**UNIVERSIDADE FEDERAL DE SANTA CATARINA**

**DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA**

**CURSO DE SISTEMAS DE INFORMAÇÃO**

# FERRAMENTA PARA TESTE DE PÁGINAS JSF GERADAS A PARTIR DE UIDS

**ERIC FELIPE BARBOZA**

**Florianópolis - SC**

**2013 / 1**

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

# FERRAMENTA PARA TESTE DE PÁGINAS JSF GERADAS A PARTIR DE UIDS

Eric Felipe Barboza

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de informação.

**Orientadora**
Profa. Dra. Patrícia Vilain

**Membros da Banca:**
Prof. Dr. José Eduardo De Lucca
Prof. Dr. Leandro José Komosinski

Florianópolis – SC
2013 / 1

Eric Felipe Barboza

FERRAMENTA PARA TESTE DE PÁGINAS JSF GERADAS A PARTIR DE UIDS

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de informação.

Orientadora: ________________________________________

Profa. Dra. Patrícia Vilain

Banca Examinadora

____________________________________________

Prof. Dr. José Eduardo De Lucca

____________________________________________

Prof. Dr. Leandro José Komosinski

# Agradecimentos

Agradeço a todos que contribuíram de alguma forma para a realização deste trabalho.

À Professora Patrícia, de maneira especial, pela dedicação e competência durante o processo de orientação. Sempre se mostrando extremamente solícita e comprometida com o projeto.

Aos meus pais, primeiro por serem espelhos e exemplos para minha trajetória, ensinando-me logo cedo os princípios morais de que preciso para tornar-me uma boa pessoa. Além de proporcionarem uma base sólida me deram a oportunidade de concluir este passo importante da minha formação. Obrigado por confiarem em mim e por toda torcida pelo meu sucesso.

À minha linda namorada que me apoiou durante todo o desenvolvimento deste trabalho e soube ser compreensiva nos momentos mais difíceis.

Aos meus amigos, por me apoiarem, torcerem por mim e até mesmo por me ajudarem a descansar quando o projeto não estava fluindo. Vocês me ajudaram a vencer, seja de forma direta ou indireta, obrigado.

# Sumário

1. Introdução ..... 1
1.1. Motivação ..... 2
1.2. Justificativa ..... 4
1.3. Objetivo Geral e Objetivos Específicos ..... 4
2. Fundamentação Teórica ..... 6
2.1. Testes de Software ..... 6
2.1.1. Teste de Desempenho ..... 7
2.1.2. Teste de Estresse ..... 7
2.1.3. Teste de Unidade ..... 7
2.1.4. Teste de Integração ..... 8
2.1.5. Teste automatizado de Interface de Usuário ..... 8
2.1.6. Teste de aceitação ..... 9
2.2. UIDs ..... 9
2.2.1. Características ..... 10
2.2.2. Notação ..... 10
2.2.3. Exemplo de UID. ..... 17
2.3. Ferramenta de Mapeamento UID2JSF ..... 18
2.3.1. Exemplo de mapeamento UID2JSF ..... 20
2.4. Selenium ..... 22
2.4.1. Selenium RC ..... 22
2.4.2. Selenium WebDriver ..... 24
2.5. Padrão de Design Page Object (objeto de página) ..... 25
2.6. JavaServer Faces ..... 26
2.6.1. JavaBeans ..... 27
2.6.2. Naming Container ..... 28
3. Trabalhos Relacionados ..... 30

3.1. Reverse Engineering of GUI Models for Testing ........................................................ 30

3.2. User Interface Test Automation and its Challenges in an Industrial Scenario............... 31

3.3. Model Based Testing Of Web Applications ................................................................ 32

3.3.1. TestOptimal™........................................................................................................ 33

3.3.2. Utilizando a ferramenta de modelagem ................................................................ 33

3.3.3. Execução do modelo ............................................................................................. 35

4. Decisões de Design ........................................................................................................ 36

4.1. Definição de identificadores para elementos JSF ........................................................ 36

4.2. Definição de identificadores de localização de elementos JSF em páginas JSF............ 38

4.3. Definição de testes para páginas JSF ........................................................................... 39

4.3.1. Processo de criação de objetos de páginas............................................................ 40

4.3.2. Métodos de teste para objetos de página............................................................... 41

4.4. Processo de criação de casos de teste........................................................................... 43

4.5. Geração de Managed Beans com dados mockados...................................................... 44

5. Desenvolvimento da ferramenta ..................................................................................... 48

5.1. Definição do processo de desenvolvimento.................................................................. 48

5.2. Desenvolvimento........................................................................................................... 49

5.2.1. Lista de Requisitos.................................................................................................. 49

5.2.2. Arquitetura do projeto............................................................................................. 51

5.3. Iterações ........................................................................................................................ 53

5.3.1. Primeira iteração ..................................................................................................... 54

5.3.2. Segunda iteração ..................................................................................................... 57

5.3.3. Terceira iteração...................................................................................................... 61

5.3.4. Quarta iteração ........................................................................................................ 64

5.4. Exemplos de testes gerados a partir do mapeamento de UIDs para JSF........................ 66

6. Considerações Finais ....................................................................................................... 67

Referências Bibliográficas ................................................................................................... 70

# Lista De Figuras

Figura 2.1 - Exemplo de UID........................................................................................................ 18
Figura 2.2 - Mostrar gravações de um CD. .................................................................................. 20
Figura 2.3 - Página referente ao estado inicial do UID Mostrar gravações de um CD .............................. 21
Figura 2.4 - Página gerada para o segundo estado de iteração do UID Mostrar gravações de um CD. ...... 22
Figura 2.5 - Modelo de arquitetura Selenium RC ......................................................................... 24
Figura 2.6 - Exemplo de objeto de página ................................................................................... 26
Figura 2.7 - Trecho de página XHTML ........................................................................................ 27
Figura 2.8 - Exemplo de classe Managed Bean ............................................................................ 28
Figura 2.9 – Exemplo Naming Container. .................................................................................... 29
Figura 3.1 – Modelo de transformação de diagramas de sequencia em casos de teste. ............................ 32
Figura 3.2 – mScript ...................................................................................................................... 34
Figura 4.1 - Exemplo de geração de IDs com prefixo. ................................................................... 38
Figura 4.2 - Exemplo de tela mapeada a partir da ferramenta UID2JSF .................................................. 41
Figura 4.3 - Exemplo de objeto de página .................................................................................... 41
Figura 4.4 - Exemplo de métodos de teste em objeto de página ........................................................... 42
Figura 4.5 - Exemplo de classe de caso de teste ......................................................................... 44
Figura 4.6 - Exemplo de classe Managed Bean gerada. ................................................................. 46
Figura 4.7 - Exemplo de classe Managed Bean Helper gerada. ........................................................ 47
Figura 5.1 - Diagrama de pacotes para a nova arquitetura do sistema. ............................................... 51
Figura 5.2 - Antigo diagrama de atividades a aplicação UID2JSF ........................................................ 52
Figura 5.3 - atual diagrama de atividades para a aplicação UID2JSF. .................................................. 53
Figura 5.4 - Diagrama de sequência para criação dos arquivos das páginas ......................................... 55
Figura 5.5 - Diagrama de sequência para representar de uma forma macro o processo de geração das classes do projeto UID2JSF 2.0. ......................................................................................... 56
Figura 5.6 - Exemplo de modelo utilizado pelo Velocity. ............................................................... 59
Figura 5.7 - Exemplo de objeto de página. ................................................................................... 61
Figura 5.8 - Exemplo de classe de teste. ...................................................................................... 63
Figura 5.9 - Exemplo de execução de classe de teste. ................................................................... 64
Figura 5.10 - Exemplo de classe Managed Bean gerada. ................................................................ 65
Figura 5.11 - Exemplo de classe auxiliar para classe Managed Bean. ................................................ 66

# Lista De Tabelas

Tabela 2.1 - Tabela de mapeamento dos elementos do UID para componentes JSF.................................. 19

Tabela 4.1 - Prefixo para definição de IDs para elementos JSF ................................................................. 37

# Resumo

Este trabalho apresenta o desenvolvimento e validação de uma ferramenta para geração de testes de interface automatizados baseados no processo de mapeamento de UIDs (Diagrama de Interação do Usuário) para JSF (Java Server Faces), realizado pela ferramenta UID2JSF, a qual foi desenvolvida por DAMIANI (2011). A ferramenta UID2JSF possui diversas regras de mapeamento que definem se, para um dado UID, uma ou mais páginas JSF serão criadas e quais componentes farão parte dessas páginas. O UID representa toda troca de informação entre o usuário e a aplicação, e seus elementos apresentam características semelhantes as dos componentes utilizados nas interfaces das aplicações. O JSF é um framework que dá suporte ao desenvolvimento de aplicações WEB desenvolvidas utilizando a linguagem Java e possui uma biblioteca padrão de componentes visuais para elaboração das páginas.

Trabalhando com o código fonte da ferramenta UID2JSF, foi possível refatorar esse código e introduzir lógicas responsáveis por capturar informações de componentes JSF mapeados a partir de UIDs. Utilizando as informações capturadas, foram geradas lógicas para a criação de classes de teste junto com o processo de mapeamento de UIDs para JSF. Essas classes são responsáveis por validar as páginas web geradas pela ferramenta UID2JSF. Os frameworks Selenium e JUnit foram utilizados na estruturação das classes de teste. O Selenium ficou encarregado de fazer a comunicação com a aplicação web e trocar informações com componentes presentes nas páginas JSF. Já o framework JUnit foi utilizado para rodar os casos de teste gerados e , quando necessário, fazer asserções de valores de componentes capturados pelo Selenium.

A validação da ferramenta desenvolvida foi realizada com a geração de páginas e classes de teste para um conjunto de UIDs elaborados para o projeto de uma aplicação para venda de CDs musicais através de uma loja virtual.

Palavras-Chave: UID, JSF, TESTES DE INTERFACE, SELENIUM, JUNIT,APLICAÇÃO WEB

# 1. Introdução

Processos de testes são de extrema importância no ciclo de desenvolvimento de um projeto, seja este na área de TI ou em qualquer outra área de desenvolvimento de produtos [CHIANG, 1994]. Nenhuma organização tem o interesse de entregar seu produto ao cliente final sem que este esteja funcionando adequadamente. Porém, devido à falta de tempo e à intensa cobrança de seus clientes, empresas acabam entregando seus produtos sem que estes tenham passado pelos testes necessários.

A confiabilidade de um software é algo bastante complexo de se discutir, tanto por não ser algo mensurável como por não ser algo possível de garantir. Por isso, para aumentar confiabilidade de um software, a confiabilidade deve ser tratada em nível de sistema e não somente em nível de componente. Isso porque em um sistema os componentes são interdependentes, e uma falha em um componente por mais simples que este seja pode se propagar pelo sistema e afetar a funcionalidade de outros componentes [AGARWAL, TAYAL e GUPTA, 2010].

Agarwal, Tayal e Gupta (2010) apresentaram três definições para confiabilidade:

*Confiabilidade de Software é definida como a probabilidade que um programa de computador tem de operar livre de falhas em um determinado ambiente durante um determinado tempo.*

*OU*

*Confiabilidade de um produto de software denota essencialmente sua confiabilidade ou segurança.*

*OU*

*Confiabilidade de um produto de software também pode ser definida como a probabilidade do produto funcionar corretamente durante um dado período de tempo.*

Hoje em dia ainda é possível ver inúmeras empresas que ao final do desenvolvimento de um módulo específico ou de um sistema inteiro, contam apenas com testes manuais para garantir a integridade e eficiência de seu produto [BERNARDO, 2011].

Deve-se saber que o processo de testes, independente da ferramenta ou produto que venha a ser testado, pode ser iniciado logo que os requisitos sejam estabelecidos pelo cliente. Uma área até então não muito explorada é a da geração de interfaces a partir de modelagens diagramáticas, tendo a ferramenta de mapeamento de UIDs para JSF, chamada UID2JSF, como um dos recentes projetos nessa área e que servirá de base para este trabalho [DAMIANI, 2011]. O UID (Diagrama de Interação do Usuário) é uma ferramenta diagramática que representa a interação do usuário com o sistema. A elaboração dos UIDs é feita pelo analista durante a tarefa de levantamento de requisitos de uma aplicação, juntamente com os casos de uso. Os UIDs contribuem com a tarefa de levantamento de requisitos, e ainda podem ser utilizados nas etapas de projeto conceitual, projeto navegacional e projeto da interface com o usuário [VILAIN, 2002].

Já o Java Server Faces (JSF) é um framework para desenvolvimento de aplicações Web que utilizam tecnologia Java. Este trabalha principalmente nas camadas de Visão e Controle do modelo MVC [BUSCHMANN, MEUNIER, ROHNERT, SOMMERLAD e STAL 1996]. Seu principal objetivo é facilitar o trabalho dos desenvolvedores na tarefa de construção das páginas das aplicações Web [MARAFON 2006].

Visto que a ferramenta de mapeamento UID2JSF irá gerar páginas JSF a partir de regras e fluxos estabelecidos nos UIDs, se torna de grande valia a aplicação de testes de interface baseados em UIDs. Os testes gerados a partir dos UIDs ajudarão a economizar parte do tempo gasto com desenvolvimento de testes e permitirá que os testadores da aplicação possam gastar esse tempo elaborando testes mais complexos ou executando outras tarefas que julguem mais importantes.

## 1.1. Motivação

Qualidade de software é um campo amplo e muito importante da engenharia de software. Este é abordado por diversos órgãos de normalização, como a ISO (International Organization for Standardization), IEEE (Institute of Electrical and Electronics Engineers), ANSI (American

National Standards Institute), etc [AGARWAL, TAYAL e GUPTA, 2010]. Porém, devido à falta de tempo e ao alto custo em manter equipes de testes, o processo de teste acaba sendo deixado de lado, o que afeta diretamente a qualidade final de um projeto.

O mais comum em empresas de desenvolvimento de software é que o processo de teste de software seja iniciado ao final do ciclo de desenvolvimento de um produto. Porém, é visto na literatura que o custo para efetuar mudanças no código fonte ou nos requisitos, é muito maior nas fases finais do que no início do projeto [PATTON, 2001].

PATTON (2001 p.17), diz o seguinte:

"Inúmeros estudos foram feitos de pequenos até extremamente grandes projetos e os resultados foram os mesmos. A causa número um de problemas nos softwares é a especificação."

Existem muitas razões para as especificações serem um grande causador de defeitos. Em muitos casos os projetos nem mesmo possuem uma especificação definida. Em outros casos pode haver especificação não detalhada o suficiente ou as especificações podem estar mudando constantemente [PATTON, 2001]. Por esses motivos, o início dos testes junto com a geração das especificações de um sistema é muito importante, pois assim, é possível validar junto ao cliente se o que ele está solicitando está de acordo com as funcionalidades do sistema sendo verificadas pelos testes.

O mercado possui, atualmente, uma grande quantidade de ferramentas que dão suporte a teste de aplicações web. Com essas ferramentas é possível abordar diversos métodos de teste, como Testes de Captura / Reprodução[1], Testes de Unidade[2].

A automação de teste de interface já vem sendo realizado por inúmeras ferramentas de teste, por exemplo, HttpUnit[3] e Selenium[4]. No entanto, nenhuma dessas ferramentas utiliza técnica de geração automatizada de testes de interface a partir de UIDs.

---

[1] **BRADLEY SOFTWARE. BADBWEB - WEB BASED APPLICATIONS TESTING TOOL.** http://www.badboy.com.au.

[2] **IEUNIT UNIT TEST FRAMEWORK FOR WEB PAGES.** http://www.ieunit.googlecode.com

[3] **HTTPUNIT AUTOMATED TESTING SOFTWARE** http://httpunit.sourceforge.net

[4] **OPEN QA. SELENIUM TEST TOOL**. http://selenium.openqa.org

## 1.2. Justificativa

Hoje em dia, com um mercado tão global como é o mercado de soluções de software, é de extrema importância que essas soluções apresentem interfaces de qualidade, a fim de manter a confiabilidade do usuário final.

Porém, teste de interface não é uma tarefa simples nem rápida. Ela consome muito tempo dos testadores e pode ser uma tarefa cara, em especial, para pequenas empresas de desenvolvimento de software ou desenvolvedores autônomos.

O número de pessoas empenhadas em melhorar o processo de automação de casos de teste e as técnicas empregadas neles vem crescendo com o passar dos anos. A área de Automação da Geração de Casos de Teste de Interface é uma das áreas que tem sido bastante estudada. Isso porque a geração manual desse tipo de testes pode consumir muito tempo e produzir erros que venham afetar a qualidade e confiabilidade dos testes [SELENIUM, 2013].

A ferramenta UID2JSF [SEKE, 2010] possui como característica predominante a geração automática de páginas JSF a partir de Diagramas de Interação com Usuário (UID), onde páginas web podem ser geradas logo na fase de especificação de um sistema.

Este projeto visa, com o apoio de boas práticas de programação e padrões de projeto, apresentar uma ferramenta que inicie o processo de teste de interfaces de um projeto a partir dos UIDs criados, iniciando o processo de automatização dos testes já na fase de especificação de um sistema. A geração dos testes de interface deve ocorrer junto com a geração das páginas JSF feita pela ferramenta UID2JSF.

## 1.3. Objetivo Geral e Objetivos Específicos

## Objetivo Geral

O objetivo geral deste trabalho consiste na definição de estratégias para criação de testes de interface automatizados baseados nas definições dos mapeamentos dos UIDs (Diagramas de Interação do Usuário) para os componentes JSF (Java Server Faces) e no desenvolvimento de um protótipo de uma ferramenta que venha a executar testes de interface a partir de casos de testes pré-determinados.

## Objetivos Específicos

Abaixo estão os objetivos específicos deste trabalho, os quais, em conjunto, irão auxiliar a alcançar o objetivo geral.

