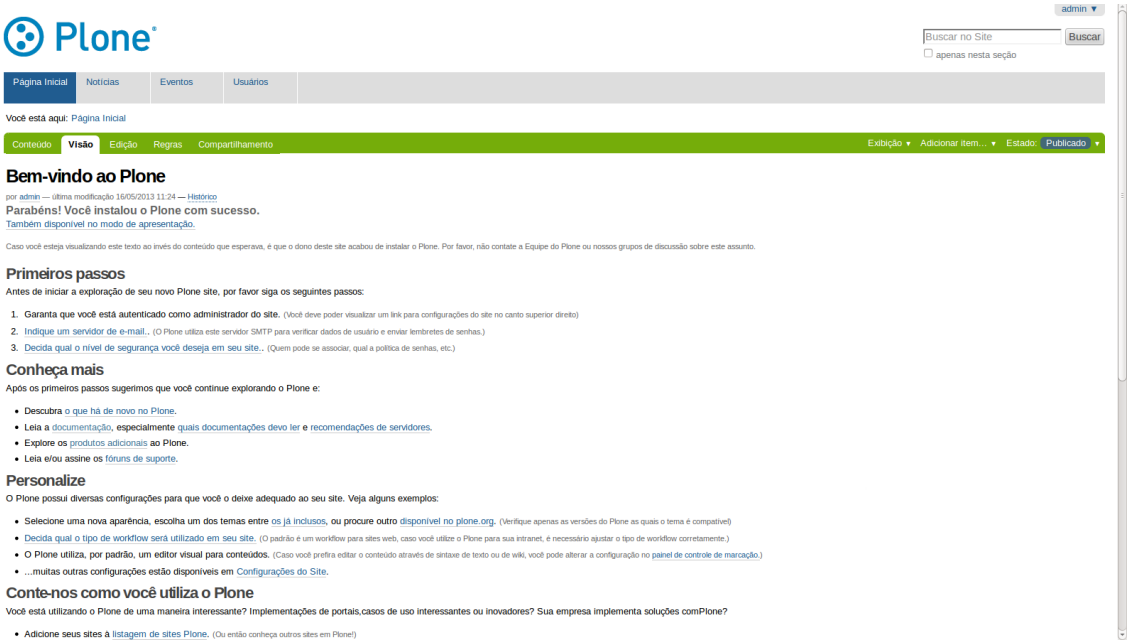

Figura 3.2: Plone

3.4 Workflow do Plone

O sistema de *workflow* do Plone é uma ferramenta de segurança que executa o controle de acesso aos objetos e que também controla o ciclo de vida de um conteúdo. O *workflow* pode ser representado como um grafo, onde os nodos representam os estados de acessibilidade a um objeto e as arestas as transições entre esses estados. Um usuário só pode executar uma transição entre estados através de um conjunto de permissões definidos por papéis específicos. Por exemplo, em um jornal tem-se um determinado usuário com permissão de adicionar notícias em um portal Plone, mas o mesmo não possui permissão para publicar as notícias em determinada área do portal. Desta forma, as notícias publicadas por esse ficam em um estado do *workflow* "aguardado publicação", por exemplo, podendo ser publicada apenas por um usuário que possui a permissão de "publicação" (ALBA, 2010).

O *workflow* padrão do Plone possui três estados que são: privado, revisão pendente e publicado. A Figura 3.3 apresenta as transições para cada estado, sendo que a partir do estado privado o usuário pode publicar o conteúdo através da transição publicar ou enviar para publicação.

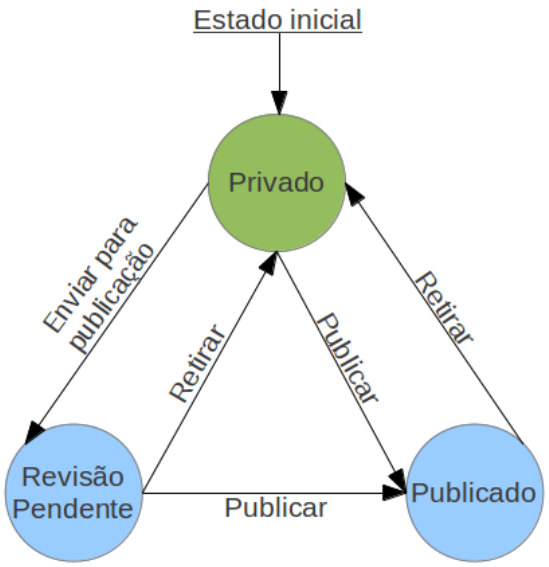

Figura 3.3: Worlflow padão Plone

É importante destacar que cada *workflow* gerencia um conjunto de permissões para a mudança de estados, aliando o grafo de estados do objeto às transição usando papéis. No exemplo da Figura 3.3 apenas os usuários com os papéis de *Owner*, *Editor*, *Reviewer*, *Manager* e *Site Administrator* possuem a permissão de executar as transições para mudança de estados.

3.4.1 Funcionamento do *workflow* em um portal Plone

Nessa seção será apresentado um exemplo do funcionamento do *workflow* padrão do Plone. O *workflow* padrão é chamado *Simple Publication Workflow* e é utilizado nos tipos de conteúdos nativos do Plone. Esse *workflow* é composto de estados e transições pré-definidas, como pode ser observado na Figura 3.3. Nos próximos parágrafos será ilustrada a criação de um conteúdo do tipo "Pasta" e as opções de transições de *workflow*.

O processo é iniciado com a escolha do tipo de conteúdo a ser criado, nesse caso o conteúdo tipo "Pasta" (Figura 3.4). No próximo passo é feito o preenchimento dos campos para o conteúdo "Pasta". A Figura 3.5 apresenta o conteúdo criado, o qual está, inicialmente, no estado "Privado" do *workflow*. Todos os conteúdos nativos do Plone possuem esse estado como padrão quando criados.

Figura 3.4: Escolha do tipo de conteúdo a ser criado em um portal Plone.

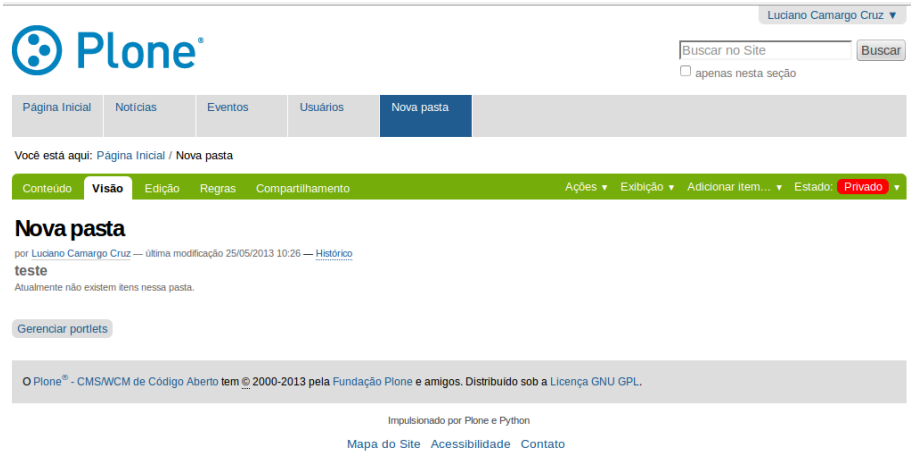

Figura 3.5: Conteúdo tipo “Pasta” com estado “Privado” do *workflow* como padrão.

A Figura 3.6 apresenta o menu “Estado”, onde pode ser observado o estado “Privado” para o tipo de conteúdo “Pasta”. No estado “Privado” somente o criador e os usuários que possuem determinados papéis podem ter acesso ao conteúdo. A Tabela 3.1 demostra uma definição básica das ações que cada papel pode exercer sobre o conteúdo. Esses papéis podem ser atribuídos aos usuários de uma forma global ou local. De forma global o usuário pode executar as ações em todo o portal Plone, isso pode ser configurado na opção “Configuração do Site” no item “ Usuários e Grupos”. De forma local as ações podem ser efetuadas em um respectivo contexto do portal, sendo que essas configurações podem ser habilitadas na aba “Compartilhamento” de cada conteúdo.

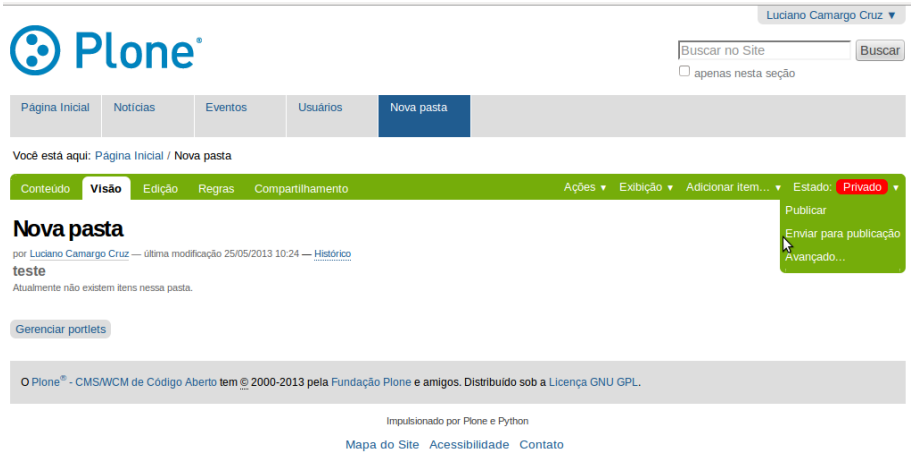

Figura 3.6: Menu para troca de “Estados”.

Tabela 3.1: Definições básicas das ações para cada tipo de papel

Papéis	Alterar estado	Visualizar	Adicionar	Editar
Leitor		X		
Revisor	X			
Contribuidor		X	X	
Editor	X	X	X	X

O menu “Estado” (Figura 3.6) possui um seletor com opções de troca de estado de *workflow*. Sendo que essas transições de estados podem ser efetuadas de acordo com a Figura 3.3. Por exemplo, na Figura 3.6 é selecionado a transição “Publicar”, sendo atribuído o estado “Publicado” ao conteúdo. O conteúdo no estado “Publicado” (Figura 3.7) passa a ser visível para todos usuários logados ou não no portal Plone. Uma transição entre os estados do *workflow* para um determinado conteúdo Plone, pode ser realizada em qualquer momento, basta o usuário possuir a devida permissão.