- Estudo das técnicas relacionadas com o trabalho: UIDs, Mapeamento dos UIDs para JSF, técnicas de testes de interface, ferramentas para testes de interface.
- Utilizando a linguagem Java, implementar classes responsáveis por gerar testes automatizados de interface baseadas em componentes JSF mapeados a partir de UIDs.
- Definição de casos de testes que validem a presença na tela de componentes JSF que serão criados pela ferramenta UID2JSF ao mapear um UID.
- Desenvolvimento e validação da ferramenta, utilizando a linguagem Java.

## Estrutura do Trabalho

Este trabalho está organizado da seguinte forma. No capítulo 2 são apresentadas as técnicas de teste de software e ferramentas relacionadas ao trabalho: teste de software, UID, ferramenta de mapeamento UID2JSF, Selenium, Padrão de Design Page Object e JSF.

No capítulo 3 é apresentado um estudo sobre algumas ferramentas para geração automatizada de testes de interface.

No capítulo 4 é apresentado o design para a geração de testes baseado em UID. O capítulo mostra a estrutura básica que as classes geradas devem ter e também apresenta exemplos dessas classes.

No capítulo 5 é descrito o desenvolvimento da ferramenta. O capítulo descreve como foi feita a implementação do projeto para que este pudesse gerar classes de teste para a aplicação web gerada pela ferramenta UID2JSF

No capítulo 6 é apresentado um exemplo de aplicação da ferramenta desenvolvida para geração de testes de interface JSF geradas a partir de UIDs.

No capítulo 7 são apresentadas as conclusões obtidas com este trabalho.

# 2. Fundamentação Teórica

## 2.1. Testes de Software

Testar uma aplicação é uma tarefa técnica que também envolve considerações importantes de economia e psicologia [MYERS, 2004]. Quando MYERS (2004) fala em seu livro sobre a psicologia de testes, este diz que uma das causas primárias de baixa qualidade nos testes de programas é o fato de os testadores começarem os testes com a falsa definição do termo "Teste de Software". Na sequência deste parágrafo estão algumas definições do conceito de teste de software comumente empregadas por testadores:

- "Teste é o processo de mostrar que erros não estão presentes."

  ou

- "O propósito de testar é mostrar que um programa executa corretamente o que foi proposto."

Porém essas definições não são as mais apropriadas.

MYERS (2004) diz que, quando se testa um programa, deseja-se agregar um valor a ele. Agregar valor através dos testes significa aumentar a confiabilidade e qualidade do programa. Aumentar a confiabilidade do programa significa encontrar e solucionar erros. Para Myers, a definição apropriada é: "Testar é o processo de executar o programa com a intenção de encontrar erros".

Em um mundo ideal, todos os possíveis casos de testes poderiam ser executados e todos os possíveis erros de um programa seriam encontrados. MYERS (2004) comenta em seu livro "The Art of Software Testing", que criar casos de teste para todas as possíveis entradas e saídas de um programa é algo impraticável. Isso porque até mesmo um simples programa pode conter centenas de possíveis combinações de entradas e saídas. Com isso, criar casos de teste para uma aplicação complexa, a fim de testá-la por completo, seria algo muito demorado e exigiria muito recurso humano para que fosse economicamente viável.

O fato de ser impraticável a execução de todos os casos de testes possíveis acaba também afetando a forma de planejar os casos de teste, a parte econômica dos testes e as suposições que os testadores terão de fazer sobre o funcionamento do programa [MYERS, 2004].

Para amenizar as consequências geradas pela não cobertura completa dos casos de erro presentes em um programa, existem diversas abordagens de testes que auxiliam na cobertura das partes que integram o programa. Algumas dessas abordagens de testes serão explicadas a seguir.

#### 2.1.1. Teste de Desempenho

O teste de desempenho tem por objetivo verificar como componentes de um sistema estão se comportando em uma situação específica de execução. Escalabilidade, uso de recursos e confiabilidade, são outros fatores analisados pelos testes [SOFTWARE TESTING HELP, 2013].

Testes de desempenho executam trechos do sistema em teste e armazenam os tempos de duração obtidos, como um cronômetro. Os resultados desses testes ajudam a identificar os gargalos do sistema que precisam ser otimizados, visando diminuir o tempo de resposta à requisições feitas por um usuário final do sistema [BERNARDO, 2011].

#### 2.1.2. Teste de Estresse

O teste de estresse, também conhecido por teste de carga máxima, tem por objetivo descobrir os limites de uso da infraestrutura de um sistema; encontrar sob quais circunstâncias o sistema deixa de funcionar. Esta estratégia de teste visa verificar qual a quantidade máxima de usuários e requisições que a infraestrutura de um sistema consegue atender em um dado tempo. Por exemplo, se o sistema foi requisitado para atender 10 requisições por segundo e uma carga de dados faz com que ele tenha que atender 20 requisições por segundo, o sistema estará sendo estressado [BURNSTEIN, 2003].

#### 2.1.3. Teste de Unidade

Os testes de unidade visam testar de forma individual os subprogramas, sub-rotinas ou procedimentos de um sistema [MYERS, 2004]. Ao invés de testar o sistema como um todo, os testes de unidade são direcionados a testarem os menores trechos de código que possuam comportamentos definidos.

Segundo AGARWAL (2010), et al., existem inúmeras razões para testar aplicações usando teste de unidades ao invés de testar a aplicação como um todo:

- O tamanho do trecho de código testado é pequeno o suficiente para que seja fácil a localização de um erro.
- O tamanho do trecho de código testado é pequeno o suficiente para que sejam executados testes de forma exaustiva.
- A confusão entre erros gerados em diferentes partes do software é eliminada.

### 2.1.4. Teste de Integração

Segundo BERNARDO (2011), teste de integração é uma denominação ampla que representa a busca de erros de relacionamento entre quaisquer módulos de um software, incluindo desde a integração de pequenas unidades até a integração de bibliotecas das quais um sistema dependa, como servidores e gerenciadores de banco de dados.

O tamanho e a complexidade da arquitetura de um sistema aumenta a chance de existirem erros de integração, visto que aumenta a troca de mensagens entre módulos. Outro fator que pode gerar erros de integração é a utilização de bibliotecas de terceiros sem um conhecimento profundo de como estas funcionam [BERNARDO, 2011].

### 2.1.5. Teste automatizado de Interface de Usuário

É através das interfaces dos sistemas que normalmente as empresas apresentam seus produtos. É também a partir dessa camada que os usuários finais dos sistemas mantêm contato direto e constante com o produto. Portanto, a avaliação sobre a qualidade de um produto pelo usuário final, será, em boa parte, a partir do que a ele for apresentado.

Os testes automatizados de interfaces verificam o correto funcionamento de interfaces através da simulação de eventos de usuários reais, como se uma pessoa física estivesse controlando dispositivos como *mouse* e teclado. A partir dos efeitos colaterais gerados pelos eventos, são feitas verificações na interface e em outras partes do sistema para garantir que o mesmo está funcionando como o esperado [BERNARDO, 2011].

Neste trabalho será apresentada uma proposta de geração de teste de interface web de forma automática. Os testes, que serão gerados junto com o processo de mapeamento de UID para JSF, feito pela ferramenta UID2JSF, irão testar algumas funcionalidades das páginas JSF mapeadas, como por exemplo, as possíveis navegações que cada página possua.

### 2.1.6. Teste de aceitação

Testes de aceitação são testes formais conduzidos para determinar se um sistema satisfaz critérios de aceitação especificados e permitir que o cliente do sistema possa determinar se o sistema desenvolvido foi aceito [IEEE Std 1012-1986].

Os testes de aceitação devem utilizar uma linguagem próxima da natural para evitar problemas de interpretação e de ambiguidades, também devem ser facilmente conectados ao código do sistema para que os comandos de verificações sejam executados no sistema em teste [MUGRIDGE, et al., 2005].

## 2.2. UIDs

> "Um Diagrama de Interação do Usuário ou UID (User Interaction Diagram) representa a interação entre o usuário e uma aplicação que apresenta intensa troca de informações e suporte à navegação. Este diagrama descreve somente a troca de informações entre o usuário e a aplicação, sem considerar aspectos específicos da interface com o usuário nem da navegação." [VILAIN 2003].

> "Um UID é composto por um conjunto de estados conectados através de transições. Os estados representam as informações que são trocadas entre o usuário e a aplicação, enquanto as transições são responsáveis pela troca do foco da interação de um estado para outro. Diz-se que um estado da interação é o foco da interação quando as informações contidas nesse estado representam as informações que estão sendo trocadas entre o usuário e a aplicação em um dado momento. As informações participantes da interação entre o usuário e a aplicação são apresentadas dentro dos estados de interação, mas algumas seleções e opções são associadas às transições. As transições são disparadas, geralmente, pela entrada ou seleção de informações pelo usuário." [VILAIN 2003].

### 2.2.1. Características

Uma das importantes características dos UIDs é o fato de que eles podem ser compreensíveis tanto para os projetistas quanto para os usuários [VILAIN, 2002]. Sendo que a definição de um diagrama compreensível pelos usuários foi o requisito mais importante no momento da especificação da notação dos UIDs. Os UIDs não apresentam, diagramaticamente, requisitos não funcionais descritos nos casos de uso. Nesses casos a representação deve ser feita através de notas textuais.

De acordo com Newman (2000, apud Vilain 2002), a representação usada não deve apresentar detalhes irrelevantes, tais como fontes, cores e imagens, pois o importante durante a fase inicial do projeto é obter o *feedback* necessário com os usuários em relação à estrutura e organização das informações. Por outro lado, a representação também não pode ser grosseira, pois neste caso dará a impressão de que o projeto não é profissional. Segundo VILAIN (2002), acredita-se que os UIDs contribuam para dar uma impressão profissional, junto aos usuários, na fase inicial do projeto, ao mesmo tempo em que focam na estrutura e organização das informações, sem abordar detalhes irrelevantes.

### 2.2.2. Notação

O UID apresenta uma notação simples, de fácil compreensão tanto para o projetista quanto para o usuário, e independente da interface com usuário que será projetada.

#### 2.2.2.1. Item de Dado

Um item de dado representa uma informação única (simples) que aparece durante a interação. O item de dado pode estar acompanhado do seu domínio, onde, neste caso, deve ser seguido por dois pontos e o nome do domínio. O projetista fica responsável por definir os domínios usados na especificação da interação entre o usuário e o sistema através dos UIDs. É importante que o projetista defina domínios que possam ser facilmente compreendidos pelos usuários (e.g., Número, Texto, Som, Imagem, Vídeo). Geralmente o nome do item de dado é escrito em letras minúsculas. Se seu domínio não for especificado, é assumido domínio Texto.

<item de dado>:<domínio>

<item de dado>

: Som

: Imagem

: Vídeo

#### 2.2.2.2. Estrutura

Uma estrutura representa uma coleção de informações (itens de dados, estruturas, conjunto de itens de dados ou conjunto de estruturas) que se relacionam de alguma maneira. O nome da estrutura deve sempre ser especificado e por padrão o nome é escrito com a primeira letra em maiúscula.

<Estrutura> (<item de dado 1>, <item de dado 2>...<item de dado n>)

<Estrutura> ()

#### 2.2.2.3. Conjunto

Representa um conjunto de itens de dados ou estruturas. Tem a sua multiplicidade representada pelos símbolos min..max em frente ao item de dado ou estrutura. A multiplicidade padrão é 1..N e é representada somente por reticências (...). Para facilitar o entendimento do diagrama pelo usuário, o rótulo Conjunto pode ser incluído após as reticências.

…<item de dado >

… <Estrutura> (<item de dado 1>, <item de dado 2> .. <item de dado n>)

… <Estrutura> ()

1..5 <item de dado>

0..5 <item de dado>

#### 2.2.2.4. Dado Opcional

Um dado opcional representa um item de dado, estrutura ou texto opcional. Para representar graficamente os dados opcionais utiliza-se o símbolo ”?”, um conjunto com multiplicidade

“0..1”, ou então, no caso de uma entrada do usuário, pode ainda ser representado por um retângulo com linhas pontilhadas.

<item de dado>?

…<item de dado>?

<Estrutura> (<item de dado 1>, <item de dado 2> .. <item de dado n>)?

“ texto “?

0..1 <item de dado>

<item de dado>

#### 2.2.2.5. Entrada do usuário

Uma entrada de usuário é um item de dado ou estrutura fornecido pelo usuário ao sistema. Para representar graficamente uma entrada de usuário, deve-se colocar a informação dentro de um retângulo.

<item de dado>

Caso haja dependência entre duas entradas de itens de dados e ao menos um precisa ser fornecido pelo usuário, utiliza-se o símbolo “OR”. Caso de somente um dos itens de dados poder ser fornecido, então é usado o símbolo “XOR” entre eles.

<item de dado1> OR <item de dado2>

<item de dado1> XOR <item de dado2>

#### 2.2.2.6. Entrada do Usuário Enumerada

É uma entrada do usuário que deve ser selecionada a partir de uma lista de opções. Estas opções aparecem entre colchetes e separadas por vírgula. A multiplicidade é representada em frente ao nome do item de dado.

<item de dado> [<opção 1>, <opção 2>...<opção n>]

<mim>..<max> <item de dado> [<opção 1>, <opção 2>...<opção n>]

#### 2.2.2.7. Saída do Sistema

É toda informação retornada pelo sistema, item de dado ou estrutura, e deve ser colocada diretamente no estado de interação.

<item de dado> : <domínio>

<item de dado>

<item de dado >?

…<item de dado>

<Estrutura> (<item de dado>)

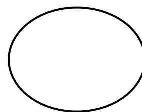
#### 2.2.2.8. Texto

Representa um texto pré-definido, de caráter explicativo, apresentado pelo sistema durante a interação. Deve estar entre aspas duplas.

“<texto>”

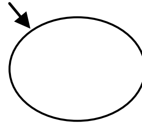
#### 2.2.2.9. Estado da Interação

Um estado de interação é representado graficamente por uma elipse. As informações trocadas entre o usuário e o sistema estão inseridas nesta elipse.



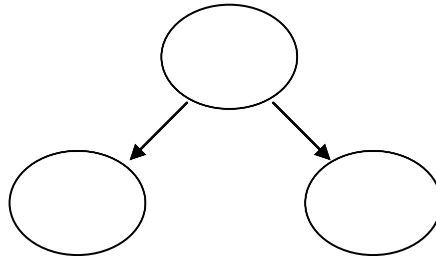
#### 2.2.2.10. Estado Inicial da Interação

É representado por uma transição sem origem. É o estado no qual começa o fluxo das trocas de informações entre o usuário e o sistema.



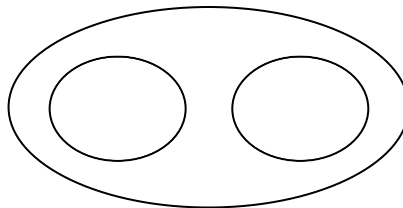
#### 2.2.2.11. Estados Alternativos da Interação

É utilizado quando existem duas ou mais saídas alternativas a partir de um estado de interação. O estado da interação que se tornará foco da interação depende da informação fornecida pelo usuário ou da opção selecionada por ele.



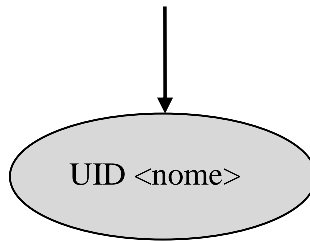
#### 2.2.2.12. Sub-Estados de um Estado da Interação

São utilizados para representar partes excludentes de um estado de interação. O usuário deve optar por qual sub-estado vai seguir durante a interação.



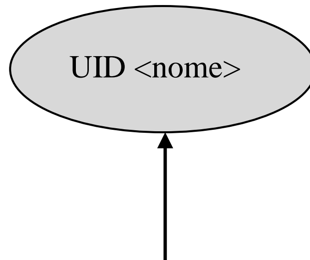
#### 2.2.2.13. Chamada de outro UID

É utilizado para representar que o foco da interação foi transferido para outro UID.



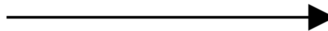

### 2.2.2.14. Chamada a partir de outro UID

Representa que o foco da interação foi transferido a partir de outro UID.




### 2.2.2.15. Transição do Estado da Interação

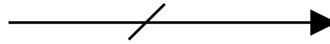
São utilizadas para representar a troca do foco da interação. O estado destino torna-se o novo foco da interação após o sistema retornar as informações necessárias e o usuário fornecer os dados requisitados no estado origem.



Uma transição deve ter como origem um estado da interação, um sub-estado de um estado da interação, ou ainda, um item de dado ou uma estrutura retornados pelo sistema.

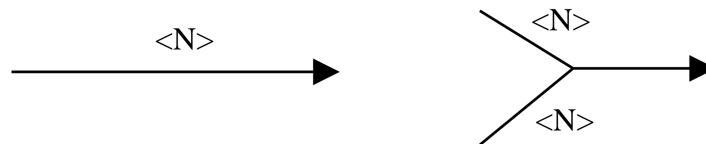
Para representar, explicitamente, que o usuário pode retornar ao estado de interação origem da transição, deve ser utilizada a seguinte representação:

Porém, quando for necessário representar que o usuário não pode retornar ao estado origem da transição, então se usa a seguinte representação:

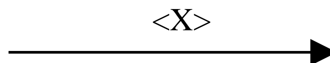


Uma transição pode apresentar três tipos de rótulos:

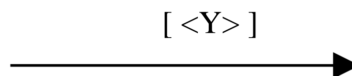
- *Transição com Seleção de N Elementos*: para uma transição receber o rótulo de quantos elementos devem ser selecionados para que o estado destino torne-se foco da interação, a origem dela deve ser um conjunto de itens de dados ou estruturas retornados pelo sistema. A seleção pode ser de elementos de conjuntos diferentes.




- *Transição com Seleção da opção X*: para que o estado destino torne-se foco da interação é necessário que a opção X seja selecionada pelo usuário.




- *Transição com Condição Y*: para que a mudança do foco da interação ocorra, a condição Y deve ser satisfeita.



#### 2.2.2.16. Pré-Condições e Pós-Condições

Quando a execução da interação representada no UID requer que algumas condições sejam satisfeitas previamente, estas condições são representadas pelas pré-condições associadas ao UID.

Pré-condições: <condições>

Para representar as condições que precisam ser satisfeitas ao término da execução da interação representada no UID, usa-se as pós-condições.

Pós-condições: <condições>

#### 2.2.2.17. Notas Textuais Anexadas ao UID

As notas textuais servem para especificar alguma informação importante que não pode ser representada graficamente no UID. Também são utilizadas para descrever os requisitos não-funcionais.

Nota: <nota textual>

### 2.2.3. Exemplo de UID.

A Figura 2.1 mostra o exemplo de um UID com os principais elementos apresentados na seção 2.2.2. Este exemplo representa o caso de uso Seleção de um CD a partir de um Título. No estado inicial o usuário entra com o título do CD. Após o usuário entrar com o título, o sistema apresenta o conjunto de CDs que combinam com o título fornecido. A partir deste conjunto, o usuário inclui na cesta de compras aquele que ele deseja comprar.

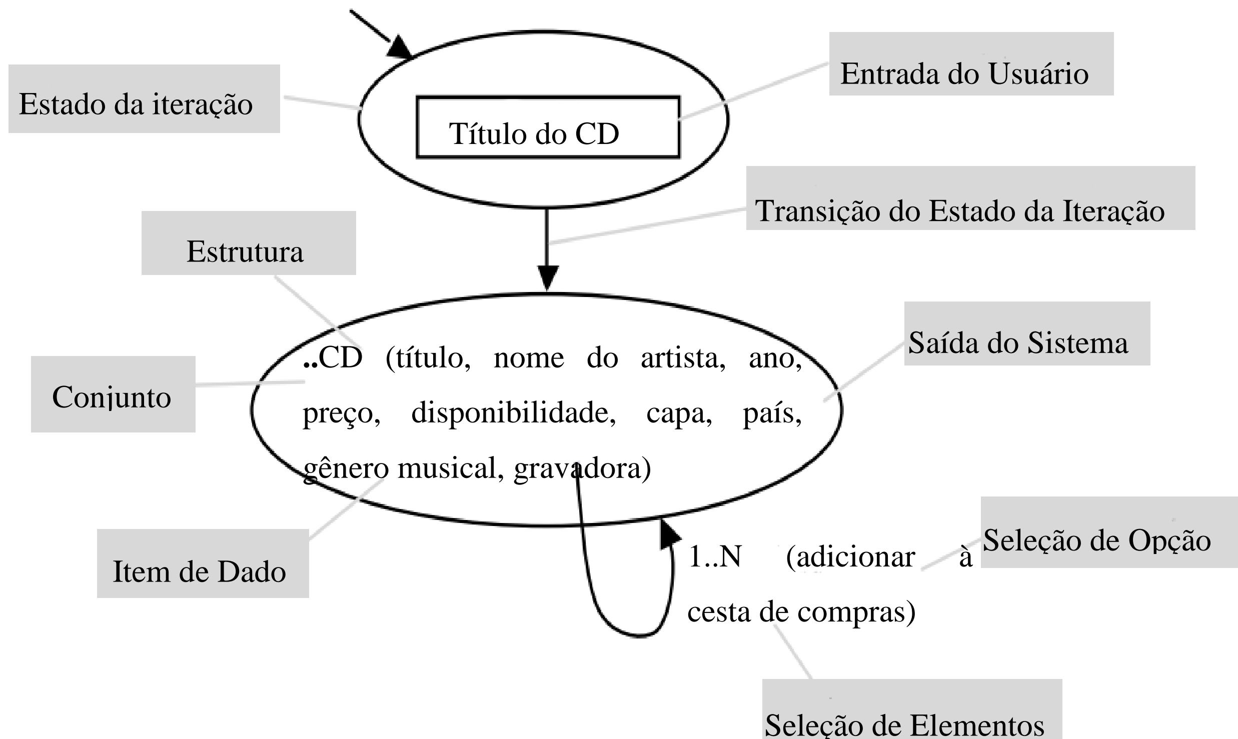



**Figura 2.1 - Exemplo de UID. [VILAIN 2003]**

## 2.3. Ferramenta de Mapeamento UID2JSF

A ferramenta UID2JSF tem por objetivo efetuar o mapeamento de UIDs (User Interaction Diagrams) para páginas JSF (Java Server Faces). Para que este mapeamento possa ocorrer com sucesso, Damiani (2011) desenvolveu um protótipo capaz de ler arquivos XML que representam os UIDs e transformam as informações obtidas em páginas JSF.

Vilain (2003), visando facilitar o armazenamento e também intercâmbio das instâncias dos UIDs entre aplicações, desenvolveu a especificação de uma DTD (Document Type Definition) para que os UIDs pudessem ser representados através de arquivos XML (Extensible Markup Language). É a partir desses arquivos XML que as interfaces serão geradas.

Durante o desenvolvimento da ferramenta UID2JSF, foram definidas regras para mapear cada componente dos UIDs em componentes das páginas JSF [DAMIANI, 2011]. A Tabela 2.1 apresenta de uma forma resumida as regras de mapeamento utilizadas pela ferramenta UID2JSF para efetuar o mapeamento de cada elemento de um UID para componentes JSF.

**Tabela 2.1 - Tabela de mapeamento dos elementos do UID para componentes JSF**

| Entradas do Usuário | |
|---|---|
| **UID** | **JSF** |
| Item de Dado | Input Text |
| Estrutura | Input Text com um Output Label<br>Panel Grid com um Output Label |
| Conjunto de Itens de Dado | Um Input Text para cada item de dado do conjunto<br>Apenas um Input Text |
| Conjunto de Estruturas | Data Table com uma coluna do tipo Input<br>Text para cada elemento da estrutura<br>Apenas uma Estrutura de entrada de usuário |
| Entrada de usuário Enumerada | Select One Radio<br>Select Many Checkbox |
| Seleção dentre dois Itens de Dados (OR) | Select Many Checkbox |
| Seleção de um Item de Dado (XOR) | Select One Radio |
| **Saídas do Sistema** | |
| **UID** | **JSF** |
| Texto | Output Text |
| Item de Dado | Output Text<br>Command Link |
| Estrutura | Output Text<br>Command Link<br>Panel Grid com Output Label |
| Conjunto de Itens de Dado | Data Table com apenas uma coluna. |
| Conjunto de Estruturas | Data Table com uma coluna para cada elemento da estrutura. |
| **Demais Elementos** | |
| **UID** | **JSF** |
| Estado da Interação, Estado Inicial da Interação | Página XHTML com Form |
| Sub-estado de um Estado de Interação | Form |
| Transição com Seleção de N Elementos | Command Link<br>Select Many Checkbox |
| Transição com Seleção da Opção X | Command Link |

### 2.3.1. Exemplo de mapeamento UID2JSF

Nesta seção será apresentado um exemplo de um UID e em seguida o seu mapeamento para páginas JSF utilizando a ferramenta UID2JSF.

O UID mapeado é correspondente ao caso de uso Mostrar gravações de um CD, mostrado a seguir.

- Caso de Uso: Mostrar gravações de um CD

  1. Para um dado CD, o sistema mostra um conjunto com todas as suas gravações. Para cada música, é apresentado o nome da música, tempo de duração, cantor, compositor e letra.

  2. Se o usuário desejar escutar um trecho de uma música, uma gravação pode ser selecionada.

  A Figura 2.2 apresenta o UID "*Mostrar gravações de um CD*".

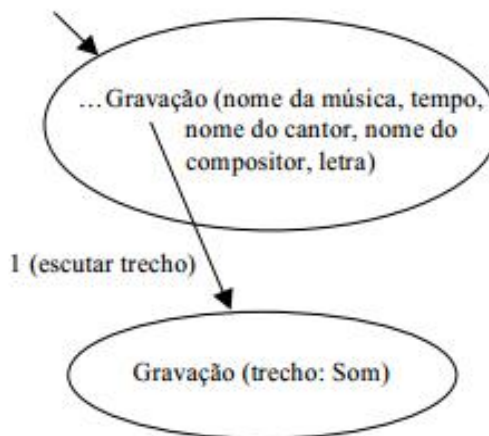
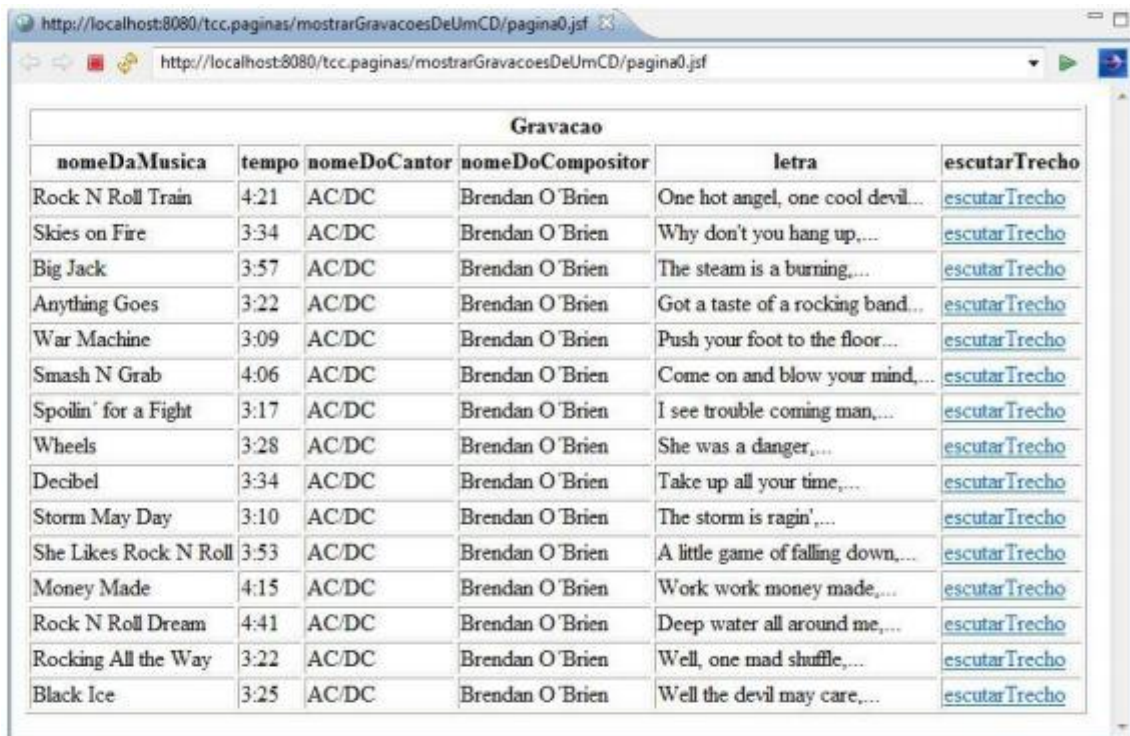



**Figura 2.2 - Mostrar gravações de um CD.**

A Figura 2.3 apresenta a página JSF gerada a partir do estado inicial de interação do UID apresentado anteriormente. O estado está representado por um componente *"form"* que não pode ser visualizado na imagem. O conjunto de estruturas originou a tabela e cada item de dado da estrutura "Gravação" ficou representado por uma coluna da tabela, assim como a transição

"escutar trecho" que é representada pela última coluna. Já os elementos do conjunto, isto é, as estruturas do tipo gravação, são as linhas da tabela.



http://localhost:8080/tcc.paginas/mostrarGravacoesDeUmCD/pagina0.jsf

| Gravacao | | | | | |
|---|---|---|---|---|---|
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |
| Rock N Roll Train | 4:21 | AC/DC | Brendan O´Brien | One hot angel, one cool devil... | escutarTrecho |
| Skies on Fire | 3:34 | AC/DC | Brendan O´Brien | Why don't you hang up,... | escutarTrecho |
| Big Jack | 3:57 | AC/DC | Brendan O´Brien | The steam is a burning,... | escutarTrecho |
| Anything Goes | 3:22 | AC/DC | Brendan O´Brien | Got a taste of a rocking band... | escutarTrecho |
| War Machine | 3:09 | AC/DC | Brendan O´Brien | Push your foot to the floor... | escutarTrecho |
| Smash N Grab | 4:06 | AC/DC | Brendan O´Brien | Come on and blow your mind,... | escutarTrecho |
| Spoilin´ for a Fight | 3:17 | AC/DC | Brendan O´Brien | I see trouble coming man,... | escutarTrecho |
| Wheels | 3:28 | AC/DC | Brendan O´Brien | She was a danger,... | escutarTrecho |
| Decibel | 3:34 | AC/DC | Brendan O´Brien | Take up all your time,... | escutarTrecho |
| Storm May Day | 3:10 | AC/DC | Brendan O´Brien | The storm is ragin',... | escutarTrecho |
| She Likes Rock N Roll | 3:53 | AC/DC | Brendan O´Brien | A little game of falling down,... | escutarTrecho |
| Money Made | 4:15 | AC/DC | Brendan O´Brien | Work work money made,... | escutarTrecho |
| Rock N Roll Dream | 4:41 | AC/DC | Brendan O´Brien | Deep water all around me,... | escutarTrecho |
| Rocking All the Way | 3:22 | AC/DC | Brendan O´Brien | Well, one mad shuffle,... | escutarTrecho |
| Black Ice | 3:25 | AC/DC | Brendan O´Brien | Well the devil may care,... | escutarTrecho |

**Figura 2.3 - Página referente ao estado inicial do UID Mostrar gravações de um CD**

A Figura 2.4 apresenta a página JSF referente ao segundo estado de interação do UID apresentado anteriormente. O estado de interação foi mapeado para um componente *"form"*, que não pode ser visto. A estrutura "Gravação" foi mapeada para um componente *"panelGrid"*, com um *"outputLabel"* onde seu valor é o nome da estrutura; e o item de dado "trecho" da estrutura foi mapeado para um *"outputText"* onde o valor é o nome do item de dado. A ferramenta não trata os domínios do tipo itens de dados. Ela simplesmente gera um *"outputText"* para cada um destes domínios. O projetista ficará encarregado de modificar o componente referente ao item de dado "trecho" para, por exemplo, um player de música.

**Figura 2.4 - Página gerada para o segundo estado de iteração do UID Mostrar gravações de um CD.**

# 2.4. Selenium

Selenium é uma ferramenta *open source* voltada, principalmente, a auxiliar na automação de testes em aplicações web. O framework Selenium é suportado pelos principais browsers utilizados na atualidade, é multiplataforma e permite que os usuários escrevam testes automatizados utilizando diferentes tipos de linguagem [SELENIUM, 2013].

O Selenium fornece uma grande gama de funções de teste com o intuito de simular as ações que um usuário comum faria em seu navegador. Essas funções permitem que o Selenium consiga localizar elementos presentes na estrutura de uma página web e a partir da localização, é possível trabalhar com esses elementos para, por exemplo, validar as informações contidas nos elementos. Outro ponto interessante do Selenium é a possibilidade de executar um mesmo teste em navegadores que estejam rodando em diferentes plataformas. Isto é feito através do Selenium-rc, o qual será explicado na sequência.

## 2.4.1. Selenium RC

O Selenium RC (Selenium Remote Control) foi por muito tempo o principal projeto do Selenium. A ferramenta foi criada em 2004 pelo desenvolvedor de software Jason Huggins, o qual trabalhava na empresa ThoughtWorks e ao se cansar do retrabalho gerado por testes internos manuais, percebeu que havia uma melhor forma de utilizar seu tempo gasto com testes. Huggins desenvolveu uma biblioteca de JavaScripts que podia se comunicar com as páginas do

browser, permitindo com que ele pudesse executar automaticamente diferentes testes em diferentes browsers.

A ferramenta Selenium RC é composta por:

#### 2.4.1.1. Selenium Server

Responsável por receber comandos enviados por programas de testes, interpretá-los, e responder esses comandos com o resultado gerado após rodar os testes. Os comandos dos programas de teste fazem simples requisições HTTP GET/POST para se comunicar com o servidor.

#### 2.4.1.2. Client Libraries (Bibliotecas de cliente)

Bibliotecas que provêem uma interface entre diferentes linguagens de programação e o servidor Selenium RC. Para cada linguagem suportada existem diferentes bibliotecas de clientes. As diferentes bibliotecas possuem conjuntos de funções que fazem com que o usuário destas consiga rodar os comandos do Selenium a partir de um programa próprio.

As bibliotecas de cliente são encarregadas de enviar ao servidor, ações específicas a serem processadas ou testes para serem feitos em cima de uma aplicação que esteja sendo testada, e posteriormente receber o resultado retornado pelo servidor. Esse resultado pode ser armazenado em alguma variável e validações podem ser feitas sobre ele.

A criação de um programa de teste não é nada muito complicada. Basta que a partir de uma das bibliotecas de cliente escolhida, escreva-se um programa que rode um grupo de comandos Selenium. Opcionalmente pode ser utilizada a ferramenta Selenium-IDE para a geração destes comandos, isso porque esta ferramenta é capaz de gravar ações de clique que usuário faça em um navegador e exportar essas ações em forma de comandos utilizados pelas bibliotecas de cliente.

A Figura 2.5 apresenta o modelo de arquitetura do Selenium RC. Ela é composta por:

- Um servidor que automaticamente inicializa e finaliza instâncias de navegadores, e atua como um Proxy HTTP para requisições feitas pelos navegadores.
- Bibliotecas de cliente utilizando linguagens específicas que podem ser escolhidas pelo desenvolvedor de teste.