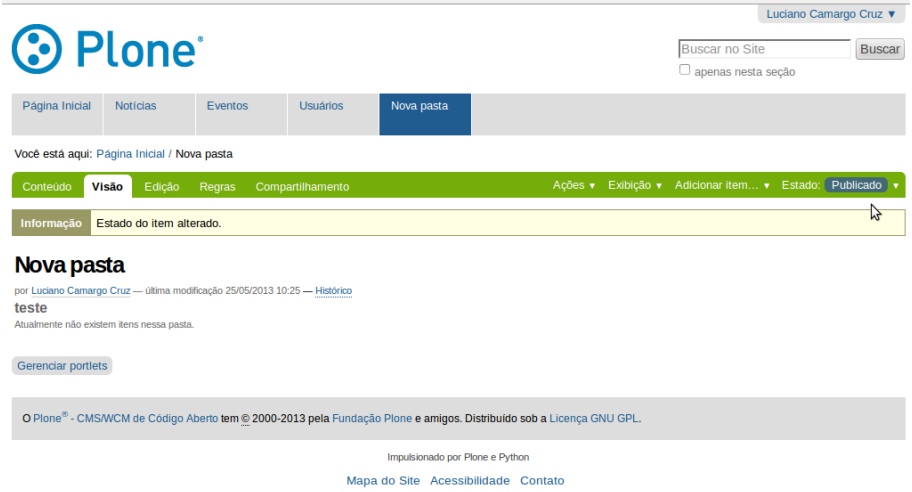

Figura 3.7: Conteúdo com estado “Publicado”.

3.4.2 Gerenciando *Workflows* no Plone

Embora definido como um grafo de fluxo, o Plone não fornece uma interface gráfica para gerenciamento dos *workflows*, tornando a tarefa manual de criação mais propensa a erros humanos. As principais entidades do *workflow*: estados, transições, permissões e papéis são manipuladas na própria ZMI através da ferramenta *portal_workflow*. A Figura 3.8 apresenta alguns dos *workflows* disponíveis em um portal Plone nativo.

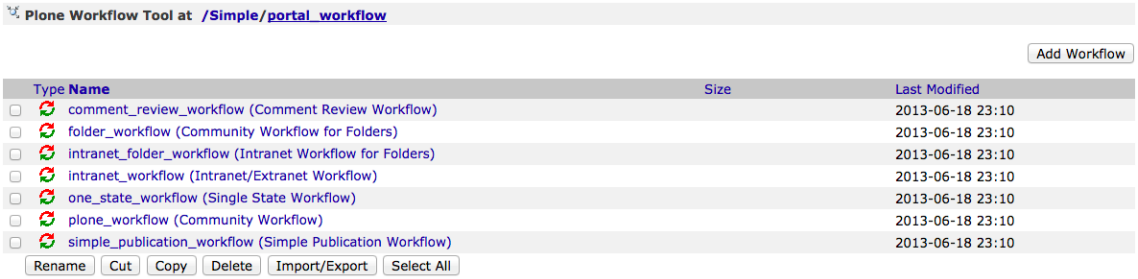

Figura 3.8: Exemplo de *workflows* nativos do Plone

Para cada *workflow*, a ferramenta *portal_workflow* oferece suporte para gestão de estados, transições, permissões e papéis. A Figura 3.9 apresenta um exemplo de tratamento de estados do *workflow simple_publication_workflow*.

Figura 3.9: Exemplo de tratamento de estados de um *workflow*

Nesta interface, cada estado possui transições associadas. As transições são acessadas através dos links oferecidos na ferramenta, por onde pode-se acessar suas propriedades. É possível, por exemplo, acessar um determinado estado, *published*, e acessar as permissões padrão disponíveis para este estado no *workflow*. A Figura 3.10 ilustra o conjunto de permissões e os papéis associados que podem executá-las sobre a transição de estado *published* no *simple_publication_workflow*. É a partir desta interface de gerenciamento que o desenvolvedor define o comportamento de cada transição de acordo com o que foi especificado na análise de requisitos. Usando esta

ferramenta também é possível criar uma grande combinação de propriedades que definem o comportamento da transição e de sua respectiva permissão. É possível adicionar ou remover mais permissões, associar os papéis que podem executá-las, criar ou remover papéis, entre outras ações.

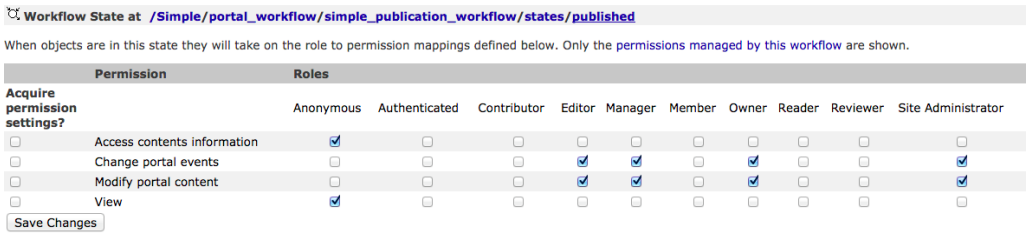

Figura 3.10: Conjunto de permissões e os papéis de um *workflow*

É possível também acessar os atributos de qualquer transição seja a partir da interface de estados ou, ainda, do menu de transições do *portal_workflow*. A Figura 3.11 ilustra a visualização da matriz de transições. Cada transição possui um conjunto de atributos associados que são manipulados pelo desenvolvedor. Através dessa é possível determinar o estado atual, o estado de destino, o disparador do evento de transição, quais permissões são requeridas para seu disparo, *scripts* livremente codificados (pedaços de código) que podem ser executados antes ou depois do disparo da transição, entre outros. É possível também adicionar novas ou remover transições.

Figura 3.11: Matriz de transições de um *workflow*

Um atributo importante de cada permissão é sua *guard_condition*, que é um

conjunto de validadores que identifica se uma permissão, papel, grupo ou uma expressão *TALES* (*Template Attribute Language Expression*) pode estar disponível ao usuário corrente ou não.

Como visto anteriormente, não há uma metodologia ou ferramenta metodológica específica para construção do *workflow*. Dependendo da aplicação, do domínio, do problema e dos requisitos, as especificações de *workflow* para segurança do conteúdo serão diferentes. Segundo (ALBA, 2010) os principais passos envolvidos no desenvolvimento de *workflows* Plone são:

- Planejamento inicial baseado nos requisitos iniciais para definir o *workflow* desejado, pelo menos determinando atributos gerais de estados, transições, ações que possam ser disparadas e papéis envolvidos;
- Inicialmente usar servidores de testes em ambientes controlados;
- Se possível partir de um *workflow* nativo existente e fazer as adaptações;
- Aproveitar ferramentas de *debugging* e checar a consistência do mapa de permissões desejado;
- Testar manualmente o *workflow* checando cada caso de uso definido nas especificações, monitorando a correta execução das transições e permissões.

Este último passo é, certamente, o mais importante e determinante no funcionamento do *workflow* desejado. Segundo (ALBA, 2010), para executar testes em *workflows* Plone é preciso desenvolver uma bateria de casos de uso para que cada transição, estado e permissão do *workflow* seja testado conforme os requisitos definidos inicialmente. Nestes casos, é fácil cometer um erro relacionando papéis e permissões, pois o grafo de estados não representa diretamente as permissões.

3.5 Considerações Finais

Este capítulo apresentou o funcionamento de *workflow* em Plone, onde foi detalhado o grafo de estados e transições que permite a modificação dos conteúdos criados em um portal Plone. Neste capítulo foram apresentadas as tecnologias utilizadas no desenvolvimento de portais Plone, como a linguagem Python e o servidor de aplicações Zope. A definição de *workflow* é feita usando um grafo de transições em que os estados são definidos pelo especificador e, separadamente, cada *workflow* deve ser representado através de um conjunto de permissões e papéis que o executem.

O *workflow* Plone é definido através de um grafo de estados em que estados são atribuídos a um conteúdo em função da regra de transições entre os estados. Como qualquer diagrama de estados, o teste básico pressupõe que determinados estados alvos sejam atingidos a partir de estados iniciais disparando determinadas transições. Existem atributos a serem validados para cada transição ser disparada que dependem de permissões, papéis, *guard_conditions* e outros atributos.

No entanto, neste ponto da definição não existe um código para ser testado, mas sim um conjunto de definições que foram construídos a partir de uma ferramenta de manipulação de transições de estados, neste caso a ferramenta *portal_workflow*. Além disso não é possível testar as saídas em função das entradas, pois os códigos não foram definidos, sendo possível apenas simular o comportamento esperado da mudança de estados através da validação dos casos de uso definidos na especificação.

A partir destes estudos, identificou-se que o problema esta associado à estrutura de definição de um *workflow*, a qual é desenvolvida no próprio Zope, não existindo uma técnica de teste específica que capture automaticamente a validação do comportamento de mudança de estados com o requisito inicial.

4 WORKFLOW.VALIDATION - FRAMEWORK DE TESTES PARA WORKFLOW PLONE

Neste capítulo será apresentado o desenvolvimento da solução para o *framework* de testes de *workflows* Plone (*workflow.validation*). O principal objetivo deste *framework* é permitir a automatização dos testes funcionais de um *workflow* Plone. Neste os requisitos iniciais do *workflow* são mapeados em uma estrutura que permite o tratamento das verificações dos comportamentos esperados.

A ideia principal é propor uma modificação na estratégia de teste unitário, para que seja possível validar, ao invés do código, as definições dos requisitos iniciais e as definições das regras de *workflow*, já finalizadas, contidas em um arquivo XML. É usado também a técnica de teste funcional para verificar se ao final do desenvolvimento das regras de *workflow* estão de acordo com as definições iniciais.