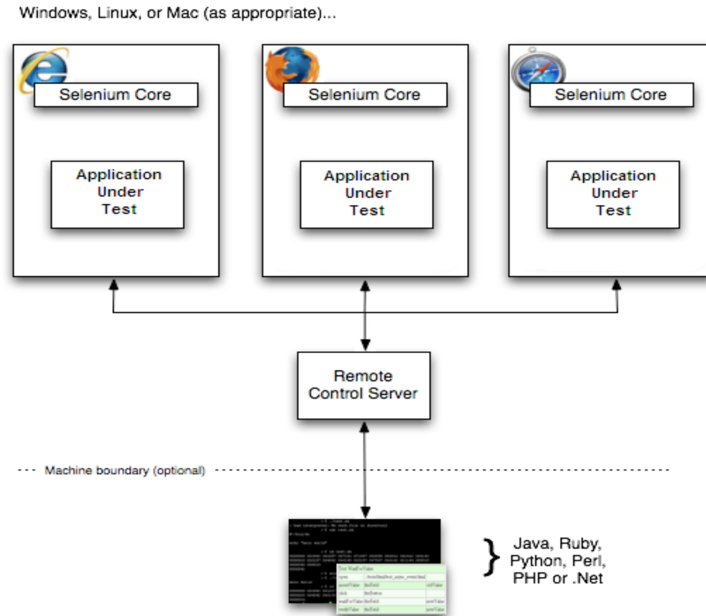



**Figura 2.5 - Modelo de arquitetura Selenium RC**

Em 2006 um engenheiro da Google chamado Simon Stewart começou a trabalhar em um projeto que visava eliminar alguns problemas pelos quais os testadores da Google passavam ao utilizar o Selenium. Simon desejava obter uma ferramenta que se comunicasse diretamente com o browser utilizando métodos 'nativos' para comunicação tanto com o browser quanto com o sistema operacional.

Atualmente a ferramenta Selenium RC entrou em desuso após junção do Selenium com a ferramenta WebDriver, a qual será explicada a seguir. Porém, mesmo em desuso, a ferramenta continua com suporte ativo (principalmente em modo de manutenção) e ainda dispõe de algumas funcionalidades que podem ainda não estar disponíveis em sua nova versão, como o suporte a várias linguagens e suporte a maioria dos browser disponíveis no mercado.

### 2.4.2. Selenium WebDriver

Selenium-WebDriver é na verdade a versão 2.0 do Selenium. Porém, como nessa nova versão houve a integração da API WebDriver e ela acabou dando uma cara nova ao projeto, os projetistas acharam justo a adoção do nome da nova API como nome principal da ferramenta. Nessa versão 2.0, o Selenium busca fornecer aos usuários uma interface simples e concisa, além de trazer melhorias em relação às deficiências antes apresentadas pelo Selenium-RC. A

ferramenta Selenium-WebDriver foi desenvolvida para prover um melhor suporte a páginas web dinâmicas onde componentes da página podem mudar de estado sem que a página seja recarregada. O intuito da ferramenta é o de prover uma API bem desenvolvida, que atenda as necessidades de teste de interface em diferentes browsers e que seja orientada a objetos, a fim de fornecer um maior suporte às novas demandas de testes que venham a surgir com o avanço da tecnologia [SELENIUM, 2013].

A utilização da ferramenta pode ser feita através do download de uma das APIs do Web Driver que estão disponíveis nas seguintes linguagens: Java, C#, Ruby, Perl, PHP e Groovy**.** Esta API provê um driver de comunicação padrão, responsável por fazer a comunicação com o navegador Firefox. Entretanto, caso o testador deseje executar seus testes em diferentes browsers, existe a possibilidade de fazer o download de outros drivers que permitem a comunicação com os seguintes browsers: IE, Chrome,Opera. Além de existir também, drivers para as interfaces dos sistemas operacionais: Android e iOS.

## 2.5. Padrão de Design Page Object (objeto de página)

A utilização do framework de testes de interface WebDriver é algo relativamente simples. Basta que a API seja baixada para a máquina onde ocorrerão os testes e que seja integrada à *IDE* (Integrated Development Environment) de preferência do desenvolvedor de testes. Porém, para que esses testes não fiquem bagunçados e escritos de qualquer forma, é interessante que eles sejam escritos orientados às páginas da aplicação, daí vem o termo objeto de página.

O padrão de design Page Object (objeto de página), apresentado pelos desenvolvedores da ferramenta Selenium, tem por objetivo modelar páginas HTML em classes Java, permitindo que elementos de uma página web possam ser acessados através dessas classes geradas. Isso ajuda a organizar lógicas, reduzir a quantidade de código duplicado e melhorar a manutenibilidade dos testes [SELENIUM, 2013].

Devem-se seguir alguns princípios básicos para que se possa fazer bom proveito dos objetos de páginas. Abaixo estão alguns desses princípios [SELENIUM, 2013]:

- Objetos de página possuem métodos públicos que representam os serviços que a página oferece.

- Deve-se expor o mínimo possível as partes internas da página.
- Métodos retornam outros objetos de página.
- Devem-se programar apenas métodos que precisam ser testados.

A Figura 2.6 apresenta um exemplo de uma classe seguindo o design objeto de página. Esta classe faz a abstração da página de busca do Google, onde a anotação *"@FindBy"* indica que um elemento deve ser buscado na tela, a palavra "*how*" informa o tipo de busca desse elemento na tela (tipos mais comuns são por id, name e xpath), e a palavra *"using"* indica qual o nome desse componente na tela. Caso esse elemento seja encontrado, ele é injetado na variável que está definida na linha logo abaixo, neste caso, na variável *"searchBox"*. Em resumo, essa classe disponibiliza a possibilidade de acessar o campo de busca da tela de pesquisa do Google, inserir uma informação neste campo e então submeter esse campo.

```
package org.openqa.selenium.example;

import org.openqa.selenium.By;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.How;
import org.openqa.selenium.WebElement;

public class GoogleSearchPage {
    // The element is now looked up using the name attribute
    @FindBy(how = How.NAME, using = "q")
    private WebElement searchBox;

    public void searchFor(String text) {
        // We continue using the element just as before
        searchBox.sendKeys(text);
        searchBox.submit();
    }
}
```

**Figura 2.6 - Exemplo de objeto de página [SELENIUM, 2013]**

## 2.6. JavaServer Faces

JavaServer Faces (JSF) é um framework desenvolvido em Java que foi projetado para simplificar o desenvolvimento de aplicações Web. Este framework pode ser utilizado na criação de interfaces do usuário e páginas web, além de fazer a conexão desses componentes com objetos de negócio de forma mais inteligível. O framework também automatiza a comunicação entre JavaBeans e a navegação entre as páginas de um projeto[JSF, 2013]. A seguir serão apresentadas

duas características do JSF importantes no desenvolvimento do trabalho sendo apresentado: JavaBeans e Naming Container.

### 2.6.1. JavaBeans

JavaBeans, são classes responsáveis por expor propriedades (atributos) e eventos (métodos) em um ambiente como JSF [JSF, 2013].

Um exemplo de Java Beans são as classes do tipo Managed Beans, onde as propriedades de um Managed Bean são valores nomeados de determinados tipos que podem ser modificados ou recuperados através de métodos get / set. As Figuras 2.7 e 2.8 apresentam, respectivamente, um trecho de código de uma página “XHTML” e um trecho de uma classe Java que funciona como um Managed Bean. Na imagem 2.7, três diferentes tipos de acesso a um Managed Bean acontecem. O acesso marcado com número 1 indica que no momento de carregamento da página em questão, por se tratar de um campo do tipo Datatable, o framework JSF irá procurar por um Managed Bean que esteja mapeado com o nome “*p1MB*”, e nessa classe buscará por um método que forneça acesso a variável “valueMusica”. Neste caso o nome do método seria “getValueMusica” e traria uma lista de valores para popular a tabela (Figura 2.8). Já o acesso marcado com o número 2, buscará, através do objeto nomeado “p1MB”, um valor do tipo booleano, isso porque está associado a um atributo HTML do tipo *“rendered”*, o qual espera um valor verdadeiro / falso para definir se renderiza ou não o componente na tela. Por convenção, no Java o acesso a variáveis do tipo booleano é feita com o prefixo *“is”* + *“nome da variável iniciando com letra maiúscula”*. Por fim, o acesso marcado com o número 3 apresenta um acesso a uma variável mais simples onde para se obter o valor dessa variável no Managed Bean, usa-se o prefixo *“get”* + *“nome da variável iniciando com letra maiúscula”*.*P*ara alterar o valor usa-se o prefixo *“set”* + *“nome da variável iniciando com letra maiúscula”* .

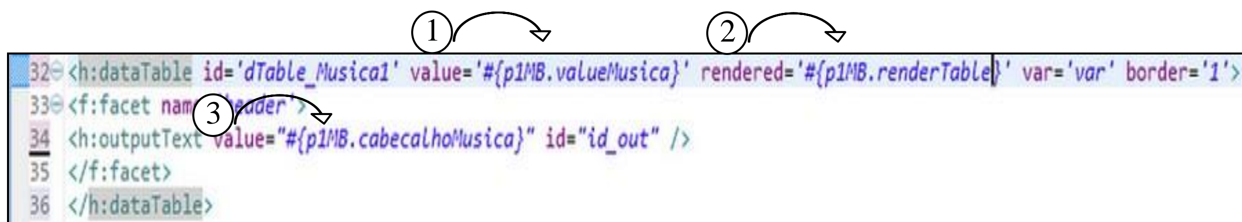



**Figura 2.7 - Trecho de página XHTML**

```
10 @ManagedBean
11 @SessionScoped
12 public class P1MB {
13
14     private String cabecalhoMusica;
15     private List<Musica> valueMusica;
16     private boolean renderTable;
17
18     public void setCabecalhoMusica(String cabecalhoMusica) {
19         this.cabecalhoMusica = cabecalhoMusica;
20     }
21
22     public String getCabecalhoMusica() {
23         return cabecalhoMusica;
24     }
25
26     public void setValueMusica(List<Musica> valueMusica) {
27         this.valueMusica = valueMusica;
28     }
29
30     public List<Musica> getValueMusica() {
31         return valueMusica;
32     }
33
34     public boolean isRenderTable() {
35         return renderTable;
36     }
37
38 }
```

**Figura 2.8 - Exemplo de classe Managed Bean**

### 2.6.2. Naming Container

Naming container é uma interface do tipo marcador. Componentes JSF que implementam a estrutura Naming Container possuem a característica de garantir que elementos que estejam dentro de um componente do tipo Naming Container (elementos filhos), possuam IDs únicos, evitando assim que sejam declarados componentes com IDs duplicados dentro de uma página [MCCLANAHAN, et al., 2004]

Elementos do tipo Naming Container possuem a característica de concatenar seu ID ao de seus componentes filhos. Componentes como **h:form**, **f:subview** e **h:dataTable** são alguns exemplos práticos de componentes do tipo Naming Container. A Figura 2.9 apresenta um exemplo de funcionamento de um componente deste tipo, onde o ID do componente Form, que é um Naming Container, é concatenado ao ID do seu componente filho após a geração do código declarado.

```
<h:form id="myForm">
    <h:inputText id="test" />
</h:form>
```

```
<form id="myForm" name="myForm" method="post">
    <input id="myForm:test" type="text" name="myForm:test" />
</form>
```

**Figura 2.9 – Exemplo Naming Container.**

# 3. Trabalhos Relacionados

A partir do conceito e crescimento do uso de ferramentas de MDA (Model Driven Architecture), intensificaram-se os estudos voltados a automatização do processo de teste de software. Muitos desses estudos estão direcionados a melhorar o processo de geração de testes baseados em modelos.

Ferramentas de teste baseadas em modelo vêm ganhando destaque, principalmente devido ao seu potencial em automatizar a geração de casos de teste e ao crescimento do uso de abordagens de engenharia de software baseado em modelo.

Entretanto, não foi encontrado nenhum trabalho que fizesse a automatização de testes a partir da transformação de UIDs.

A seguir serão apresentados trabalhos que efetuam a geração de testes de interface através de modelos.

## 3.1. Reverse Engineering of GUI Models for Testing

Este trabalho apresenta uma ferramenta que tem por finalidade extrair modelos de interfaces através de engenharia reversa e então usar esses modelos no processo de teste de software [GRILO, et al., 2010].

O processo de engenharia reversa de interfaces começa com a construção de um modelo preliminar da aplicação. Este modelo é construído através de interações de um usuário com a interface da aplicação e o armazenamento das ações e resultados gerados pelas interações. Logo que esse modelo preliminar é extraído, o testador deve completar o modelo e validar se o que foi gerado está de acordo com o comportamento ideal da aplicação. O conteúdo obtido através da engenharia reversa é mantido em um arquivo XML para que depois possa ser traduzido para uma linguagem de especificação que possibilite gerar modelos em diferentes linguagens [GRILO, et al., 2010].

A partir do modelo gerado com a engenharia reversa é possível obter informações estruturais sobre as interfaces (estruturas hierárquicas, fluxos de controle, fluxos de navegação,

ações permitidas, etc.). Quando um modelo não contiver todas as informações necessárias, essas devem ser adicionadas manualmente pelo testador [GRILO, et al., 2010].

Informações de mapeamento são adquiridas durante o processo de engenharia reversa e enquanto o testador completa manualmente o modelo. Após adquirir as informações de mapeamento, essas são salvas em um novo arquivo XML o qual mantém informações sobre características físicas dos controles das interfaces (para poder identificar controles existentes entre telas durante a execução de casos de teste), e mantém também informação sobre o mapeamento entre ações concretas. Ações concretas são capazes de simular ações de usuários reais sobre controles de uma interface [GRILO, et al., 2010].

Por fim, o modelo final do processo de engenharia reversa é utilizado para gerar automaticamente um conjunto de casos de teste, que são conjuntos de segmentos de teste os quais modelam ações de usuário intervaladas com operações que checam se o resultado dessas ações está de acordo com o esperado [GRILO, et al., 2010].

## 3.2. User Interface Test Automation and its Challenges in an Industrial Scenario

Pradhan (2011) cita que MDT (Model Driven Testing) é uma forma de teste baseado em modelo que é baseada em MDE (Model Driven Engineering) ou MDA, ou em outras palavras, usam metamodelos, modelos e conjuntos de modelos de regras de transformação.

Pradhan (2011) utiliza em sua tese a metodologia proposta em (MEMON, et al., 2001), onde foram aplicados princípios de MDA nas verificações e validações de software utilizando diagramas de sequência.

Na metodologia citada, foi proposto escrever casos de teste utilizando diagramas de sequência e finalmente gerar casos de teste específicos a partir desses diagramas fazendo o uso de MDA.

A Figura 3.1 apresenta o modelo de transformações que a aplicação efetua até a geração dos casos de teste. Para a geração dos casos de teste a ferramenta parte de um modelo UML, faz

a transformação para um modelo xUnit utilizando a ferramenta Tefkat e, em seguida, utiliza as regras do MOFScript para gerar os casos de teste com JUnit baseados no modelo xUnit gerado.

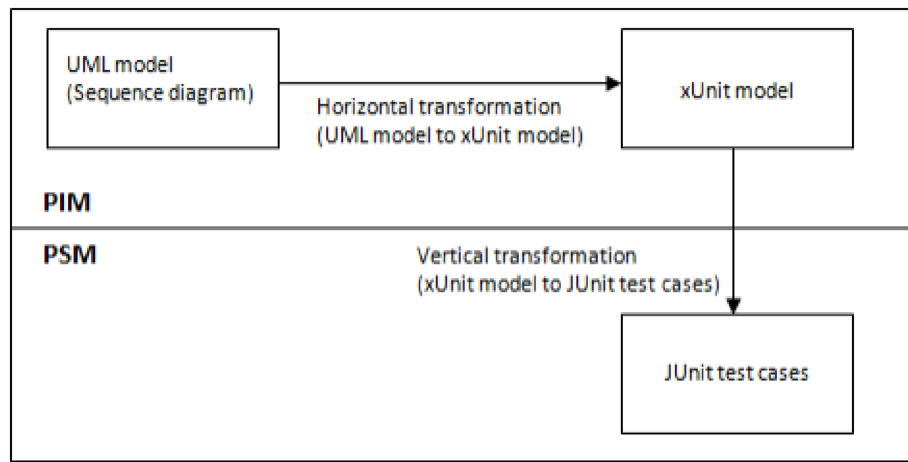



**Figura 3.1 – Modelo de transformação de diagramas de sequencia em casos de teste.**

## 3.3. Model Based Testing Of Web Applications

Nesta seção será apresentado o trabalho desenvolvido por ACHKAR(2010). O trabalho tem como objetivo a geração de testes de interfaces web a partir de Máquina de Estados Finitos (MEF) para um sistema médico na Austrália.

Segundo ACHKAR (2010), o significado do termo Finitos no contexto de testes a partir de Máquina de Estados Finitos, simplesmente indica o número limitado de estados de uma aplicação sob testes. Portanto, um modelo MEF é representado como um grafo ou um mapa de estados composto por um limitado (finito), número de nós (estados) interligados através de arestas diretas (transições - ações).

Para a construção de modelos MEF para aplicações web são levadas em consideração regras simples, como:

- Cada página da aplicação pode ser equiparada a um estado no modelo.
- Cada aba de cada página pode ser considerada como um sub-Estado da página.

- Cada ação que altera a página de uma forma importante resulta em uma mudança de estado.
- Cada ação tomada dentro de uma página pode ser equiparada a uma transição no modelo.

Para execução do trabalho proposto, ACHKAR (2010) contou com o auxílio da ferramenta TestOptimal™ (http://testoptimal.com/TestOptimalHome.html), uma ferramenta comercial direcionada a modelagem MEF e conta tanto com uma versão com licença completa quanto com uma versão de teste por 30 dias.

### 3.3.1. TestOptimal™

A ferramenta TestOptimal™, que aqui será chamada de TO, apresenta uma interface gráfica com o usuário e uma forma de popular modelos MEF através de scripts em Java ou XML, que preenchem os modelos com os métodos de comportamento e verificação necessários para interagir e testar aplicações web. Esta interação com as aplicações web é feita através do Selenium Server o qual está incluso na instalação da ferramenta TO.

### 3.3.2. Utilizando a ferramenta de modelagem

A modelagem MEF da aplicação é feita através da ferramenta TO, que gera uma esqueleto abstrato ou uma representação lógica de framework do potencial comportamento da aplicação a ser testada, porém isso ainda não é o suficiente para prover qualquer interação concreta com o mundo real. Esta interação concreta deve ser feita manualmente.

Depois de feita a modelagem utilizando a ferramenta TO, esta ferramenta produz um arquivo chamado mScript o qual é baseado em XML. Este arquivo permite que o testador possa introduzir os dados referentes à aplicação sendo testada. Um exemplo do arquivo mScript gerado pode ser visto na Figura 3.2.

```
Model mScript  xml viewer | compile |
269         <assert lid="269" value1="$containsText('$getData('TextDS','ErrAddDoctor')')" op="eq" value2="$getData('BadDocDS','ErrAddDoctor
270         <action lid="270" code="$nextDataSetRow('BadDocDS')"/>
271       </script>
272     </transition>
273     <transition lid="273" event="RegisterReject">
274       <script lid="274" type="prep">
275         <log lid="275" message="RegisterReject with 1 Doc iteration $getSysVar('curTransTravCount'):"/>
276         <action lid="276" code="$type('$getData('ElementNamesDS','UsernameEdit')','$getData('BadDS','Username')')"/>
277         <action lid="277" code="$type('$getData('ElementNamesDS','PasswordEdit')','$getData('BadDS','Password')')"/>
278         <action lid="278" code="$type('$getData('ElementNamesDS','ConfirmPasswordEdit')','$getData('BadDS','ConfirmPwd')')"/>
279         <action lid="279" code="$type('$getData('ElementNamesDS','ContactTitleEdit')','$getData('BadDS','Title')')"/>
280         <action lid="280" code="$type('$getData('ElementNamesDS','ContactGivenNameEdit')','$getData('BadDS','GivenName')')"/>
281         <action lid="281" code="$type('$getData('ElementNamesDS','ContactSurnameEdit')','$getData('BadDS','Surname')')"/>
282         <action lid="282" code="$type('$getData('ElementNamesDS','ContactPositionEdit')','$getData('BadDS','Position')')"/>
283         <action lid="283" code="$type('$getData('ElementNamesDS','ContactPhoneEdit')','$getData('BadDS','Phone')')"/>
284         <action lid="284" code="$type('$getData('ElementNamesDS','ContactMobileEdit')','$getData('BadDS','Mobile')')"/>
285         <action lid="285" code="$type('$getData('ElementNamesDS','OrganisationNameEdit')','$getData('BadDS','PracticeName')')"/>
286         <action lid="286" code="$type('$getData('ElementNamesDS','AddressLine1Edit')','$getData('BadDS','AddressLine1')')"/>
287         <action lid="287" code="$type('$getData('ElementNamesDS','AddressLine2Edit')','$getData('BadDS','AddressLine2')')"/>
288         <action lid="288" code="$type('$getData('ElementNamesDS','SuburbEdit')','$getData('BadDS','Suburb')')"/>
289         <action lid="289" code="$selectOption('$getData('ElementNamesDS','StateList')','value=$getData('BadDS','State')')"/>
290         <action lid="290" code="$type('$getData('ElementNamesDS','PostalCodeEdit')','$getData('BadDS','PostCode')')"/>
291         <action lid="291" code="$type('$getData('ElementNamesDS','EmailEdit')','$getData('BadDS','Email')')"/>
292         <action lid="292" code="$type('$getData('ElementNamesDS','OrgEmailEdit')','$getData('BadDS','OrgEmail')')"/>
293         <action lid="293" code="$selectOption('$getData('ElementNamesDS','BusinessUnitList')','value=$getData('BadDS','MyProvider')')"/
294         <action lid="294" code="$type('$getData('ElementNamesDS','NotesEdit')','$getData('BadDS','Notes')')"/>
295       </script>
```

**Figura 3.2 – mScript [ACHKAR, 2010]**

O modelo apresentado na Figura 3.2 mostra que os dados a serem testados não estão explicitamente inseridos no modelo gerado. Isso foi feito para permitir o reuso do modelo. A linha 276 pode ser usada como exemplo para mostrar essa possibilidade de reuso. O conteúdo gerado para a linha 276 foi:

<actionlid="276"
code="$type('$getData('ElementNamesDS','UsernameEdit')','$getData('BadDS','Username')')"/>

Onde para:

- $getData('ElementNamesDS','UsernameEdit'): o valor 'ElementNamesDS' deve ser substituído com o identificador do componente na interface web.

- $getData('BadDS','Username'): o valor 'BadDS' deve ser substituído com o valor que o campo da interface web deve receber.

### 3.3.3. Execução do modelo

A execução dos modelos de teste gerados fica também por conta da ferramenta TO. Com a conclusão do preenchimento do mScript gerado pela ferramenta, este estará pronto para ser executado e iniciar os testes de interface da aplicação. Segundo ACHKAR (2010) este processo é relativamente simples e dependendo da complexidade do modelo e extensão do modelo, ele completará sua execução rapidamente.

# 4. Decisões de Design

Neste capítulo serão explicadas as decisões de design utilizadas para o desenvolvimento da ferramenta proposta neste trabalho. A ferramenta será chamada de UID2JSF 2.0 e tem por objetivo adicionar à ferramenta UID2JSF a funcionalidade de criação de classes Managed Bean com dados mockados (fictícios) e a geração de testes automatizados de interface. Para que essa geração possa ser feita, serão usadas informações de componentes JSF extraídas do mapeamento de um UID para JSF, feito pela ferramenta UID2JSF. Como resultado final, o protótipo de projeto irá gerar automaticamente as seguintes estruturas:

- Páginas JSF com extensão “XHTML” (já era feito pela ferramenta UID2JSF).
- Classes Managed Bean, para gerenciar os dados das páginas JSF.
- Classes de objeto de página, para gerenciar a comunicação dos testes com as páginas JSF.
- Classes de teste para fazer a invocação de funcionalidades de uma página através de objetos de página e validar se as funcionalidades das páginas testadas estão corretas.

Primeiramente serão explicadas as regras gerais para a definição de IDs (Identificadores) em componentes JSF e quais regras serão criadas para a geração desses IDs. Os identificadores em componentes JSF são importantes para que a ferramenta Selenium consiga, no momento da execução de testes sobre uma página, localizar os componentes a partir de seu identificador. Na sequência serão apresentados os processos de extração de informações de componentes JSF gerados pela ferramenta UID2JSF, estes são necessários para a geração de classes e casos de testes. Em seguida serão apresentados os processos para geração de classes e casos de teste a partir das informações extraídas no processo explicado anteriormente. Por fim, será mostrado como as classes Managed Beans e suas auxiliares são geradas, classes estas que serão úteis para agilizar o processo (evitando trabalho manual) de apresentação das páginas JSF e da execução dos testes gerados pela ferramenta aqui sendo proposta.

## 4.1. Definição de identificadores para elementos JSF

Os identificadores de componentes são gerados no momento da transformação de uma informação de um UID para um componente JSF. Durante essa etapa de transformação é

possível obter as informações necessárias relativas ao componente gerado e sua hierarquia de componentes pais que já foram gerados.

De acordo com a especificação 4.1 do HTML [RAGGETT, et al., 1999], a definição do atributo *ID* para um elemento deve começar com uma letra ([A-Za-z]), essa letra pode ser seguida por qualquer número de letras, dígitos ([0-9]), hífens ("-"), underscores ("_"), dois pontos (":"), e pontos ("."). O identificador deve também ser definido de forma que seja único dentro de um arquivo HTML.

Para respeitar as regras acima citadas, o processo de geração de identificadores para elementos JSF irá respeitar um padrão de prefixação (tabela 4.1) definido neste trabalho. Esse padrão será utilizado para que cada identificador criado tenha uma relação direta com tipo de componente cujo ID será definido. A definição de IDs sempre será apresentada da seguinte forma: **Prefixo do componente** + "_" + **ID vindo do UID** + "_" + **número incremental por tipo de elemento**. O único elemento JSF que não seguirá o mesmo padrão é o elemento do tipo Form. Este elemento irá utilizar, além do seu prefixo padrão, o nome da página onde ele está inserido. A Figura 4.1 mostra um exemplo de definição de ID para um elemento do tipo form dentro de uma página JSF chamada P1. Como pode ser observado na Figura 4.1, o nome da página foi concatenado ao identificador do form1. Isso ajuda a garantir a localização de um form, visto que em uma operação entre duas páginas diferentes, ambas poderiam ter um elemento form com IDs idênticos, porém o nome da página como prefixo garante a qual página cada form pertence.

**Tabela 4.1 - Prefixo para definição de IDs para elementos JSF**

| | |
|---|---|
| *COLUMN* | "col_" |
| *COMMANDLINK* | "cLink_" |
| *DATATABLE* | "dTable_" |
| *FORM* | "form_" |
| *INPUTTEXT* | "iText_" |
| *OUTPUTLABEL* | "outLbl_" |
| *OUTPUTLINK* | "outLink_" |
| *OUTPUTTEXT* | "outText_" |
| *PANELGRID* | "pGrid_" |
| *SELECT* | "select_" |
| *SELECTITEM* | "sItem_" |

| *SELECTITEMS* | "sItems_" |
|---|---|
| *SELECTMANYCHECKBOX* | "sMany_" |
| *SELECTONERADIO* | "sRadio_" |

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN' 'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
<html xmlns='http://www.w3.org/1999/xhtml'  xmlns:h='http://java.sun.com/jsf/html'  xmlns:f='http://java.sun.com/jsf/core'>
<h:head>
    <title>P1</title>
</h:head>
<h:body>
    <h:form id='P1_form1'>
        <h:outputText value="Descricao" id="outText_014" />

        <h:outputLabel id='outLbl_CD3' value='#{p1MB.labelCD}'>
            <h:panelGrid columns='1'>
                <h:outputText value="descricao" id="outText_descricao15" />

                <h:outputText value="artista" id="outText_artista16" />

                <h:outputText value="ano" id="outText_ano17" />

                <h:outputText value="procedencia" id="outText_procedencia18" />

                <h:outputText value="label" id="outText_label19" />

                <h:outputText value="issn" id="outText_issn20" />

                <h:commandLink id='cLink_CD44' value='comprar'
                    action='PinitialState' styleClass='cLink_CD44' />

                <h:commandLink id='cLink_CD Space-separated list of CSS style class(es) to be applied when this element is rendered. This value must be p
                    styleClass='cLink_CD55'  through as the "class" attribute on generated markup.

            </h:panelGrid>
        </h:outputLabel>
    </h:form>
    <h:inputHidden id='noeWAz' />
</h:body>
</html>
```

**Figura 4.1 - Exemplo de geração de IDs com prefixo.**

## 4.2. Definição de identificadores de localização de elementos JSF em páginas JSF

Para a utilização da ferramenta Selenium (responsável pelos testes de interface) neste trabalho, foi necessária a adequação da definição de elementos JSF dentro de uma página JSF para que estes pudessem conter características a partir das quais o Selenium pudesse encontrar esses elementos. Essas características serão aqui chamadas de identificadores de localização. Os identificadores de localização criados nesse trabalho são:

- **id -** representa o valor definido para o atributo id de um componente JSF. Porém vale lembrar que devido à característica de componentes do tipo Naming Container, explicado no capítulo 2, alguns componentes podem herdar os identificadores de componentes pais.

Portanto, para que um elemento possa ser localizado pelo Selenium, seu identificador completo deve ser informado.

- **name** – este tipo de localização faz referência ao atributo *styleClass* dos componentes JSF. Portanto, como um mesmo valor de *styleClass* pode ser definido em diferentes componentes, o Selenium pode encontrar mais de um componente para um dado valor de *styleClass*.

Neste trabalho, a principal maneira de localizar elementos em uma página JSF será a partir do atributo *styleClass,* onde cada componente que permita a definição dessa propriedade, receberá como valor o mesmo valor que foi gerado para o campo identificador do componente JSF. Outra forma que o Selenium terá de localizar os elementos em uma página será através do atributo id do componente JSF. Como dito anteriormente, componentes que estejam definidos dentro de um componente do tipo Naming Container, irão ter o ID desse componente "pai" concatenado ao seu ID. Portanto, para que o Selenium possa localizar esse elemento, ele buscará pelo elemento usando um ID concatenado da seguinte forma: *ID do elemento pai + separador* ":" + *ID do componente a ser localizado.*

## 4.3. Definição de testes para páginas JSF

Seguindo definições de melhores práticas de programação, foi utilizado no desenvolvimento deste trabalho o padrão de projeto nomeado Page Objects (objetos de página), o qual foi apresentado anteriormente. De acordo com esse padrão, cada página JSF da aplicação deve possuir um objeto de página referente a ela.

Um objeto de página funciona como uma ligação entre os casos de teste e as interfaces a serem testadas. Esses objetos permitem que os casos de teste acessem elementos e informações presentes em uma página JSF.

Para que os objetos de página possam ser construídos, faz-se necessário obter as informações, características e funcionalidades que cada página possua. Neste trabalho essas informações foram obtidas a partir dos elementos JSF oriundos das transformações de UIDs feitas pela ferramenta UID2JSF.

A ideia de extração das informações necessárias para geração do projeto consiste em, no momento da criação do componente JSF, este deve ser registrado em uma classe do projeto, chamada **ProjectInfoHolder**. Esta foi definida neste trabalho e terá por função guardar a lista de componentes criados para cada página. Cada elemento armazenado terá consigo informações úteis no momento da criação dos testes, como por exemplo, o identificador de localização e a maneira pela qual o Selenium deverá localizar cada elemento.

### 4.3.1. Processo de criação de objetos de páginas

Para cada página JSF mapeada pela ferramenta UID2JSF, será criado um Objeto de página com o seguinte padrão de nomenclatura: ***P*** + *nome da página* + ***PageObject.*** Cada objeto de página gerado conterá atributos de classe do tipo WebElement, os quais farão referência a objetos presentes nas páginas JSF. Além disso, os objetos de página também conterão métodos de validação que serão executados pelos casos de teste gerados. A geração desses casos de teste serão explicados adiante.

#### 4.3.1.1. Localização de elementos JSF dentro de objetos de página

Neste trabalho, a forma de localização padrão de um elemento JSF pela ferramenta Selenium se dará através do atributo "styleClass" de cada componente JSF. Para isso, elementos que foram mapeados pela ferramenta UID2JSF e armazenados na classe ProjectInfoHolder, serão declarados como atributos de classe do tipo WebElement e receberão a anotação @FindBy(className="*ID componente*"), onde o valor entre aspas corresponderá ao ID gerado para o componente JSF. O componente JSF do tipo SelectOneRadio, responsável por renderizar *radio buttons* na tela, não suporta a definição do atributo "styleClass" para seus componentes filhos, portanto, quando for necessário trabalhar com esse tipo de campo o mesmo terá de ser localizado através do driver do Selenium, o qual consegue encontrar elementos pais e seus descendentes através do método *findElements(By)*.

A Figura 4.2 apresenta um exemplo de uma página JSF gerada a partir da ferramenta UID2JSF. Esta tela é composta por três componentes visíveis que, em ordem de apresentação, são: Outputlabel, InputText, CommandLink.

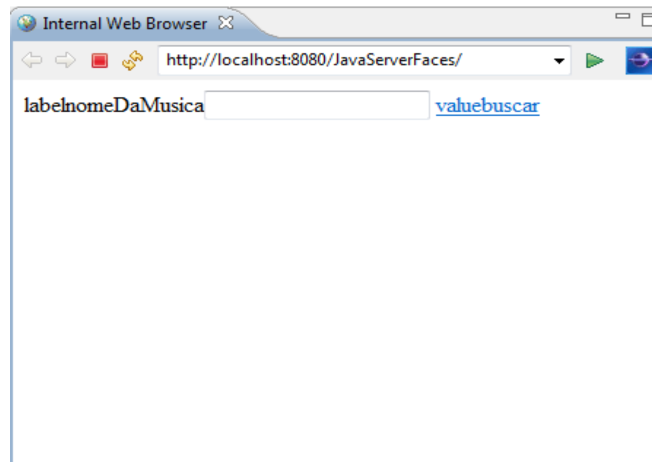



**Figura 4.2 - Exemplo de tela mapeada a partir da ferramenta UID2JSF**

Na sequência, a Figura 4.3 apresenta o objeto de página gerado para esta tela. Este objeto possui duas instâncias de objetos do tipo WebElement que fazem referência aos objetos Outputlabel e CommandLink presentes na Figura 4.2. O elemento InputText não está presente, pois não serão aplicados teste sobre ele.

```
public class PinitialStatePageObject extends AbstractPageObject {
    // OUTPUTLABEL
    @FindBy(className = "outLbl_nomeDaMusicalabel0")
    private WebElement outLbl_nomeDaMusicalabel0;
    // COMMANDLINK
    @FindBy(className = "cLink_00")
    private WebElement cLink_00;

    /**
     * CONSTRUCTOR
     **/
    public PinitialStatePageObject(WebDriver driver) {
        super(driver);
    }
```

**Figura 4.3 - Exemplo de objeto de página**

### 4.3.2. Métodos de teste para objetos de página

Métodos de página são métodos que auxiliam no processo dos testes de interface. Esses métodos são responsáveis por prover acesso a elementos de uma interface e serviços que esta possua, como por exemplo, a validação de textos em componentes, as possíveis transições que a página possa fazer de uma interface para outra, além de outras validações que o testador necessite fazer, bastando implementá-las.

Os métodos de testes a serem gerados automaticamente dentro dos objetos de página serão os métodos utilizados pelas classes de caso de teste no momento em que os testes forem executados. Cada método será invocado uma única vez no momento em que for executada a classe de casos de teste referente a cada página.

A Figura 4.4 apresenta os quatro tipos de teste de página desenvolvidos neste trabalho. O primeiro método funciona como um identificador para validação de página. Este método é executado sempre que uma nova instância de objeto de página for criada. Isso ocorre com o objetivo de garantir que o identificador único que o objeto de página possui é o mesmo presente na página JSF que está rodando no browser durante os testes. Caso a asserção entre os identificadores falhe, o teste também falhará. Os dois métodos seguintes serão invocados pelos casos de teste gerados. Esses métodos servem, respectivamente, para garantir que os elementos gerados pela ferramenta UID2JSF foram renderizados corretamente na tela, e que os textos de links definidos nos UIDs estão associados corretamente aos seus links. Por fim, o último método faz referência à navegação entre páginas. Para cada link presente na tela, haverá um método de transição para testar se a navegação feita pelo link está de acordo com o que foi definido no UID.