Como já foi apresentado no Capítulo 1, o Plone não apresenta recursos que permitam armazenar os requisitos iniciais do *workflow*. Isso dificulta a geração de testes funcionais de forma automática, já que os requisitos não são conhecidos. Desta forma, o *framework* proposto está subdividido em:

- um produto Plone com interfaces para entrada dos dados, exibição das definições das regras para o *workflow*;
- um esquema de exportação das definições iniciais em um arquivo XML;
- um esquema de exportação de um conjunto de testes unitários, utilizando a ferramenta *unittest*;
- execução do esquema de testes de forma automatizada.

4.1 Caso de uso

A Figura 4.1 ilustra o caso de uso global do *framework*. O papel analista representa o projetista e especificador do *workflow* que irá definir a estrutura de papéis, permissões e estados a partir da análise de requisitos da aplicação em desenvolvimento. O papel Desenvolvedor representa o programador que desenvolverá a implementação do *workflow*. Nas próximas seções cada caso de uso será apresentado em detalhes.

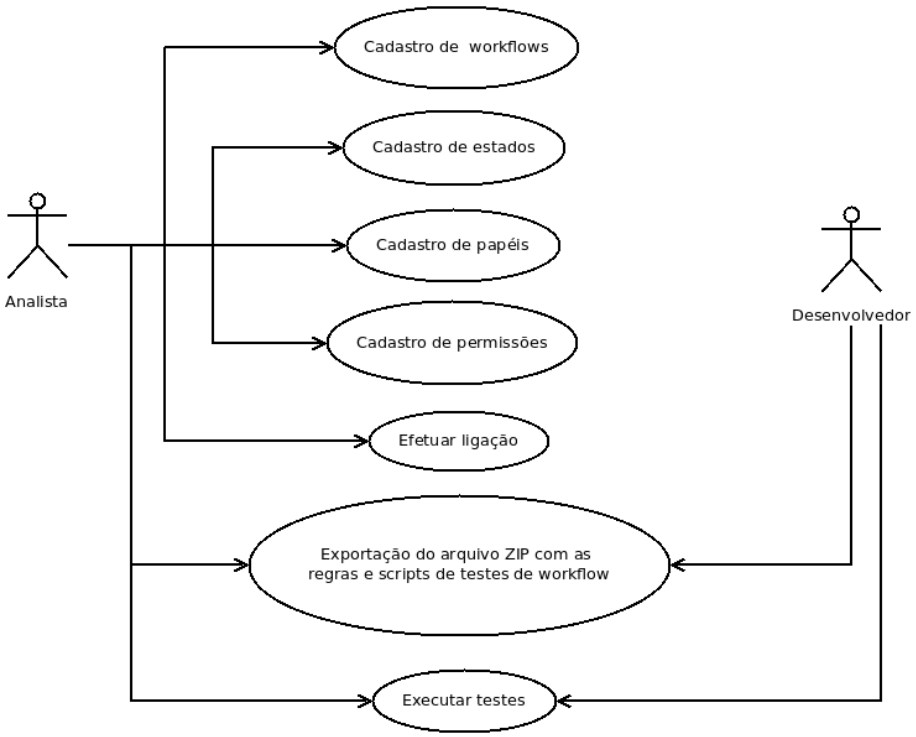

Figura 4.1: Diagrama de caso de uso (conforme (LARMAN, 2002))

4.1.1 Cadastro de *workflows*

A Tabela 4.1 descreve como um usuário efetua o cadastramento dos dados de um *workflow* no sistema (*workflow.validation*). Já Figura 4.2 apresenta a tela de inserção de dados.

Tabela 4.1: Cadastro de *workflows*

Caso de Uso:	Cadastro de *workflows*
Objetivo:	Cadastrar as informações de *workflows*
Descrição:	O usuário, depois de logar-se no portal, acessa a área de cadastro de *workflow*, informando o título. O sistema valida permissão do usuário e as informações do formulário, efetua a persistência dos dados a partir das informações relacionadas.
Atores:	Analista
Pré-condição:	Usuário válido autenticado ao Portal com permissão de *manager*
Pós-condição:	*Workflow* inserido
Fluxo Principal:	1. Usuário informa os dados do *workflow* (título) 2. Sistema confirma o cadastramento do *workflow*
Fluxos Alternativos:	1. Usuário consulta o *workflow* (a) Sistema apresenta os dados do *workflow* 2. Usuário altera os dados do *workflow* (a) Sistema confirma as alterações do *workflow*

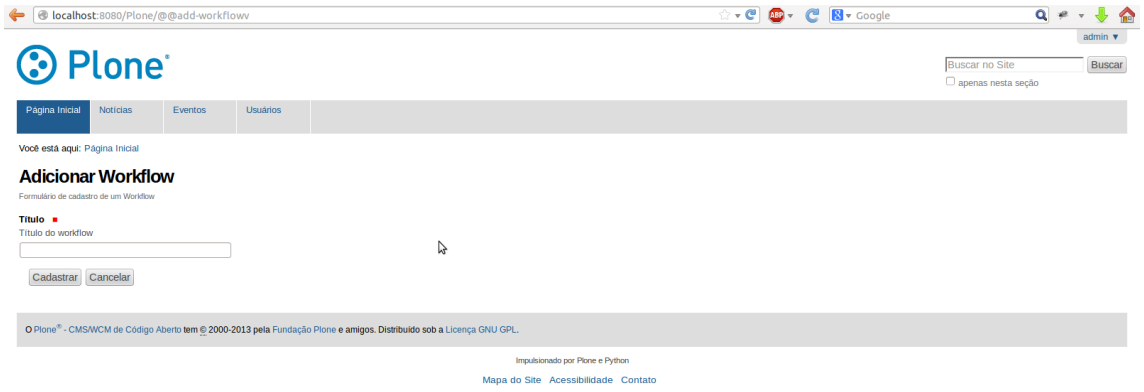

Figura 4.2: Tela de cadastro de *workflow*

4.1.2 Cadastro de estados

A Tabela 4.2 demonstra como um usuário efetua o cadastro dos dados de estado no sistema (*workflow.validation*). A Figura 4.3 apresenta a tela de inserção de dados.

Tabela 4.2: Cadastro de estados

Caso de Uso:	Cadastro de estados
Objetivo:	Cadastrar as informações de estados
Descrição:	O usuário, depois de logar-se no portal, acessa a área de cadastro de estados, selecionando o *workflow* e informando um título. O sistema valida permissão do usuário e as informações do formulário, efetua a persistência dos dados a partir das informações relacionadas.
Atores:	Analista
Pré-condição:	Usuário válido autenticado ao Portal com permissão de *manager*
Pós-condição:	Estados criados
Fluxo Principal:	1. Usuário informa os dados do estado (*workflow*, título) 2. Sistema confirma o cadastramento do estado
Fluxos Alternativos:	1. Usuário consulta o estado (a) Sistema apresenta os dados do estado 2. Usuário altera os dados do estado (a) Sistema confirma as alterações do estado

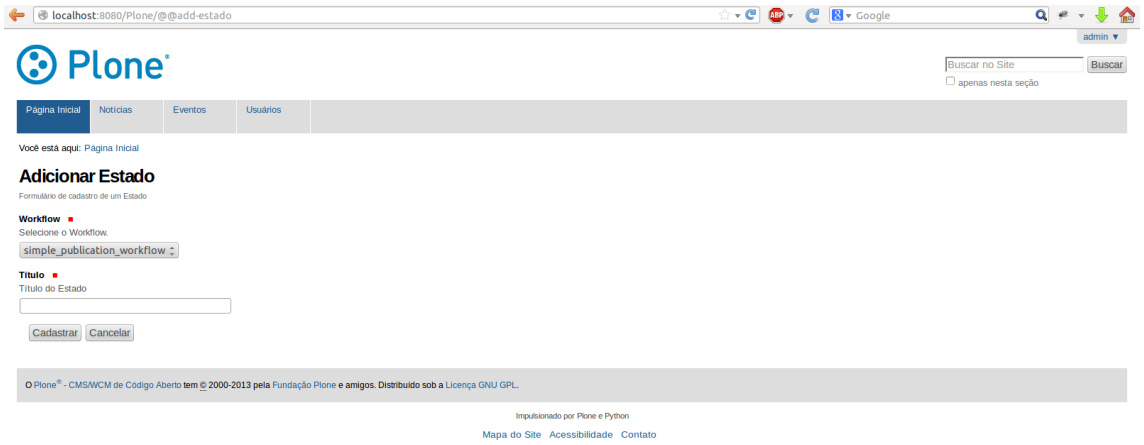

Figura 4.3: Tela de cadastro de estados

4.1.3 Cadastro de papéis

A Tabela 4.3 demonstra como um usuário efetua o cadastro dos dados do papel no sistema (*workflow.validation*). A Figura 4.4 apresenta a tela de inserção de dados.