```
    @Override
    protected By getUniqueElement() {
        return By.id("IDOUO0");
    }

    public void assertPresence() {
        Assert.assertTrue(outLbl_CD3.isDisplayed());
        Assert.assertTrue(cLink_CD44.isDisplayed());
        Assert.assertTrue(outText_014.isDisplayed());
    }

    public void assertCLinksTexts() {
        Assert.assertEquals("comprar", cLink_CD44.getText());
    }

    public PinitialStatePageObject transition_From_P1PageObject_to_PinitialStatePageObject_cLink_CD44() {
        this.cLink_CD44.click();
        return PageFactory.initElements(driver, PinitialStatePageObject.class);
    }
}
```

**Figura 4.4 - Exemplo de métodos de teste em objeto de página**

## 4.4. Processo de criação de casos de teste

Assim como ocorre com os objetos de página e classes Managed Bean, será gerada uma classe de caso de teste para cada página JSF. Essa classe terá por responsabilidade efetuar a invocação de métodos presentes no objeto de página criado para a página JSF a ser testada.

A Figura 4.5 apresenta um exemplo de classe de caso de teste gerado automaticamente. Essa classe utiliza comandos da biblioteca JUnit (http://junit.org/) visando estruturar o processo de execução dos testes. Assim como na Figura 4.5, todas as classes de casos teste que forem geradas neste trabalho possuirão a seguinte estrutura:

- Método init() utilizando a anotação @BeforeClass – tem como funcionalidade a abertura do navegador antes de iniciar a execução geral dos testes. Neste trabalho os testes acontecerão utilizando apenas o browser Firefox.

- Método startPage() utilizando a anotação @Before – tem por funcionalidade fazer com que cada teste a ser executado comece a partir da página principal sendo testada. Este método é executado antes da execução de cada método de teste.

- Métodos com a anotação @Test – serão os métodos de teste que farão a invocação dos métodos definidos nos objetos de página.

- Método end() utilizando a anotação @AfterClass – tem por funcionalidade fechar o browser após a execução de todos os casos de teste.

```java
public class PinitialStateTestCase {

    private static WebDriver driver;
    @BeforeClass
    public static void init() {
        File ffpath = new File("C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe");
        FirefoxBinary ffbinary = new FirefoxBinary(ffpath);
        FirefoxProfile ffprofile = new FirefoxProfile();
        driver = new FirefoxDriver(ffbinary, ffprofile);
    }

    @Before
    public void startPage() {
        driver.get("http://localhost:8080/JavaServerFaces/faces/PinitialState.xhtml");
    }

    @Test
    public void testLinksTransitions0() {
        PinitialStatePageObject var1 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var1.transition_From_PinitialStatePageObject_to_PinitialStatePageObject_cLink_00();
    }

    @Test
    public void assertPresence() {
        PinitialStatePageObject var3 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var3.assertPresence();
    }

    @Test
    public void assertCLinksTexts() {
        PinitialStatePageObject var4 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var4.assertCLinksTexts();
    }

    @AfterClass
    public static void end() {
        driver.quit();
    }
```

**Figura 4.5 - Exemplo de classe de caso de teste**

## 4.5. Geração de Managed Beans com dados mockados

A ideia da criação de classes Managed Beans com dados mockados surgiu da verificação de que após uma nova transformação de UID feita pela ferramenta UID2JSF, era necessário criar manualmente a estrutura de Managed Beans e classes relacionadas para que fosse possível verificar os resultados do mapeamento feito. Além disso, para que a aplicação pudesse ser visualizada com dados preenchidos, era necessário fazer a conexão da aplicação com uma base dados, ou então programar lógicas nos Managed Beans para gerar conteúdo para a interface.

Para a geração automática das classes Managed Bean, foi utilizada uma lógica de armazenamento similar à apresentada na seção 4.3, utilizada no processo de armazenamento de

elementos JSF úteis a construção dos testes. A única diferença é que para os Managed Beans nenhum dos componentes mapeados deixou de ser armazenados na classe ProjectInfoHolder.

Para cada página JSF gerada foi também criada uma classe do tipo Managed Bean e outra Managed Bean Helper. A classe Managed Bean é responsável por atender as chamadas de métodos feitas pelas páginas JSF. Com isso, toda classe Managed Bean deve possuir métodos de acesso que forneçam ou recebam dados solicitados pelas páginas. A classe Helper será a responsável por fornecer dados fictícios quando uma classe Managed Bean for requisitada. Estes dados serão gerados no momento em que uma classe Managed Bean for criada e os valores gerados para cada elemento JSF serão iguais aos valores de ID definidos para cada componente JSF.

Com o auxílio da biblioteca CodeModel (https://codemodel.java.net/), responsável pela geração de classes Java programaticamente, serão geradas as classes Managed Bean e classes Managed Bean Helper. As classes Managed Bean possuirão métodos de acesso (getters / setters) e quando um desses métodos for invocado essa invocação será repassada para a classe helper, que ficará encarregada de prover e receber dados. Para a criação dos dados mockados (fictícios), cada atributo da classe helper receberá uma anotação disponibilizada pela biblioteca Podam (Pojo Data Mocker) (http://www.jemos.eu/projects/podam/), anotação essa que pode gerar diferentes tipos de dados permitidos pela linguagem Java.

As Figuras 4.6 e 4.7 apresentam, respectivamente, uma classe Managed Bean e sua classe Helper correspondente. Na Figura 4.7 estão apresentados três tipos de anotações fornecidas pela biblioteca Podam. São elas:

- @PodamCollection(nbrElements=5), que irá gerar 5 objetos do tipo Gravacao para o atributo valueGravacao.

- @PodamStringValue(), onde caso seja definido um valor para o campo *strValue*, esse será o valor que sempre será usado. Caso apenas o valor do campo *length* seja fornecido, então será gerado um valor do tipo String com o número de caracteres informado neste campo.

- @PodamBooleanValue, gera um valor do tipo booleano (true / false) para o atributo anotado.

```java
@ManagedBean
@SessionScoped
public class PinitialStateMB {

    private PinitialStateMBHelper mBeanHelper = new PinitialStateMBHelper();

    @PostConstruct
    protected void initMBHelper() {
        PodamFactoryImpl f = new PodamFactoryImpl();
        this.mBeanHelper = f.manufacturePojo(PinitialStateMBHelper.class, PinitialStateMBHelper.class);
    }

    public List<Gravacao> getValueGravacao() {
        return mBeanHelper.getValueGravacao();
    }

    public void setValueGravacao(List<Gravacao> valueGravacao) {
        mBeanHelper.setValueGravacao(valueGravacao);
    }

    public boolean isRenderTableValuedTable_Gravacao0() {
        return mBeanHelper.getRenderTableValuedTable_Gravacao0();
    }

    public String getValuenomeDaMusica() {
        return mBeanHelper.getValuenomeDaMusica();
    }
```

**Figura 4.6 - Exemplo de classe Managed Bean gerada.**

```java
public class PinitialStateMBHelper {

    @PodamCollection(nbrElements = 5)
    private List<Gravacao> valueGravacao;
    @PodamBooleanValue(boolValue = false)
    private boolean renderTableValuedTable_Gravacao0;
    @PodamStringValue(length = 0)
    private String valuenomeDaMusica;
    @PodamStringValue(strValue = "cabecalhoGravacao")
    private String cabecalhoGravacao;
    @PodamStringValue(strValue = "valuebuscar")
    private String valuebuscar;
    @PodamStringValue(strValue = "labelnomeDaMusica")
    private String labelnomeDaMusica;

    public void setValueGravacao(List<Gravacao> valueGravacao) {
        this.valueGravacao = valueGravacao;
    }

    public List<Gravacao> getValueGravacao() {
        return valueGravacao;
    }

    public void setRenderTableValuedTable_Gravacao0(boolean renderTableValuedTable_Gravacao0) {
        this.renderTableValuedTable_Gravacao0 = renderTableValuedTable_Gravacao0;
    }

    public boolean getRenderTableValuedTable_Gravacao0() {
        return renderTableValuedTable_Gravacao0;
    }

    public void setValuenomeDaMusica(String valuenomeDaMusica) {
        this.valuenomeDaMusica = valuenomeDaMusica;
    }