Tabela 4.3: Cadastro de papéis

Caso de Uso:	Cadastro de papéis
Objetivo:	Cadastrar as informações de papéis
Descrição:	O usuário, depois de logar-se no portal, acessa a área de cadastro de papéis, informando um título. O sistema valida permissão do usuário e as informações do formulário, efetua a persistência dos dados a partir das informações relacionadas.
Atores:	Analista
Pré-condição:	Usuário válido autenticado ao Portal com permissão de *manager*
Pós-condição:	Papel criado
Fluxo Principal:	1. Usuário informa os dados do papel (título) 2. Sistema confirma o cadastramento do papel
Fluxos Alternativos:	1. Usuário consulta o papel (a) Sistema apresenta os dados do papel 2. Usuário altera os dados do papel (a) Sistema confirma as alterações do papel

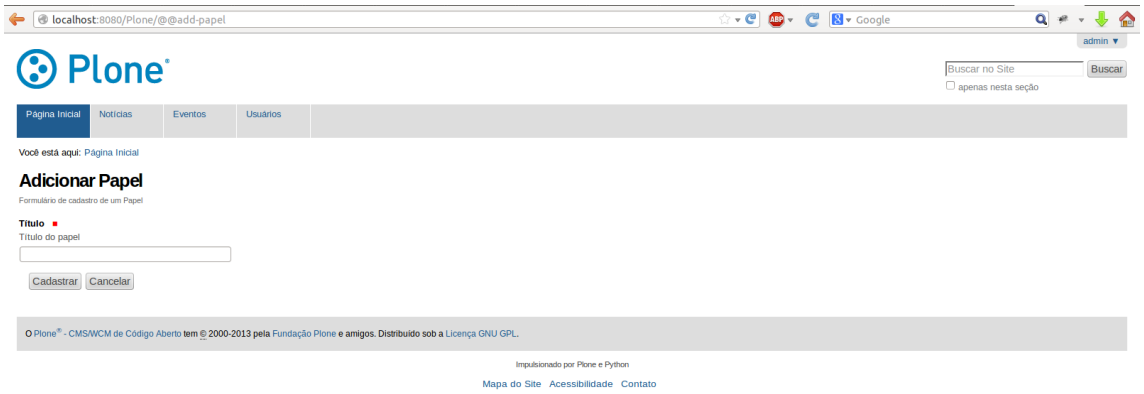

Figura 4.4: Tela de cadastro de papéis

4.1.4 Cadastro de permissões

A Tabela 4.4 demonstra como um usuário efetua o cadastro dos dados de permissões no sistema (*workflow.validation*). A Figura 4.5 apresenta a tela de inserção de dados.

Tabela 4.4: Cadastro de permissões

Caso de Uso:	Cadastro de permissões
Objetivo:	Cadastrar as informações de permissões
Descrição:	O usuário, depois de logar-se no portal, acessa a área de cadastro de permissões, informa um título, seleciona ou não uma permissão. O usuário pode cadastrar uma permissão sem vincular a uma respectiva permissão padrão Plone. Esse vínculo com a permissão do Plone é utilizada no conteúdo do arquivo XML a ser gerado, com isso o usuário não precisa informar o nome correto da permissão do Plone em inglês, facilitando o entendimento.
Atores:	Analista
Pré-condição:	Usuário válido autenticado ao Portal com permissão de *manager*
Pós-condição:	Permissão criada
Fluxo Principal:	1. Usuário informa os dados da permissão (título, tipo) 2. Sistema confirma o cadastramento da permissão
Fluxos Alternativos:	1. Usuário consulta as permissões (a) Sistema apresenta os dados da permissão 2. Usuário altera os dados da permissão (a) Sistema confirma as alterações da permissão

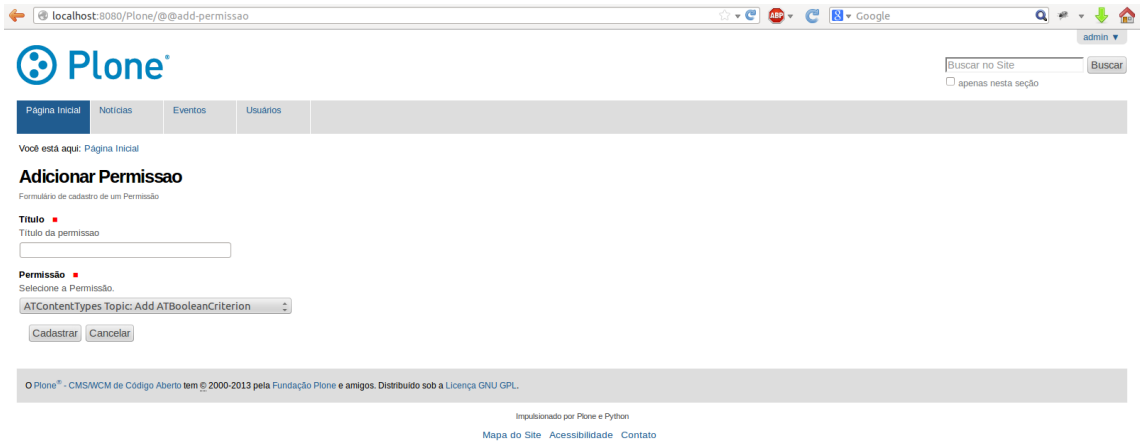

Figura 4.5: Tela de cadastro de permissões

4.1.5 Efetuar ligação

A Tabela 4.5 demonstra como um usuário efetua a ligação entre "Permissão—Papel" para o respectivo estado. A Figura 4.6 apresenta a tela de ligação.

Tabela 4.5: Efetuar ligação

Caso de Uso:	Efetuar ligação
Objetivo:	Efetuar a ligação entre papel e permissão
Descrição:	O usuário, depois de logar-se no portal, acessa a área de busca de estados, então acessa o *link* "Ligação" do respectivo estado. É apresentado uma tabela onde as linhas são as permissões e as colunas os papéis, nessa tabela o usuário pode marcar as opções "Permissão—Papel". Após as marcações o sistema efetua a persistência dos dados.
Atores:	Analista
Pré-condição:	Usuário válido autenticado ao Portal com permissão de *manager*
Pós-condição:	Ligação criada
Fluxo Principal:	1. Usuário efetua as marcações 2. Sistema confirma o cadastramento
Fluxos Alternativos:	1. Usuário consulta as ligações (a) Sistema apresenta os dados 2. Usuário altera os dados da ligação (a) Sistema confirma as alterações

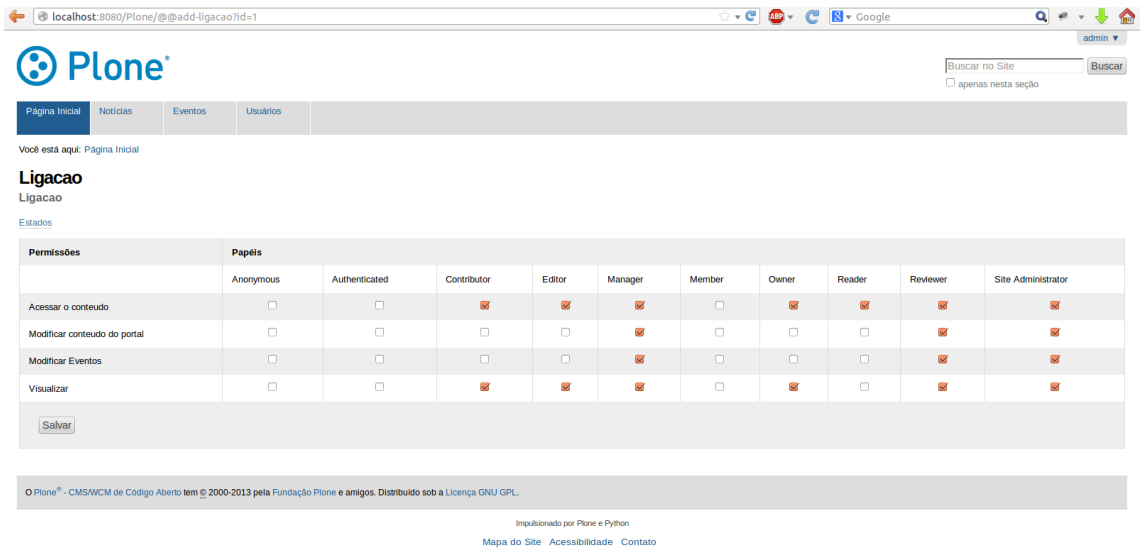

Figura 4.6: Tela de ligação

4.1.6 Exportação do arquivo ZIP com as regras e *scripts* de testes de *workflow*

A Tabela 4.6 apresenta como um usuário efetua a exportação de um arquivo no formato ZIP com as definições das regras de *workflow* e os *scripts* responsáveis pela validação. A Figura 4.7 demonstra essa ação.

Tabela 4.6: Exportação do arquivo ZIP com as regras e *scripts* de testes de *workflow*

Caso de Uso:	Exportação do arquivo ZIP com as regras e *scripts* de testes de *workflow*
Objetivo:	Efetuar exportação do arquivo ZIP
Descrição:	O usuário depois de logar-se no portal acessa a área de busca de *workflows*, então acessa o link "Exportar *Workflow*" do respectivo *workflow*. O sistema efetua a geração de um arquivo XML, titulado *"data.xml"*, esse arquivo contém as regras iniciais para esse *workflow*, ou seja, os requisitos iniciais. É gerado também dois *scripts* Python: o primeiro, *"read.py"*, que contém as funções que efetuam a leitura e conversão do XML para uma estrutura de dados; o segundo, *script "test.py"*, contendo as funções que validam as regras de *workflow*.
Atores:	Analista e Desenvolvedor
Pré-condição:	Usuário válido autenticado ao Portal com permissão de *manager* ou editor
Pós-condição:	Regras de *workflow* e *scripts* de testes exportados em um arquivo ZIP
Fluxo Principal:	1. Usuário acessa a tela de busca de *workflow* 2. Sistema gera os arquivos 3. Sistema devolve um arquivo no formato ZIP para o usuário
Fluxos Alternativos:	

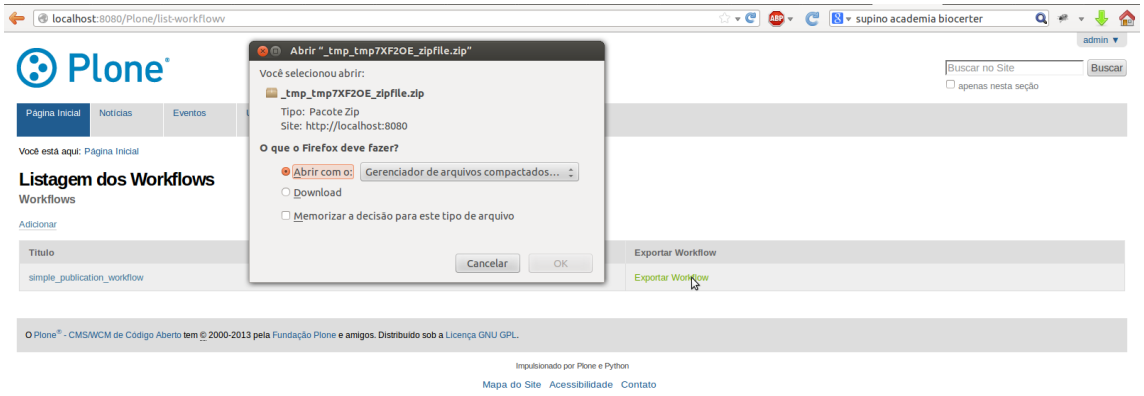

Figura 4.7: Tela de exportação os arquivo .ZIP

4.1.7 Executar teste

A Tabela 4.7 demonstra como um usuário efetua a execução dos testes. Já a Figura 4.8 apresenta um exemplo de execução do *script* de validação.

Tabela 4.7: Executar teste

Caso de Uso:	Executar teste
Objetivo:	Executar os *scripts* de testes
Descrição:	O usuário após a exportação do arquivo ZIP irá fazer a extração do mesmo. Após isso irá executar o comando "python test.py", informando o arquivo XML com as definições do *workflow* Plone a ser validado, com isso o *script* irá validar se esse arquivo atende aos requisitos iniciais, apresentando as respectivas mensagens ao usuário.
Atores:	Analista e Desenvolvedor
Pré-condição:	Ter a linguagem de programação Python instalada na máquina
Pós-condição:	Validar se o *workflow* desenvolvido atende aos requisitos iniciais
Fluxo Principal:	1. Usuário roda o *script* de teste 2. *Script* valida as regras 3. *Script* apresenta uma resposta para o usuário
Fluxos Alternativos:	

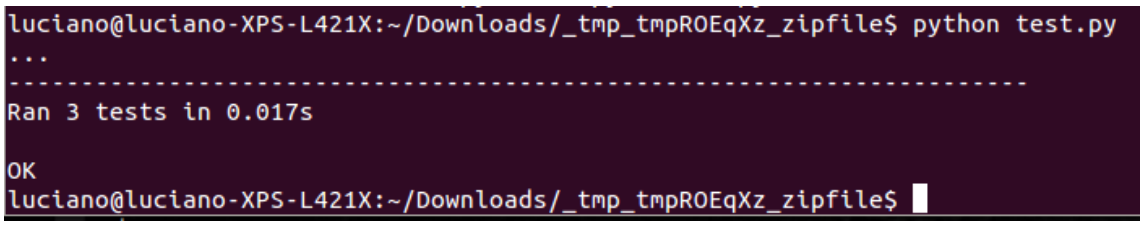

Figura 4.8: Tela de execução dos testes de validação

4.2 Arquitetura do *framework workflow.validation*

A Figura 4.9 ilustra a arquitetura do *framework workflow.validation*, sendo que essa arquitetura é dividida em 5 camadas. Na camada 1 encontra-se o "*framework workflow.validation*" que é executado como um produto Plone. Na camada 2 encontra-se o "Plone", atuando como gerenciador de conteúdo. Na camada 3 encontra-se o "Zope" operando como o servidor de aplicações. Na camada 4 tem-se a linguagem de programação "Python", sendo que juntamente com o Python encontra-se o *SQLAlchemy* que efetua a conexão com o banco de dados e possuí as ferramentas de manipulação dos dados. Na camada 5 o banco de dados responsável pela persistência dos dados, sendo que neste caso foi utilizado o banco de dados relacional *PostgreSQL*[1].

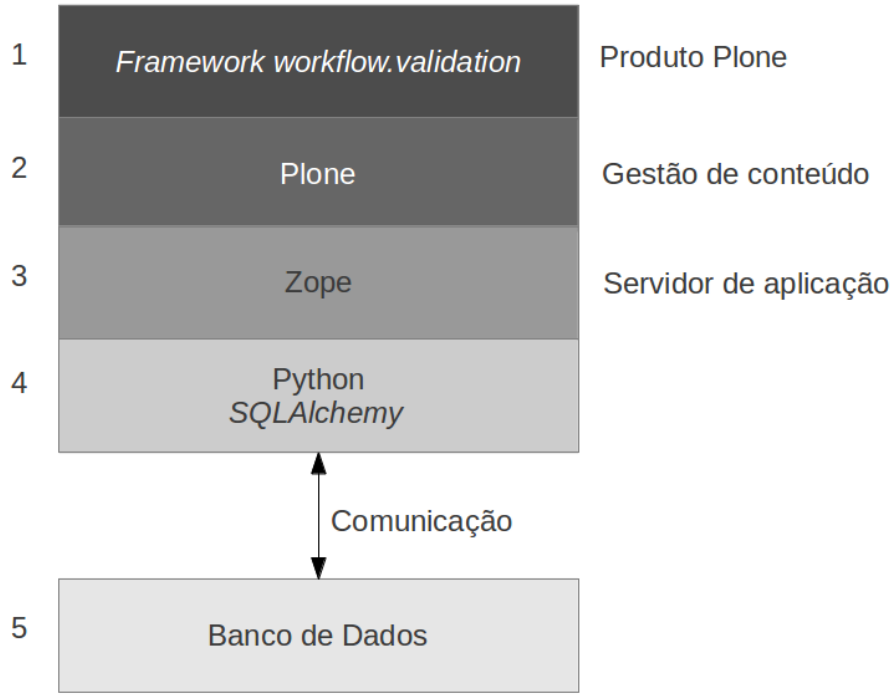

Figura 4.9: Diagrama da arquitetura do *framework workflow.validation*

[1]http://www.postgresql.org/

4.3 Diagrama do Banco de Dados

A Figura 4.10 apresenta o diagrama do banco de dados. Esse diagrama é composto por cinco tabelas, que são a tabela "papel", "permissao", "estado", "ligacao" e "*workflow*".

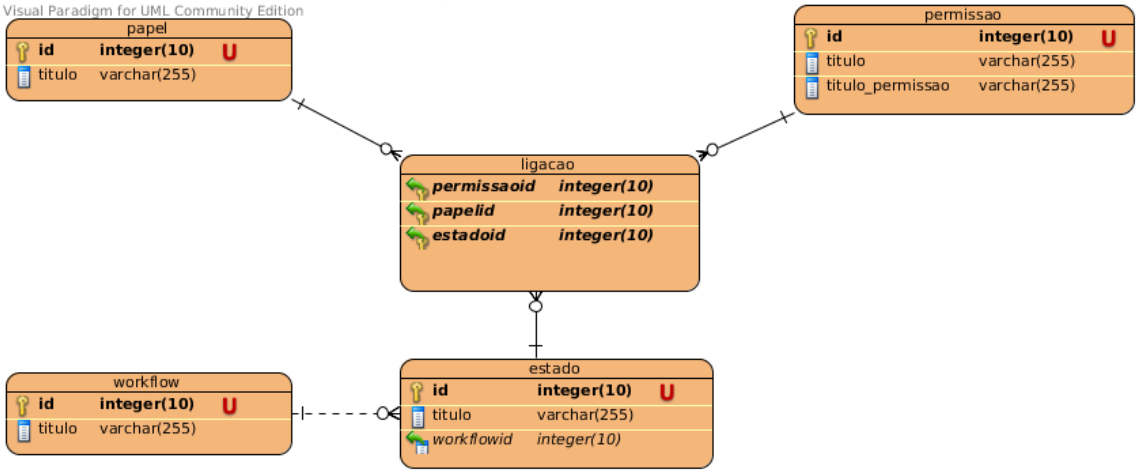

Figura 4.10: Diagrama do Banco de Dados

A tabela "*workflow*" tem como atributos o identificador e título. A tabela "estado" possui uma ligação com a tabela "*workflow*", ou seja, quando for criado um estado é obrigatório informar qual *workflow* ele pertencerá. Essa tabela é composta pelos atributos identificador, título e identificador do *workflow*. A tabela "papel" tem como atributos o identificador e título. A tabela "permissao" tem como atributos o identificador, título e título permissão. O atributo título é utilizado, por exemplo, como um nome fantasia para a permissão. O atributo título permissão é utilizado para informar o real título da permissão, sendo que deve ser informado em Inglês. Por fim, a tabela "ligacao" que é constituída pela relação entre uma permissão, um estado e um papel.

4.4 Diagrama de Classe

A Figura 4.11 apresenta o diagrama de classe da implementação desenvolvida. Cada uma das classes desse diagrama é associada a uma respectiva tabela da Figura 4.10. A implementação segue este formato pois optou-se pela utilização do *SQLAlchemy*.

O *SQLAlchemy* é um conjunto de ferramentas para o Mapeamento Objeto-Relacional *ORM* (*Object Relational Mapper*) desenvolvida em Python para efetuar o mapeamento das tabelas de um banco de dados relacional. Através do desenvolvimento de classes Python, o *SQLAlchemy* faz a associação das classes com as tabelas do banco de dados, assim como todas as funcionalidades de inserção, edição, exclusão e consulta dos dados (SQLALCHEMY, 2013).