    public String getValuenomeDaMusica() {
        return valuenomeDaMusica;
    }
```

**Figura 4.7 - Exemplo de classe Managed Bean Helper gerada.**

# 5. Desenvolvimento da ferramenta

Objetivando acrescentar à ferramenta de UID2JSF a funcionalidade de iniciação de testes de interface antecipadamente, foi desenvolvida a versão 2.0 dessa ferramenta. Esta segunda versão fará a automatização do processo de geração de classes do tipo Managed Beans com dados mockados (fictícios) e também a geração de casos de testes básicos para cada página JSF mapeada. A ferramenta foi desenvolvida em linguagem Java com o auxílio do IDE (Integrated Development Environment) Eclipse.

Práticas Ágeis de desenvolvimento foram utilizadas no processo de desenvolvimento da ferramenta. Para auxiliar na organização do desenvolvimento foi utilizado o framework de comparação e análise de diversos métodos ágeis, denominado Framework de Práticas Ágeis (FAP do inglês Framework of Agile Practices) [SEKE 2007, FAGUNDES 2005]. O FAP engloba práticas de diversos métodos ágeis com o intuito de facilitar a definição de novos processos ágeis.

## 5.1. Definição do processo de desenvolvimento

O processo de desenvolvimento desse trabalho foi elaborado a partir da escolha de algumas práticas ágeis apresentadas pelo FAP, citado anteriormente. As práticas selecionadas serão apresentadas a seguir, agrupadas por atividades:

- Atividade de Definição dos Requisitos
    - Lista de Requisitos – Elaboração de um documento contendo todos os requisitos do sistema.
- Atividade de Projeto da Arquitetura do Sistema
    - Projeto Geral do Sistema – A partir dos requisitos conhecidos até o momento é elaborado um diagrama de pacotes do sistema representando a arquitetura lógica do sistema.
- Atividade de atribuição dos requisitos às iterações

- Planejamento das Iterações – No início de cada iteração deve ser feito um planejamento para definir quais requisitos serão implementados.
- Duração das Iterações – A partir dos requisitos que devem ser implementados em cada iteração, é definida sua duração. Neste trabalho as duas primeiras iterações tiveram a duração de três semanas. As duas últimas iterações duraram duas semanas.

- Atividades de Desenvolvimento do Incremento do Sistema
  - Implementação dos Requisitos durante cada Iteração - Consiste na geração de código para os requisitos pertencentes à iteração corrente.
  - Integração Paralela ao Desenvolvimento - Integração do código gerado na iteração corrente com os das iterações passadas.

## 5.2. Desenvolvimento

A ferramenta de mapeamento de UID para JSF desenvolvida por [DAMIANI, 2011], tem por objetivo receber um arquivo XML (Extensible Markup Language) que representa um UID, processar esse arquivo e gerar as interfaces de acordo com regras de mapeamento definidas. Para a representação dos UIDs em XML, existe uma especificação de uma DTD (Document Type Definition). Esta especificação foi desenvolvida por [VILAIN, 2003], visando facilitar o armazenamento e também intercâmbio das instâncias dos UIDs entre aplicações.

Durante o processo de mapeamento de um UID para elementos JSF, serão capturadas informações que serão úteis na geração automática de Páginas de Objetos, casos de testes e classes Managed Beans.

### 5.2.1. Lista de Requisitos

1. Refatorar código do projeto UID2JSF para suportar as novas regras de implementação.
2. Componentes JSF devem possuir IDs únicos definidos.
3. Cada página JSF deve possuir um campo oculto único que a identifique.

4. Geração de IDs completos para componentes, para que estes possam ser localizados pela ferramenta Selenium quando necessário.
5. Adicionar atributo *styleClass* a todos os elementos JSF, para que esta seja a forma padrão de localização de elementos pelo Selenium.
6. Implementar lógica para armazenamento de dados gerados na transformação de um UID em JSF para que estes dados possam ser usados na construção das classes de teste do projeto.
7. Criação de objeto de página para cada página gerada.
8. Incluir nas classes de objeto de página as informações dos componentes mapeados.
9. Geração de métodos nos objetos de página que serão utilizados pelos casos de teste para validação de regras.
10. Geração de método que retorne o identificador único do objeto de página.
11. Geração de caso de teste que faça a validação se os componentes mapeados foram renderizados na tela.
12. Geração de caso de teste que faça a asserção de nomes de elementos JSF do tipo CommandLink que tenham sido pré-definidos nos UID.
13. Geração de caso de teste que valide a navegação feita após o clique de um componente do tipo CommandLink.
14. Implementar lógica para armazenamento de dados gerados na transformação de um UID em JSF para que estes dados possam ser usados na construção das classes Managed Bean e classes Manged Bean Helper.
15. Criar classe do tipo Managed Bean para cada página JSF gerada pela ferramenta UID2JSF.
16. Criar classe auxiliar para cada classe Managed Bean para fornecer dados fictícios.

17. Criar lógica para que os dados fictícios gerados tenham relação com o tipo de informação cujo componente JSF foi criado para receber.

### 5.2.2. Arquitetura do projeto

Para que o desenvolvimento desse trabalho pudesse ser realizado de forma satisfatória, o antigo projeto UID2JSF foi refatorado e passou a possuir um novo pacote de classes denominado "tests", que ficou responsável por armazenar os arquivos com formato Java responsáveis pelas regras de negócio para a geração de testes de interfaces web. Além das classes geradas, foram também adicionadas ao projeto algumas APIs externas que deram suporte ao desenvolvimento. Com isso, a Figura 5.1, tem por objetivo representar, de forma a gráfica, a estruturação dos pacotes da ferramenta desenvolvida.

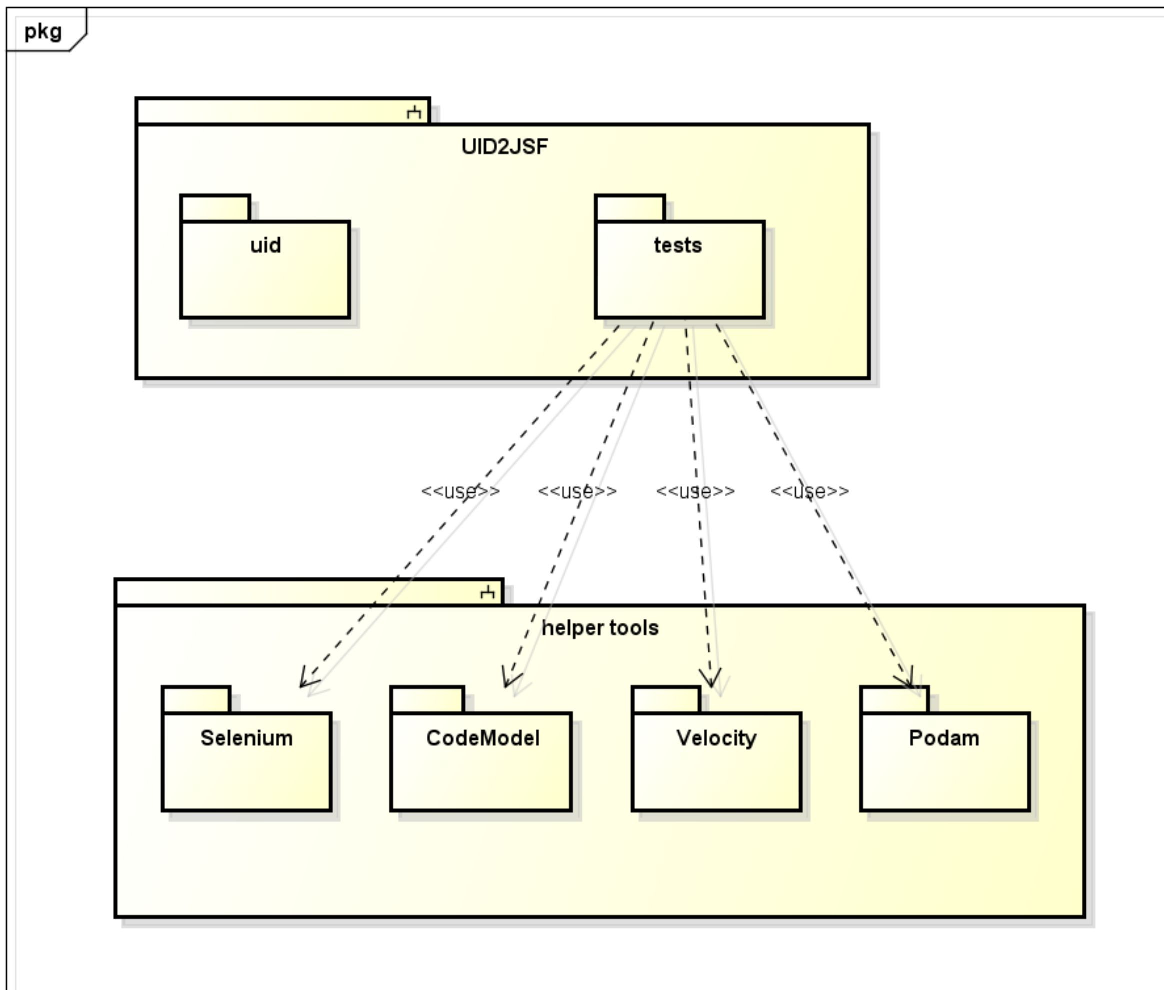



**Figura 5.1 - Diagrama de pacotes para a nova arquitetura do sistema.**

Para um melhor entendimento e representação do que foi realizado neste trabalho, foram gerados dois diagramas de atividades que representam de uma forma macro, as atividades realizadas pela ferramenta UID2JSF antes e depois do desenvolvimento das novas funcionalidades. A Figura 5.2 apresenta as atividades realizadas pela aplicação antes do início deste desenvolvimento, onde, para um UID informado, este era processado e os dados eram mapeados para páginas JSF com extensão *xhtml.*

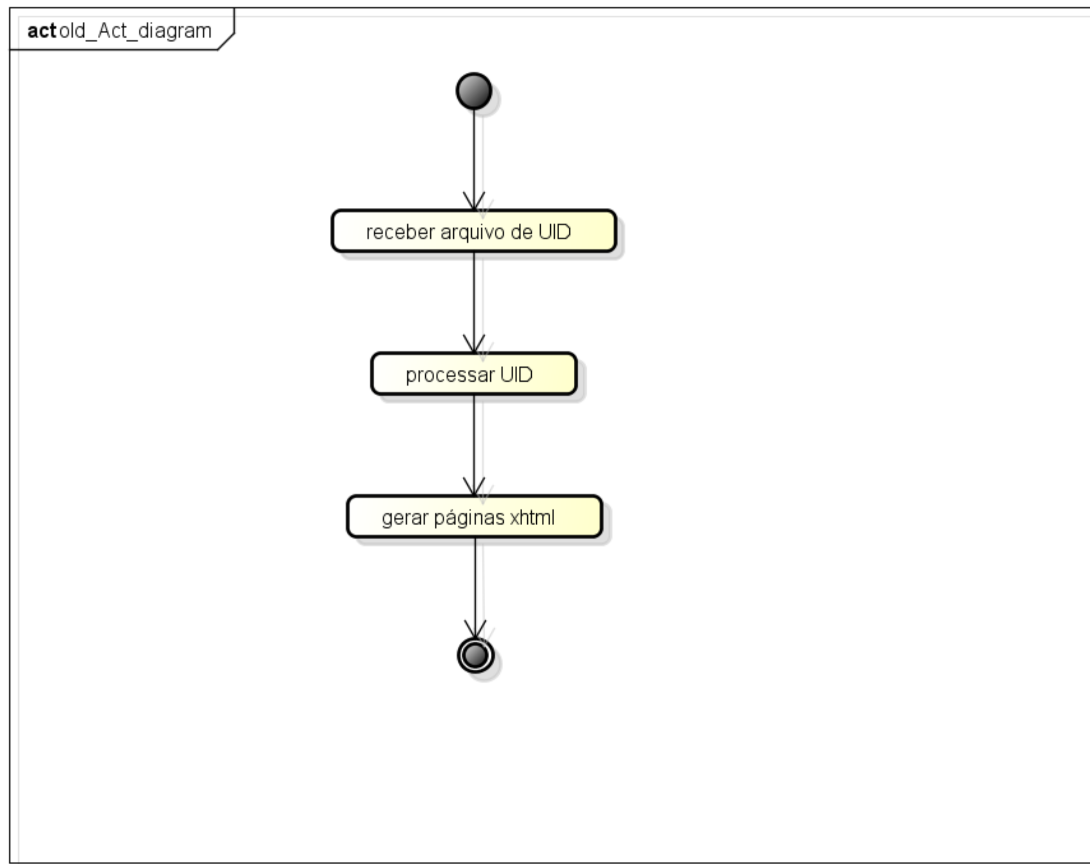



**Figura 5.2 - Antigo diagrama de atividades a aplicação UID2JSF**

Na sequência, a Figura 5.3 apresenta as novas atividades realizadas pela aplicação, onde além da geração das páginas JSF, também é realizada a criação de um objeto de página para cada página JSF, uma classe com casos de teste referente à página criada e uma classe Managed Bean responsável por prover a troca de dados entre as interfaces e o servidor de aplicação.

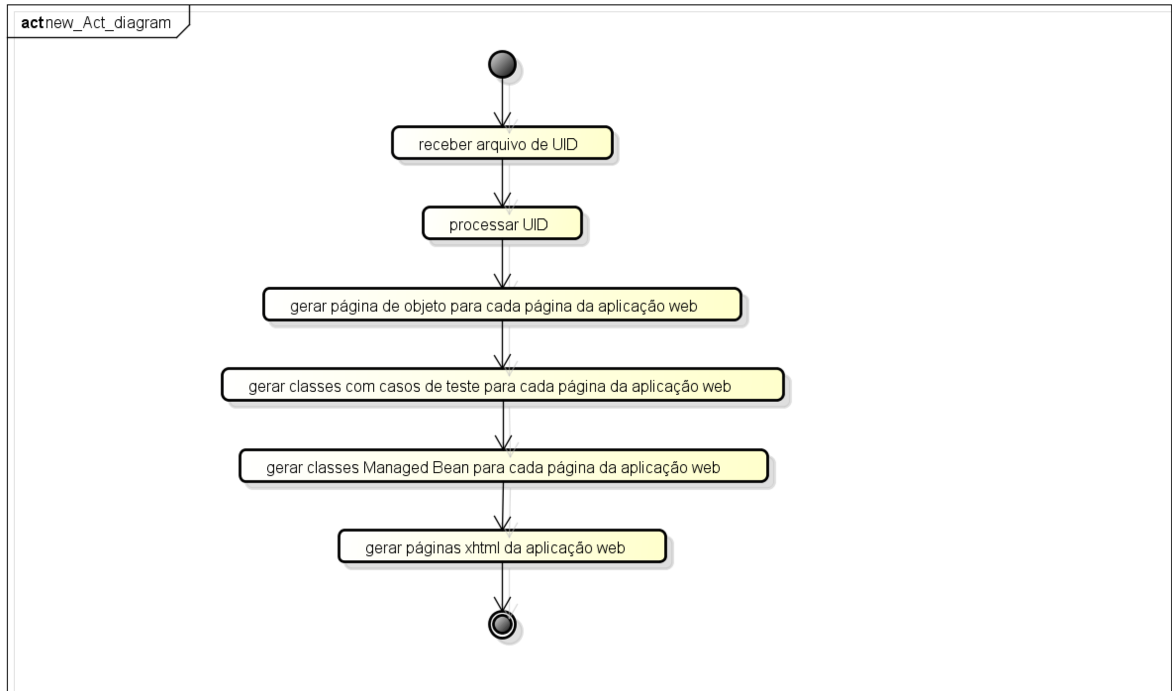



**Figura 5.3 - atual diagrama de atividades para a aplicação UID2JSF**

## 5.3. Iterações

Inicialmente o projeto teria três iterações principais. Sendo a primeira responsável por estruturar o projeto de uma forma em que fosse possível coletar os componentes gerados pela ferramenta UID2JSF e em um processo final essas informações pudessem ser utilizadas para a montagem das páginas de teste e os casos de teste. Na segunda iteração seriam criadas as lógicas de geração automática de PageObjects (objetos de página) para cada uma das páginas JSF e fazer com que esses objetos de página contivessem informações necessárias, como por exemplo: componentes referenciando elementos de página JSF e métodos de serviço referente à pagina a ser testada.. Por fim, a terceira iteração englobaria o processo de criação de uma estrutura de projetos que auxiliasse na geração de casos de teste automatizados, partindo dos dados extraídos durante o processo de mapeamento de UID2JSF (primeira iteração). Porém, ao chegar à terceira iteração, notou-se a possibilidade de facilitar a usabilidade do projeto gerado pela ferramenta UID2JSF 2.0. Isso porque, toda vez que um mapeamento de UID para JSF era feito, o usuário precisava criar manualmente as classes Managed Beans e suas relacionadas para que pudesse iniciar o servidor da aplicação e testá-las, o que dificultava a rápida apresentação dos resultados. Foi a partir dessa dificuldade que foi decidido incluir uma quarta iteração ao projeto, que ficaria

responsável por gerar classes do tipo Managed Bean de uma forma automatizada, onde essas classes e suas relacionadas conteriam dados mockados (fictícios). Com isso o usuário do projeto gerado teria apenas o trabalho de iniciar o servidor web para onde as classes geradas foram exportadas e na sequência poderia executar as classes de casos de testes desejadas.

### 5.3.1. Primeira iteração

Os requisitos selecionados para a primeira iteração foram aqueles relacionados com o processo de adequação da ferramenta UID2JSF para que essa suportasse a geração de testes de interface. Os requisitos selecionados estão listados a seguir:

1. Refatorar código do projeto UID2JSF para suportar as novas regras de implementação.

2. Componentes JSF devem possuir IDs únicos definidos.

3. Cada página JSF deve possuir um campo oculto único que a identifique.

4. Geração de IDs completos para componentes para que estes possam ser localizados pela ferramenta Selenium quando necessário.

5. Adicionar atributo *styleClass* a todos os elementos JSF para que seja a forma padrão de localização de elementos pelo Selenium.

O processo inicial de reestruturação do projeto foi, sem dúvidas, a etapa mais custosa de todo o desenvolvimento realizado. Isso ocorreu porque a ferramenta UID2JSF tinha como única função receber um arquivo de UID no formato “XML”, processá-lo e extrair informações necessárias para gerar páginas JSF com extensão “XHTML”. A Figura 5.4 apresenta o diagrama de sequência que representa parte do processo realizado pela ferramenta UID2JSF antes do desenvolvimento deste trabalho. O diagrama mostra que quando o nodo raiz, componente da classe “Página”, retornar sua string, esta será a página JSF que representa toda a árvore de componentes e após esta etapa, basta apenas salvar esta página em um arquivo com extensão “XHTML” [DAMIANI, 2011].

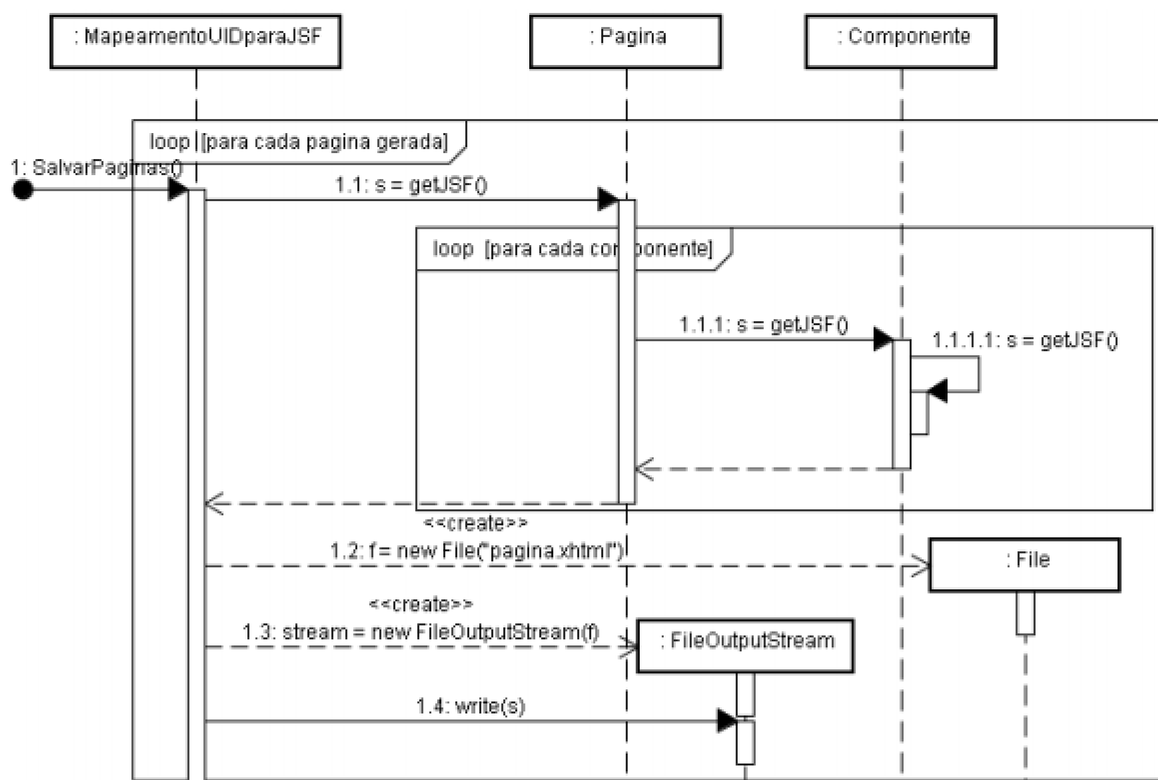



**Figura 5.4 - Diagrama de sequência para criação dos arquivos das páginas [Damiani, 2011]**

A primeira etapa do desenvolvimento deste trabalho ficou em torno da reestruturação do projeto UID2JSF, onde a antiga classe principal, chamada de MapeamentoUIDParaJSF passou a se chamar *MapeamentoUID2Project.* Com isso, a ferramenta além de gerar uma página JSF para cada classe de Página mapeada, também irá gerar uma classe do tipo objeto de página, uma classe de caso de teste, uma classe Managed Bean e uma classe Helper para cada classe Managed Bean. A Figura 5.5 apresenta o diagrama de sequência que representa esse novo fluxo da aplicação.

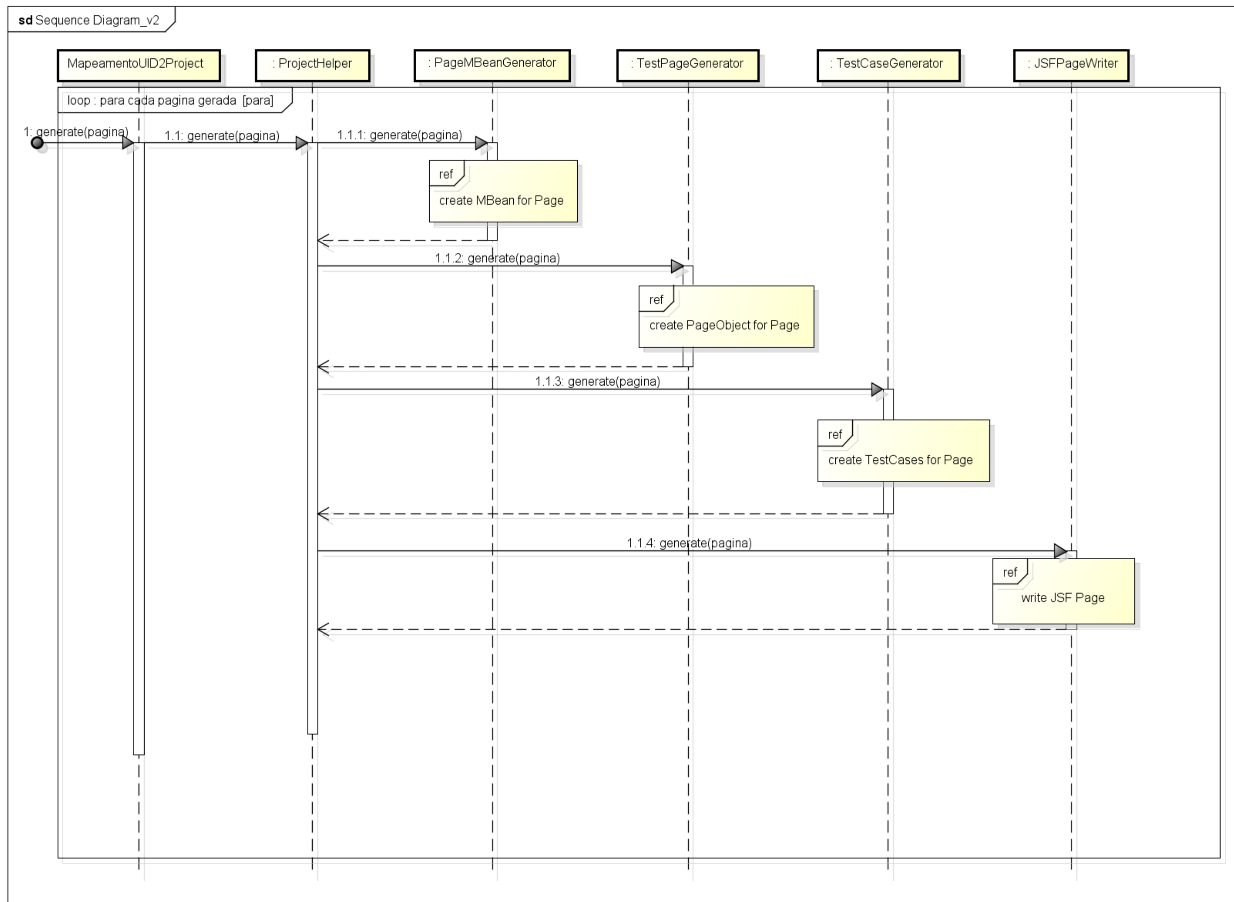



**Figura 5.5 - Diagrama de sequência para representar de uma forma macro o processo de geração das classes do projeto UID2JSF 2.0.**

As classes já existentes no projeto UID2JSF, responsáveis por transformar dados oriundos de um UID em componentes JSF, passaram também a definir os atributos ID e styleClass para esses componentes. Esta atividade auxilia na localização dos elementos de uma página pelo Selenium (framework apresentado anteriormente) no momento da execução dos casos de teste. Para definição dos atributos ID e styleClass, foi criada uma classe responsável por gerar IDs únicos utilizando a seguinte regra de concatenação:

- Prefixo do componente + "_" + ID vindo do UID + "_" + número incremental por tipo de elemento;

Onde o prefixo do componente seria uma das opções listadas na Tabela 4.1. Um exemplo de ID gerado seria: ***cLink_Gravacao1_***, onde, cLink é o prefixo, Gravação1 é o ID vindo do UID e 1 é o número incremental inicial. A mesma ideia foi utilizada para poder gerar o caminho completo de

um ID dentro de um Naming Container pai. A única diferença é que para gerar o ID completo de um elemento, o ID de um elemento pai foi concatenado ao ID de um elemento filho usando ":" como separador.

A tarefa de adicionar um ID único para cada página gerada foi implementada usando uma string randômica de seis dígitos com letras apenas e colocando esse ID em um componente do tipo *inputHidden* que não será renderizado na tela.

### 5.3.2. Segunda iteração

Os requisitos selecionados para a segunda iteração foram os requisitos responsáveis pelo armazenamento dos componentes JSF gerados a partir da transformação de um UID para JSF, e em seguida a utilização desses componentes para a geração das classes de objeto de página:

6. Implementar lógica para armazenamento de dados gerados na transformação de um UID em JSF para que estes dados possam ser usados na construção das classes de teste do projeto.

7. Criação de Objeto de página para cada página gerada

8. Incluir nas classes de objeto de página as informações dos componentes mapeados (atributos).

9. Geração de método que retorne o identificador único da Pagina de Objeto.

10. Geração de métodos nos objetos de página que serão utilizados pelos casos de teste para validação de regras.

A segunda iteração consistiu em implementar a forma de armazenamento de dados a serem utilizados durante o processo de geração de testes e trabalhar com essas informações para gerar as classes objeto de página para cada uma das páginas JSF mapeadas.

Para poder definir quais componentes seriam guardados para utilização futura, foi implementado uma arquitetura onde para cada componente HTML criado, pudesse ser definido se este componente estaria ou não presente nos testes e, em caso afirmativo, ao ser criado, ele seria registrado em um setor do código responsável por armazenar os componentes HTML que foram mapeados.

Para definir se um componente JSF fará parte dos testes, foi definida a seguinte regra:

- Todos os componentes que no fim do construtor de sua classe, tenham a chamada de método *bindTest (Componente comp),* serão registrados em uma classe estática chamada ProjectInfoHolder, que guarda informações do projeto ao longo de sua execução e permite que outras classes possam obter essas informações para executar tarefas futuras.

Já os processos de geração de objetos de página com seus atributos de classe, construtor de classe e métodos de teste, foram feitos utilizando o framework Velocity (http://velocity.apache.org/), que a partir de um arquivo de modelo definido e da definição de objetos para serem processados por esse modelo, constrói arquivos utilizando a característica de reflexão da linguagem Java. A Figura 5.6 mostra o arquivo de modelo criado para gerar as classes de objeto de página. Nesse modelo, foram usadas duas características de processamento da ferramenta Velocity. São elas:

- O que foi declarado após o caracter **'$'**, indica um valor que será fornecido e então processado pelo Velocity. O valor pode ser um objeto qualquer da linguagem Java.

- As palavras precedidas pelo caráter especial # indicam que esta é uma palavra reservada da linguagem. Ex: #foreach indica que será executado um loop sobre uma variável definida, no exemplo as variáveis foram *'list'* e *'assertMethods'*. Já a definição #end indica o fim de alguma execução, nesse caso indica o fim do loop.