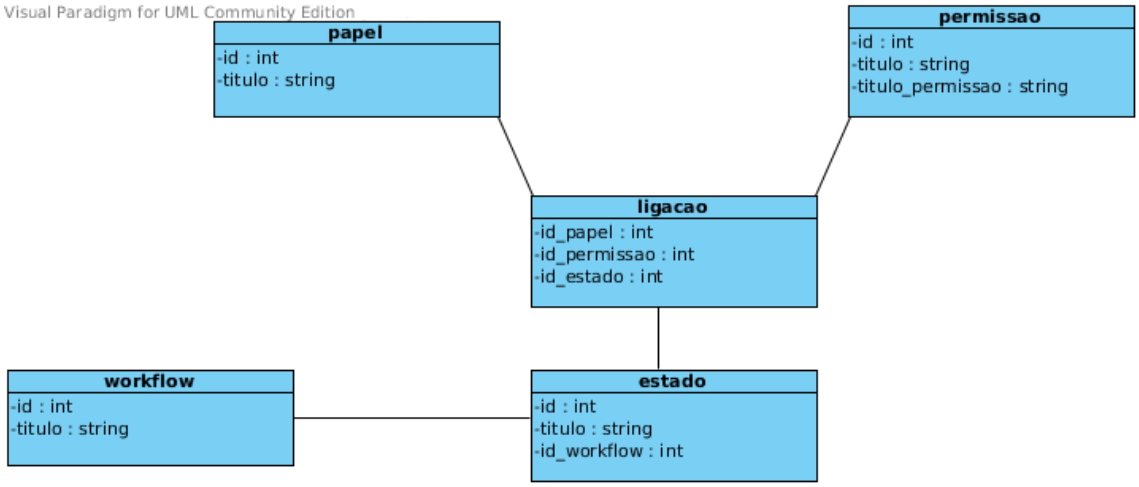

Figura 4.11: Diagrama de Classe

4.5 Desenvolvimento da Solução

O produto desenvolvido foi chamado de *workflow.validation*[2]. Ele é disponibilizado como software livre sob licença GPLv2[3]. O sistema é composto por inúmeros *scripts* que serão apresentados apenas a partir de suas principais funcionalidades, por aspectos didáticos.

No Trecho de Código 4.2, tem-se a classe *Workflowv* desenvolvida no padrão do *SQLAlchemy*. A classe possui dois atributos: o *id* que é chave primária da tabela *workflowv* e o atributo *titulo* que contém o título do *workflow*. Para usar o *SQLAlchemy* no Plone, é necessário a criação de um conjunto de regras de configuração, que são exemplificadas no Trecho de Código 4.1. Destaca-se que o Trecho de Código 4.1 é uma dependência do Trecho de Código 4.2.

Trecho de Código 4.1: Arquivo de configuração do produto.

```
1 #-*- coding: utf-8 -*-
2
3 from sqlalchemy.ext.declarative import declarative_base
4 from z3c.saconfig import named_scoped_session
5
6 Base = declarative_base()
7 SCOPED_SESSION_NAME = "session.workflow.validation.db"
8 session = named_scoped_session(SCOPED_SESSION_NAME)
```

Trecho de Código 4.2: Classe Workflow no padrão do *SQLAlchemy*.

```
1 #-*- coding: utf-8 -*-
2
3 from sqlalchemy import Column, Integer, String, ForeignKey
4 from sqlalchemy.orm import relationship, backref
5 from workflow.validation.config import Base
6 from workflow.validation import interfaces
7
8 from zope.interface import implements
9
10 class Workflowv(Base):
11     implements(interfaces.IWorkflowv)
12     __tablename__ = 'workflowv'
13
14     id = Column(Integer, primary_key = True)
15     titulo = Column(String)
16
17     def __init__(self, titulo):
18         self.titulo = titulo
```

No Trecho de Código 4.3, tem-se as classes responsáveis pelo funcionamento de inserção, edição e visualização de um *workflow*. A funcionalidade de exclusão está contida na classe base, sendo utilizada nas demais classes do sistema.

[2]https://bitbucket.org/lccruz/workflow.validation
[3]http://www.gnu.org/licenses/gpl-2.0.html

Trecho de Código 4.3: Funcionalidades de manipulação dos dados de *workflow*.

```
#-*- coding: utf-8 -*-

from five import grok
from plone.app.layout.navigation.interfaces import INavigationRoot

from workflow.validation.browser.forms import base
from workflow.validation.config import MessageFactory as _
from workflow.validation.config import session
from workflow.validation.interfaces import IWorkflowv
from workflow.validation.db import Workflowv

class WorkflowvAddForm(base.AddFormWv):
    """
        Formulario de cadastro de um Workflow.
    """

    grok.context(INavigationRoot)
    grok.name('add-workflowv')
    grok.require('cmf.ManagePortal')

    schema = IWorkflowv
    klass = Workflowv
    label = _(u'Adicionar Workflow')
    description = _(u'Formulario de cadastro de um Workflow')

    def createAndAdd(self, data):
        w = Workflowv(data['titulo'])
        session.add(w)
        session.flush()


class WorkflowvEditForm(base.EditFormWv):
    """
        Formulario de edicao de um Workflow.
    """

    grok.context(INavigationRoot)
    grok.name('edit-workflowv')
    grok.require('cmf.ManagePortal')

    schema = IWorkflowv
    klass = Workflowv
    label = _(u'Editar Workflow')
    descrition = _(u'Formulario de edicao de um Workflow.')


class WorkflowvShowForm(base.ShowFormWv):
    """
        Formulario de visualizacao de um Workflow.
    """

    grok.context(INavigationRoot)
    grok.name('show-workflowv')
    grok.require('cmf.ManagePortal')

    schema = IWorkflowv
    klass = Workflowv
    label = _(u'Detalhes do Workflow')
```

```
59      description = _(u'Formulario de visualizacao de uma Workflow.')
```

No Trecho de Código 4.4, tem-se as funções que são empacotadas no arquivo ZIP a ser exportado. A primeira função *get_arquivo_read()* retorna uma variável, cujo conteúdo é um código, cuja funcionalidade é efetuar a leitura do arquivo XML e transformar seu conteúdo em um dicionário Python.

Trecho de Código 4.4: Funções que são empacotadas no arquivo ZIP a ser exportado.

```
1  #-*- coding: utf-8 -*-
2
3  def get_arquivo_read():
4      """ Gera arquivo com DEF que le o XML
5      """
6      arquivo_read ="""# -*- coding:utf-8 -*-
7
8  import os
9  from xml.dom import minidom
10
11 def get_workflow_xml(workflow_file, padrao=True):
12
13     if padrao:
14         tags = dict(workflow = {'tag':'dc-workflow', 'attr':'name'},
15                     state = {'tag':'state', 'attr':'state_id'},
16                     permission = {'tag':'permission-map', 'attr':'name'},
17                     roles = {'tag':'permission-role', 'attr':''})
18     else:
19         tags = dict(workflow = {'tag':'workflow', 'attr':'name'},
20                     state = {'tag':'state', 'attr':'name'},
21                     permission = {'tag':'permission', 'attr':'name'},
22                     roles = {'tag':'role', 'attr':'name'})
23
24     PROJECT_PATH = os.path.abspath(os.path.dirname(__file__))
25     try:
26         doc = minidom.parse(("%s/%s") % (PROJECT_PATH, workflow_file))
27     except:
28         return False
29     workflow_name = doc.getElementsByTagName(tags['workflow']['tag'])[0].
           getAttribute(tags['workflow']['attr'])
30     permission_roles = {}
31     states_permission_roles = {}
32     role_list = []
33
34     for state in doc.getElementsByTagName(tags['state']['tag']):
35         for permission in state.getElementsByTagName(tags['permission']['tag']):
36             for role in permission.getElementsByTagName(tags['roles']['tag']):
37                 role_list.append(role.childNodes[0].data)
38             permission_roles[permission.getAttribute(tags['permission']['attr'])
                   ] = tuple(role_list)
39             role_list = []
40         states_permission_roles[state.getAttribute(tags['state']['attr'])] =
               permission_roles
41         permission_roles = {}
42     return dict(workflow_name=workflow_name, states_permission_roles=
           states_permission_roles)"""
43     return arquivo_read
```

O Trecho de Código 4.5 apresenta o empacotamento da segunda função, *get_arquivo_test()*, que possui a classe e os métodos principais, responsáveis pelos testes automatizados. A partir dos dados extraídos do arquivo XML, o arquivo de teste é automaticamente gerado contendo as regras que serão utilizadas para uma comparação entre os estados e permissões especificados com estados e permissões implementados. Esse define também a regra que compara os papéis relacionados às permissões especificadas.

Trecho de Código 4.5: Classe que é empacotada no arquivo ZIP a ser exportado responsável pelos testes.

```
#-*- coding: utf-8 -*-

def get_arquivo_test():
    """ Gera arquivo de teste
    """

    arquivo_test = """#-*- coding: utf-8 -*-
import unittest
from read import *

class TestWorkflow(unittest.TestCase):

    def setUp(self):
        #definicoes iniciais
        self.requisites = get_workflow_xml('data.xml', False)
        self.requisites_name = self.requisites.get('workflow_name')
        self.requisites_content = self.requisites.get('states_permission_roles')
        self.requisites_content_states = self.requisites_content.keys()
        self.requisites_content_permissions = self.requisites_content.values()
            [0].keys()
        #workflow Plone
        self.definition = get_workflow_xml('definition.xml', True)
        self.definition_name = self.definition.get('workflow_name')
        self.definition_content = self.definition.get('states_permission_roles')
        self.definition_content_states = self.definition_content.keys()
        self.definition_content_permissions = self.definition_content.values()
            [0].keys()

    def test_states(self):
        self.assertItemsEqual(self.requisites_content_states, self.
            definition_content_states)

    def test_permissions(self):
        self.assertItemsEqual(self.requisites_content_permissions, self.
            definition_content_permissions)

    def test_permissions_roles(self):
        for state in self.requisites_content_states:
            permissions = self.requisites_content.get(state)
            for permission,roles in permissions.items():
                self.assertItemsEqual(roles,self.definition_content[state][
                    permission])

if __name__ == '__main__':
    unittest.main()"""
```

```
41     return arquivo_test
```

No Trecho de Código 4.6, tem-se a classe *ExportWorkflow* que apresenta dois métodos. O método *_gera_xml()* tem a função de gerar um arquivo XML a partir de consultas no banco de dados, contendo os dados do *Workflow* cadastrados pelo Analista. O método *render* é responsável por responder a requisição do usuário com o arquivo ZIP já construído. Esse método efetua a criação de um diretório temporário no servidor, para efetuar a criação do arquivo *"data.xml"*, que é gerado pelo método *_gera_xml()*. Este método cria o *script "read.py"* com o conteúdo retornado através da chamada de função *get_arquivo_read()* e o arquivo *"test.py"* através da função *get_arquivo_test()*, essas duas funções são importadas do Trecho de Código 4.4 e do Trecho de Código 4.5. Após a geração do arquivo ZIP, o cabeçalho de retorno da requisição é modificado de forma a retornar um arquivo ao invés de conteúdo HTML. Após o ZIP ser adicionado ao cabeçalho de retorno ele é fechado e excluído do servidor, ficando somente na memória do servidor. Por fim, é feito o retorno da requisição, retornando o arquivo ZIP para o usuário. Esse arquivo ZIP contém todos os *scripts* necessários para que o usuário possa validar as regras de *workflow* implementadas com as regras contidas no requisito inicial. Com o *download* do arquivo ZIP concluído, o usuário irá fazer a descompactação do mesmo em qualquer diretório e executará o *script* Python chamado de *"test.py"*. Para a execução desse *script* basta executar o comando "python test.py" no terminal e informar o caminho das regras de *workflow* padrão do Plone para iniciar a validação. O *script* retornará sucesso se as regras de workflow respeitarem aos requisitos iniciais, caso contrário retornará falha e informará ao usuário qual foi o erro.

Trecho de Código 4.6: Classe que gera e exporta o arquivo ZIP.

```
 1 #-*- coding: utf-8 -*-
 2
 3 # -*- coding: utf-8 -*-
 4
 5 import os
 6 import shutil
 7 import zipfile
 8 import tempfile
 9
10 from xml.dom.minidom import Document
11 from five import grok
12 from plone.app.layout.navigation.interfaces import INavigationRoot
13
14 from workflow.validation.config import session
15 from workflow.validation import db
16 from workflow.validation.config import MessageFactory as _
17 from workflow.validation.nav import url
18 from workflow.validation.browser.content_export import get_arquivo_read
19 from workflow.validation.browser.content_export import get_arquivo_test
20
21 class ExportWorkflowv(grok.View):
```

```
22
23      grok.name('export-workflowv')
24      grok.context(INavigationRoot)
25      grok.require('cmf.ManagePortal')
26
27      def _gera_xml(self, workflow_id):
28          doc = Document()
29          root = doc.createElement('workflow')
30          workflow = session.query(db.Workflowv).get(workflow_id)
31          root.setAttribute( "name", workflow.titulo)
32          doc.appendChild(root)
33
34          for estado in workflow.estado:
35              estadoxml = doc.createElement('state')
36              estadoxml.setAttribute("name", estado.titulo)
37              root.appendChild(estadoxml)
38              permissoes = session.query(db.Ligacao.permissao_id).filter_by(
                    estado_id=estado.id).group_by(db.Ligacao.permissao_id).order_by(
                    'permissao_id').all()
39              for permissao in permissoes:
40                  papeis = session.query(db.Ligacao).filter(db.Ligacao.estado_id==
                        estado.id,db.Ligacao.permissao_id==permissao[0]).all()
41                  permissaoxml = doc.createElement('permission')
42                  permissaoxml.setAttribute("name", papeis[0].permissao.
                        titulo_permissao)
43                  estadoxml.appendChild(permissaoxml)
44                  for papel in papeis:
45                      text = doc.createTextNode(papel.papel.titulo)
46                      rolexml = doc.createElement('role')
47                      rolexml.appendChild(text)
48                      permissaoxml.appendChild(rolexml)
49          return doc.toprettyxml()
50
51      def render(self):
52          workflow = session.query(db.Workflowv).get(self.request.get('id'))
53          if not workflow:
54              return
55          tempdir = tempfile.mkdtemp(dir='/tmp/')
56
57          zipname = "%s/%s.zip" % (tempdir,'zipfile')
58          zf = zipfile.ZipFile(zipname, mode='w')
59
60          self.context.REQUEST.RESPONSE.setHeader('Content-Type','application/zip'
                )
61          self.context.REQUEST.RESPONSE.setHeader('Content-Disposition','
                attachment; filename=%s'%(zipname))
62
63          try:
64              #zf.write(outfile.name)
65              zf.writestr("data.xml", self._gera_xml(self.request.get('id')))
66              zf.writestr("read.py", get_arquivo_read())
67              zf.writestr("test.py", get_arquivo_test())
68          finally:
69              zf.close()
70
71          fp = open(zipname, 'rb')
72          self.context.REQUEST.RESPONSE.write(fp.read())
73          fp.close()
74
```

```
75          shutil.rmtree(tempdir)
76          return
```

4.6 Cenário de Uso

O produto *workflow.validation* foi utilizado em um problema real para validar uma demanda de um cliente da empresa HaDi.Com - Soluções para Educação Ltda. O principal requisito é o desenvolvimento de um produto Plone para a inscrição de estudantes no Centro de Educação Tecnológica e de Pesquisa em Saúde. O produto a ser desenvolvido possuía um requisito funcional, cujo objetivo era impossibilitar ao estudante a edição do seu cadastro após a submissão do mesmo. Para atender esse requisito foi necessário o desenvolvimento de um *workflow*.

A etapa de criação do *workflow* iniciou-se com o cadastramento dos requisitos no produto *workflow.validation* pelo analista (4.12). Com os requisitos devidamente cadastrados, o desenvolvedor inicia a etapa de codificação do novo *workflow*.

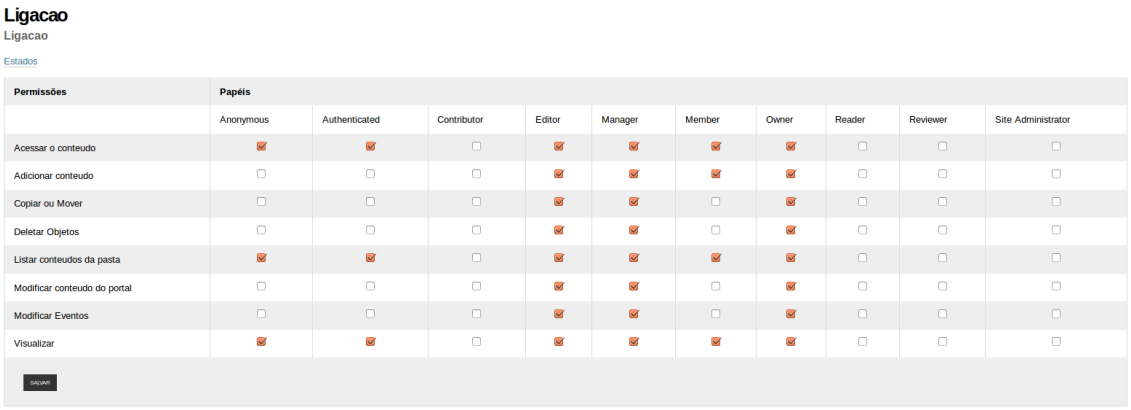

Figura 4.12: Requisito cadastrado no produto "workflow.validation"

Após a conclusão da etapa de codificação, o analista pode validar a codificação do *workflow*. Para isso o analista efetua os procedimentos descritos no caso de uso *Exportação do arquivo ZIP* (Tabela 4.6) e *Executar Teste* (Tabela 4.7).

O Trecho de Código 4.7 apresenta o código gerado automaticamente a partir da execução do procedimento (Tabela 4.6). As linhas 10 à 14 definem o *workflow* especificado. As linhas 15 à 20 representam o *workflow* desenvolvido na etapa de implementação. O teste de validação dos estados dos dois *workflows* é executado na linha 23. Igualmente, o teste de validação das permissões é feito na linha 26. As linhas 28 à 32 validam se os papéis definidos para as permissões estão consistentes entre o *workflow* especificado e o *workflow* implementado.

Trecho de Código 4.7: *Script* Python que realiza dos testes.

```
1 #-*- coding: utf-8 -*-
```

```
 2
 3 import unittest
 4 from read import *
 5
 6 class TestWorkflow(unittest.TestCase):
 7
 8     def setUp(self):
 9         #definicoes iniciais
10         self.requisites = get_workflow_xml('data.xml', False)
11         self.requisites_name = self.requisites.get('workflow_name')
12         self.requisites_content = self.requisites.get('states_permission_roles')
13         self.requisites_content_states = self.requisites_content.keys()
14         self.requisites_content_permissions = self.requisites_content.values()
               [0].keys()
15         #workflow Plone
16         self.definition = get_workflow_xml('definition.xml', True)
17         self.definition_name = self.definition.get('workflow_name')
18         self.definition_content = self.definition.get('states_permission_roles')
19         self.definition_content_states = self.definition_content.keys()
20         self.definition_content_permissions = self.definition_content.values()
               [0].keys()
21
22     def test_states(self):
23         self.assertItemsEqual(self.requisites_content_states, self.
               definition_content_states)
24
25     def test_permissions(self):
26         self.assertItemsEqual(self.requisites_content_permissions, self.
               definition_content_permissions)
27
28     def test_permissions_roles(self):
29         for state in self.requisites_content_states:
30             permissions = self.requisites_content.get(state)
31             for permission,roles in permissions.items():
32                 self.assertItemsEqual(roles,self.definition_content[state][
                       permission])
33
34 if __name__ == '__main__':
35     unittest.main()
```

Ao executar o procedimento *Executar Teste* (Tabela 4.7) um erro é localizado. Esse corresponde a um erro de implementação, onde o desenvolvedor atribuiu incorretamente papéis a uma respectiva permissão. Este erro é mostrado na Figura 4.13.

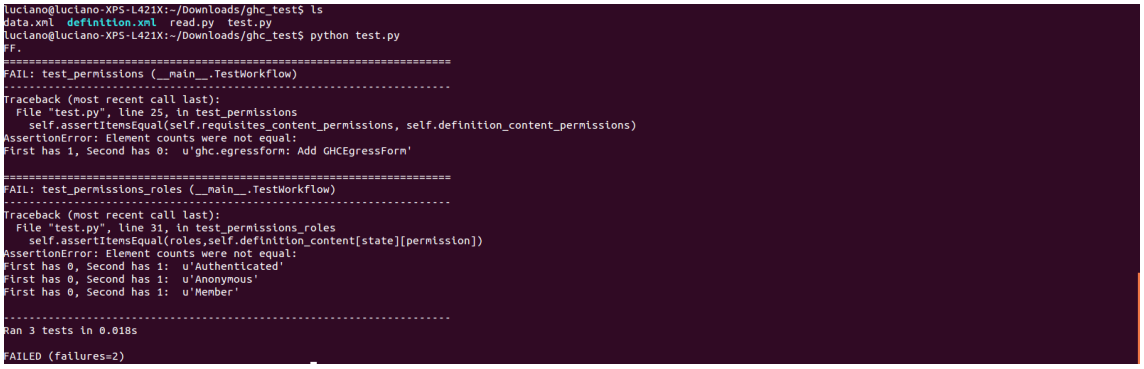

Figura 4.13: Execução dos testes de validação de *workflow* identificando um erro

Após a identificação do erro, é aberto um *ticket* para o desenvolvedor efetuar a correção. Concluída a correção, o analista efetua um novo teste que não sinaliza diferenças entre os requisitos iniciais e o *workflow* desenvolvido. A Figura 4.14 mostra que a execução dos testes foi efetuado com sucesso.

Figura 4.14: Execução dos testes de validação de *workflow* sem identificar erros

4.7 Considerações Finais

Neste capítulo, foi apresentada uma proposta de teste funcional de um *workflow* Plone, procurando confrontar a especificação de um *workflow* por um analista e a correta implementação. Normalmente não existem testes automatizados para esta etapa da construção em um portal Plone.

Essa proposta partiu de um conjunto de casos de uso que oferecem possibilidades de edição da especificação do *workflow* pelo analista. A seguir, o analista ou o desenvolvedor podem utilizar o produto implementado, *workflow.validation*, para mapear a especificação em arquivos XML do próprio *workflow* Plone. A partir deste mapeamento, essa proposta de solução de teste foi por uma abordagem funcional que, a partir do próprio mapeamento, gera automaticamente um conjunto de regras *Unittest* que valida o *workflow* Plone especificado contra o *workflow* Plone desenvolvido.

As etapas para o desenvolvimento do produto *workflow.validation* abrangeram a descrição dos casos de uso, a definição do diagrama do banco de dados, do diagrama de classe, da arquitetura do *framework workflow.validation* e explicações dos principais trechos de códigos.

Para validar e experimentar a abordagem proposta, foi utilizado um cenário real de uso em que existem duas pessoas que ocupam os papéis de analista e

desenvolvedor. O fluxo de operação para um cenário sem o produto de teste consistia simplesmente na definição manual do *workflow* Plone, na etapa posterior de implementação e no teste manual efetuado pelo analista, onde podiam ser constatado os erros. No entanto, o analista, juntamente com o desenvolvedor deveria testar todas as combinações possíveis (caminhamentos) no grafo de estados e permissões para "cada um dos papéis definidos", o que exigia um tempo considerável de teste, além de estar sujeito a erros humanos durante o processo.

O cenário de teste com o produto *workflow.validation*, exigiu apenas a especificação do *workflow* Plone pelo analista, com uma etapa de mapeamento, geração do código de teste e execução do teste, mesmo antes da implementação do *workflow* Plone. A partir do *workflow.validation* o analista pode detectar os erros de implementação sem precisar definir exaustivamente e manualmente as permissões para todos os papéis definidos. Uma vez aprovado o *workflow* Plone, o desenvolvedor terá apenas a tarefa de adaptar o arquivo XML gerado à aplicação Plone específica.

5 CONCLUSÃO

O presente trabalho abordou questões relativas à testes de software, a linguagem de programação Python, ao servidor de aplicação Zope, ao gerenciador de conteúdo Plone e ao funcionamento de *workflows* Plone. A partir desses estudos foi encontrada uma limitação na abordagem de testes automatizados usados no Plone, essa limitação consiste na inviabilidade de realização de testes funcionais nas regras de *workflow*, ou seja, a realização de testes que permitam verificar se o desenvolvimento de um *workflow* Plone atende aos requisitos iniciais. Como solução desse problema foi desenvolvido um produto (*workflow.validation*) Plone. O *workflow.validation* tem como principal objetivo validar se o *workflow* desenvolvido está coerente aos requisitos iniciais, sendo que essa validação é efetuada através da automatização de testes funcionais. Destaca-se que o produto desenvolvido pode ser utilizado para validar qualquer *workflow* Plone.

5.1 Contribuições

A conclusão desse trabalho resultou em um produto Plone que facilita o processo de validação para o desenvolvimento de *workflows* Plone. Esse produto foi validado em um caso de uso real apresentado no Capítulo 4 e que contribuiu para a diminuição do tempo no processo de validação da etapa de testes de um *workflow*.

Como trabalhos futuros, é possível adicionar funcionalidades de visualização das regras iniciais de um *workflow* de forma gráfica e descritiva, com a opção de exportação ou impressão. Também é possível adicionar funcionalidades para a seleção da troca de transições entre os estados de um *workflow*. O produto desenvolvido nesse trabalho, como já referido no texto, é software livre disponibilizado no repositório *https://bitbucket.org/lccruz/workflow.validation*. Qualquer pessoa poderá utilizá-lo livremente e, se preferir, contribuir no desenvolvimento de novas funcionalidades.

REFERÊNCIAS

ABNT. **ABNT NBR ISO 9000 - Sistema de gestão da qualidade - Fundamentos e vocabulário**. Brasil: ABNT - Associação Brasileira de Normas Técnicas, 2005.

ALBA, V. F. de. **Plone 3 Intranets**. 32 Lincoln Road Olton Birmingham - B27 6PA - UK: Packt Publishing Ltd, 2010. ISBN 978-1-847199-08-9.

ASPELI, M. **Professional Plone Development**. 1.ed. Birmingham, B27 6PA, UK: Packt Publishing Ltd, 2007. ISBN 978-1-847191-98-4.

BERNARDO, P. C.; KON, F. **A Importância dos testes automatizados**. Rio de Janeiro: Engenharia de Software Magazine, 2008.

BORGES, L. E. **Python para Desenvolvedores**. 2.ed. Rio de Janeiro: Edição do Autor, 2010. ISBN 978-85-909451-1-6.

COOPER, C. **Building Websites with Plone**. 32 Lincoln Road Olton Birmingham - B27 6PA - UK: Packt Publishing Ltd, 2004. ISBN 1-904811-02-7.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. Rio de Janeiro: Elsevier, 2007. ISBN 978-85-352-2634-8.

FEWSTER, M.; GRAHAM, D. **Software Test Automation – Effective use of test execution tools**. New York: Pearson Education, 1999. ISBN 0-201-33140-3.

GITPLONE. **GitPlone**. [S.l.: s.n.], 2013. <Disponível em: `https://github.com/plone`>. Acesso em: 8 de Abril de 2013.

IEEE, S. C. C. of the Computer Society of the. **IEEE Standard Glossary of Software Engineering Terminology**. New York, USA: The Institute of Electrical and Electronics Engineers, 1990. ISBN 1-55937467.

LARMAN, C. **Applying UML and Patterns**: an introduction to object-oriented analysis and design and the unified process. 2.ed. USA: Prentice Hall PTR, 2002. ISBN 0-13-092569-1.

LUTZ, M.; ASCHER, D. **Aprendendo Python**. 2.ed. Porto Alegre: Bookman, 2007. ISBN 978-85-7780-013-1.

MYERS, G. J. **The Art of Software Testing**. 2.ed. New Jersey: Word Association, 2004. ISBN 0-471-46912-2.

NETO, A. C. D. **Planejamento de Testes a partir de Casos de Uso**. 6.ed. Rio de Janeiro: Engenharia de Software Magazine, 2008.

PLONE. **Plone.org**. [S.l.: s.n.], 2013. <Disponível em: `http://www.plone.org`>. Acesso em: 8 de Abril de 2013.

PRESSMAN, R. S. **Engenharia de software**. 6.ed. São Paulo: McGraw-Hill, 2006. ISBN 85-86804-57-6.

PRODUTOS, P. **Add-on Product Releases**. [S.l.: s.n.], 2013. <Disponível em: `http://plone.org/products`>. Acesso em: 22 de Maio de 2013.

PYTHON. **Python.org**. [S.l.: s.n.], 2013. <Disponível em: `http://www.python.org`>. Acesso em: 8 de Abril de 2013.

PYTHONBRASIL. **Documentação Python**. [S.l.: s.n.], 2013. <Disponível em: `http://www.python.org.br/wiki/DocumentacaoPython`>. Acesso em: 17 de Maio de 2013.

ROCHA, A. R. C. da; MALDONADO, J. C.; WEBER, K. C. **Qualidade de software**. 1.ed. São Paulo: Prentice Hall, 2001. ISBN 85-87918-54-0.

ROSSUM, G. van. **Tutorial Python**. Release 2.4.2.ed. [S.l.]: Python Software Foundation, 2005.

SOMMERVILLE, I. **Engenharia de software**. 6.ed. São Paulo: Addison Wesley, 2003. ISBN 85-88639-07-6.

SQLALCHEMY. **Documentação SQLAlchemy**. [S.l.: s.n.], 2013. <Disponível em: `http://www.sqlalchemy.org/`>. Acesso em: 3 de Novenbro de 2013.

STAHL, J. **Creating Workflows in Plone**. [S.l.]: plone.org, 2013. <Disponível em: `http://plone.org/documentation/kb/creating-workflows-in-plone/tutorial-all-pages`>. Acesso em: 8 de Abril de 2013.

TALES, V. M. **Extreme programming**: aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade. 3.ed. São Paulo: Novatec Editora, 2009. ISBN 85-7522-047-0.

UNITTEST, P. S. F. **Unit testing framework**. [S.l.]: python.org, 2013. <Disponível em: `http://docs.python.org/2/library/unittest.html`>. Acesso em: 17 de Junho de 2013.

VENNERS, B. **The Making of Python**. [S.l.]: artima.com, 2013. <Disponível em: `http://www.artima.com/intv/python.html`>. Acesso em: 17 de Maio de 2013.

ZOPE. **Zope.org**. [S.l.: s.n.], 2013. <Disponível em: `http://www.zope.org`>. Acesso em: 8 de Abril de 2013.