```
testtemplate.vm
1 package pageobject.gen;
2
3 import java.util.List;
4 import junit.framework.Assert;
5 import org.openqa.selenium.By;
6 import org.openqa.selenium.WebDriver;
7 import org.openqa.selenium.WebElement;
8 import org.openqa.selenium.support.FindBy;
9 import pageobject.AbstractPageObject;
10 import org.openqa.selenium.support.PageFactory;
11 public class ${clazzName}  extends AbstractPageObject {
12 #set( $myobject =$varMap)
13 #foreach ($mapEntry in $myobject.entrySet())
14     //$mapEntry.key
15     #set( $list =$mapEntry.value)
16     #foreach ($obj in $list)
17         @FindBy($obj.getFindBy())
18         private WebElement $obj.getID() ;
19     #end
20 #end
21 /**
22 * CONSTRUCTOR
23 **/
24 public ${clazzName}(WebDriver driver) {
25 super(driver);
26 }
27 @Override
28 protected By getUniqueElement() {
29 return By.id("${uniqueElement}");
30 }
31 /**
32 * ASSERTIONS - TEST METHODS
33 **/
34 #foreach ($method in $assertMethods)
35     public void ${method.getMethodName()}(){
36     $method.toString()
```

**Figura 5.6 - Exemplo de modelo utilizado pelo Velocity.**

Após ter sido feito o modelo da classe, o passo seguinte foi processar as informações de componentes guardadas na classe ProjectInfoHolder e passá-las para o modelo para que esse pudesse processar os objetos e gerar as classes de objeto de página. A Figura 5.7 mostra um exemplo de uma página de objeto gerada como resultado das implementações feitas nesta iteração.

#### 5.3.2.1. Localização de elementos

Logo no início da classe estão definidos três atributos que fazem referência a elementos da página HTML associada a esta classe. Cada elemento possui uma anotação do tipo "@FindBy" que indica a forma como esse elemento deve ser encontrado na página HTML. Como mostrado na Figura 5.7, o padrão de localização dos elementos é através do atributo "*className*", que faz referência ao atributo "styleClass" do HTML. Os atributos serão buscados e inicializados no momento em que a classe PageFactory invocar seu método "initElements" passando como

argumentos o controlador do browser (driver) e a o nome de uma classe. A invocação inicial desse método será feita no momento de execução dos casos de testes.

#### 5.3.2.2. Geração de métodos

A cada invocação do método *"initElements"*, ao tentar retornar uma instância de um objeto de página, será verificado se o identificador único da página JSF corresponde ao identificar único do Objeto de página. Garantindo assim que o objeto de página estará executando suas ações sobre a página correta. Como pode ser visto na Figura 5.7, o identificador do objeto de página é obtido através do método "*getUniqueElement".*

Os três métodos seguintes, serão invocados a partir das classes de caso de teste desenvolvidas na terceira iteração. Os dois primeiros tem o intuito de validar a consistência das informações presentes na interface do usuário. Com o auxílio do framework JUnit, foram feitos dois tipos de método de asserção. No primeiro método são utilizadas as referências dos elementos da página JSF para testar se os elementos estão ou não renderizados na tela. A resposta desse método é processada pela classe Assert do JUnit e se o que retornar não estiver correto o teste falhará. O segundo método de asserção verifica se o texto definido para os links da página estão de acordo com texto fornecido no diagrama de interação com o usuário (UID).

Por fim, o último tipo de método é responsável por testar as navegações possíveis de serem feitas a partir da página testada. Um método de transição é criado para cada link da página e esses métodos funcionam da seguinte forma:

1. O elemento que faz referência ao componente do JSF simula um clique.
2. O método *"initElements"* da classe PageFactory é chamado passando como parâmetros o driver do browser e a página para onde este link deveria redirecionar a aplicação.
3. É feita a validação de igualdade entre o elemento único do objeto de página criado no passo 2 e o elemento único da página JSF. Caso a validação falhe, o teste falhará indicando que a navegação não está correta.

```java
13 public class P1PageObject extends AbstractPageObject {
14     // COMMANDLINK
15     @FindBy(className = "cLink_CD2_2")
16     private WebElement cLink_CD2_2;
17     // OUTPUTTEXT
18     @FindBy(className = "outText_dTable_CD_1_Header_10")
19     private WebElement outText_dTable_CD_1_Header_10;
20
21     /**
22      * CONSTRUCTOR
23      **/
24     public P1PageObject(WebDriver driver) {
25         super(driver);
26     }
27
28     @Override
29     protected By getUniqueElement() {
30         return By.id("SNpNwj");
31     }
32     /**
33      * ASSERTIONS - TEST METHODS
34      **/
35     public void assertPresence() {
36         Assert.assertTrue(cLink_CD2_2.isDisplayed());
37         Assert.assertTrue(outText_dTable_CD_1_Header_10.isDisplayed());
38     }
39
40     public void assertCLinksTexts() {
41         Assert.assertEquals("mostrarCD", cLink_CD2_2.getText());
42     }
43
44     /**
45      * TRANSACTIONS - TEST METHODS
46      **/
47     public PinitialStatePageObject transition_From_P1PageObject_to_PinitialStatePageObject_cLink_CD2_2() {
48         this.cLink_CD2_2.click();
49         return PageFactory.initElements(driver, PinitialStatePageObject.class);
50     }
```

**Figura 5.7 - Exemplo de objeto de página.**

### 5.3.3. Terceira iteração

Os requisitos selecionados para a terceira iteração foram:

11. Geração de caso de teste que faça a validação se os componentes mapeados foram renderizados na tela.

12. Geração de caso de teste que faça a asserção de nomes de elementos JSF do tipo CommandLink que tenham sido pré-definidos nos UID.

13. Geração de caso de teste que valide a navegação feita após o clique de um componente do tipo CommandLink.

O objetivo principal da terceira iteração era a criação de uma estrutura de código que permitisse que a partir dos dados extraídos na primeira iteração e dos objetos de página criados

na segunda iteração, fosse possível automatizar o processo de geração de casos de testes que auxiliariam o usuário final da aplicação no início do processo de testes de sua aplicação.

Para a criação das classes de caso de teste, foi utilizada a biblioteca CodeModel (https://codemodel.java.net/) que vem inclusa no JDK (Java Development Kit) do Java. Essa biblioteca disponibiliza classes com métodos que permitem a criação de classes Java de uma forma programática.

Como explicado na seção 4.4 deste trabalho, as classes de caso de teste devem possuir os métodos *init* anotado com @BeforeClass, *startPage* anotado com @Before, os métodos de teste a serem executados anotados com @Test, e o método *end* anotado com @AfterClass. Todas as classes de caso de teste geradas foram implementadas para conter os métodos *init* e *end* implementados da mesma forma, isso porque esses métodos, respectivamente, são responsáveis por abrir e fechar o browser do usuário. O que irá mudar de uma classe para a outra são os métodos startPage e os métodos de testes a serem executados. O método startPage irá ser diferente de uma classe de teste para outra, pois ele é o responsável por abrir a página inicial dos testes antes da execução de cada método anotado com @Test. A Figura 5.8 mostra um exemplo do método startPage indicando que a URL "*http://localhost:8080/JavaServerFaces/faces/PinitialState.xhtml*" será acessada antes da execução de cada método anotado com @Test. Na Figura 5.8 também pode ser visto os métodos anotados com @Test, que serão invocados quando a classe for executada. Esses métodos farão a chamada de métodos da classe de objeto de página correspondente à classe de teste sendo executada.

```java
public class PinitialStateTestCase {

    private static WebDriver driver;
    @BeforeClass
    public static void init() {
        File ffpath = new File("C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe");
        FirefoxBinary ffbinary = new FirefoxBinary(ffpath);
        FirefoxProfile ffprofile = new FirefoxProfile();
        driver = new FirefoxDriver(ffbinary, ffprofile);
    }

    @Before
    public void startPage() {
        driver.get("http://localhost:8080/JavaServerFaces/faces/PinitialState.xhtml");
    }

    @Test
    public void testLinksTransitions0() {
        PinitialStatePageObject var1 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var1.transition_From_PinitialStatePageObject_to_PinitialStatePageObject_cLink_00();
    }

    @Test
    public void assertPresence() {
        PinitialStatePageObject var3 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var3.assertPresence();
    }

    @Test
    public void assertCLinksTexts() {
        PinitialStatePageObject var4 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var4.assertCLinksTexts();
    }

    @AfterClass
    public static void end() {
        driver.quit();
    }
```

**Figura 5.8 - Exemplo de classe de teste.**

Por fim, para que o usuário teste o funcionamento de um caso de teste, basta que ele deve seguir os seguintes passos:

1. Selecionar a classe a ser testada.
2. Ir até o menu Run do eclipse (Figura 5.9)
3. Escolher a opção Run As > JUnit Test.

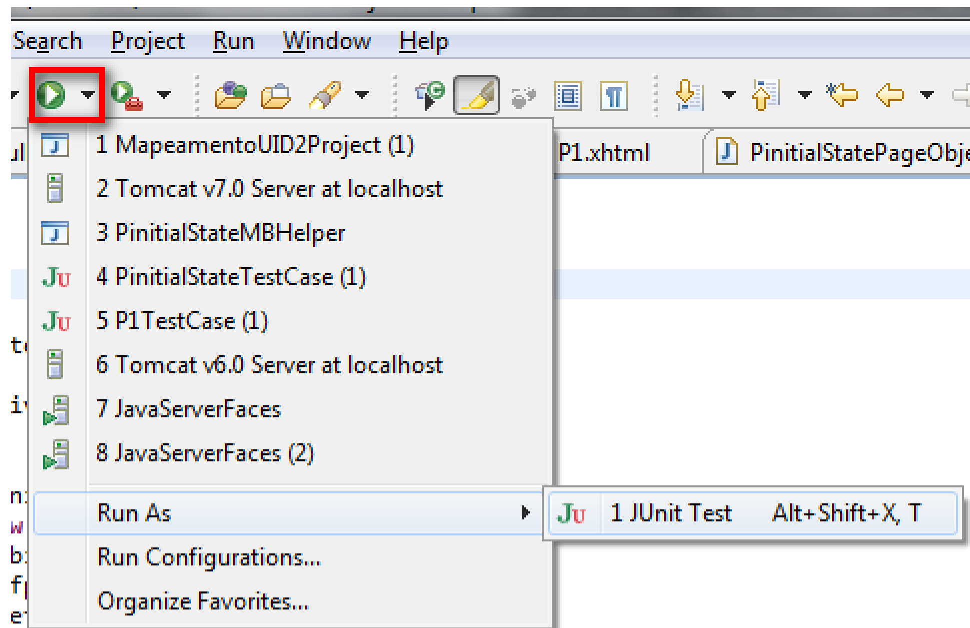



**Figura 5.9 - Exemplo de execução de classe de teste.**

### 5.3.4. Quarta iteração

A última iteração consistiu em desenvolver os requisitos faltantes, sendo estes os novos requisitos definidos durante a terceira iteração. Portanto, os requisitos selecionados foram:

14. Implementar lógica para armazenamento de dados gerados na transformação de um UID em JSF para que estes dados possam ser usados na construção das classes Managed Bean e classes Manged Bean Helper.

15. Criar classe do tipo Managed Bean para cada página JSF gerada pela ferramenta UID2JSF.

16. Criar classe auxiliar para cada classe Managed Bean para fornecer dados fictícios.

17. Criar lógica para que os dados fictícios tenham relação com o tipo de informação cujo componente JSF foi criado para receber.

#### 5.3.4.1. Extração de informações

O processo de extração de informações úteis para criar as classes Managed Bean e classes Helper foi feita da mesma maneira realizada para extrair informações para geração das classes de testes. Todas as classes de componentes JSF receberam no fim de seus construtores a chamada do

método bindMBean(Component), fazendo com que esses componentes fossem adicionados à classe ProjectInfoHolder, responsável por guardar informações enquanto a aplicação está executando.

Para obter as informações de atributos que cada classe Managed Bean deveria ter como conteúdo, foi criado um método que percorre a informação JSF de cada componente e através de uma expressão regular encontra os textos com padrão "#{Bean.*valor*}", com isso é possível extrair a constante valor, que represente o nome do atributo na classe Managed Bean.

#### 5.3.4.2. Geração de classes

Após a extração dos atributos foi utilizada a biblioteca CodeModel para a geração das classes Managed Bean e classes Managed Bean Helper. Para cada atributo pertencente a um Managed Bean, foram criados métodos de acesso (getters / setters) a essas atributos. Quando um desses métodos for invocado, essa invocação será repassada para a classe helper que ficará encarregada de prover e receber dados. A declaração dos atributos de um Managed Bean acontecerá apenas na classe. No momento da criação desta classe helper, cada atributo receberá uma anotação fornecida pela biblioteca Podam (Pojo Data Mocker) (http://www.jemos.eu/projects/podam/), responsável por gerar dados aleatórios de acordo com o tipo da anotação fornecida. As Figuras 5.10 e 5.11 apresentam, respectivamente, as classes Managed Bean e sua classe helper gerada.

```
@ManagedBean
@SessionScoped
public class PinitialStateMB {

    private PinitialStateMBHelper mBeanHelper = new PinitialStateMBHelper();

    @PostConstruct
    protected void initMBHelper() {
        PodamFactoryImpl f = new PodamFactoryImpl();
        this.mBeanHelper = f.manufacturePojo(PinitialStateMBHelper.class, PinitialStateMBHelper.class);
    }

    public List<Gravacao> getValueGravacao() {
        return mBeanHelper.getValueGravacao();
    }

    public void setValueGravacao(List<Gravacao> valueGravacao) {
        mBeanHelper.setValueGravacao(valueGravacao);
    }

    public boolean isRenderTableValuedTable_Gravacao0() {
        return mBeanHelper.getRenderTableValuedTable_Gravacao0();
    }

    public String getValuenomeDaMusica() {
        return mBeanHelper.getValuenomeDaMusica();
    }
```

**Figura 5.10 - Exemplo de classe Managed Bean gerada.**

```java
public class PinitialStateMBHelper {

    @PodamCollection(nbrElements = 5)
    private List<Gravacao> valueGravacao;
    @PodamBooleanValue(boolValue = false)
    private boolean renderTableValuedTable_Gravacao0;
    @PodamStringValue(length = 0)
    private String valuenomeDaMusica;
    @PodamStringValue(strValue = "cabecalhoGravacao")
    private String cabecalhoGravacao;
    @PodamStringValue(strValue = "valuebuscar")
    private String valuebuscar;
    @PodamStringValue(strValue = "labelnomeDaMusica")
    private String labelnomeDaMusica;

    public void setValueGravacao(List<Gravacao> valueGravacao) {
        this.valueGravacao = valueGravacao;
    }

    public List<Gravacao> getValueGravacao() {
        return valueGravacao;
    }

    public void setRenderTableValuedTable_Gravacao0(boolean renderTableValuedTable_Gravacao0) {
        this.renderTableValuedTable_Gravacao0 = renderTableValuedTable_Gravacao0;
    }

    public boolean getRenderTableValuedTable_Gravacao0() {
        return renderTableValuedTable_Gravacao0;
    }

    public void setValuenomeDaMusica(String valuenomeDaMusica) {
        this.valuenomeDaMusica = valuenomeDaMusica;
    }

    public String getValuenomeDaMusica() {
        return valuenomeDaMusica;
    }
```

**Figura 5.11 - Exemplo de classe auxiliar para classe Managed Bean**

## 5.4. Exemplos de testes gerados a partir do mapeamento de UIDs para JSF

Esta seção visa apresentar de forma prática a aplicação da ferramenta UID2JSF 2.0. Para a exemplificação de uso da ferramenta, foi escolhido o UID *Mostrar gravações de um CD.* As imagens e explicações da geração dos testes para o UID proposto podem ser encontradas no Apêndice A deste trabalho.

# 6. Considerações Finais

Este trabalho apresentou uma proposta de geração de testes para interfaces JSF (Java Server Faces) geradas a partir do mapeamento de UIDs (User Interaction Diagrams). Tomando como base o processo de mapeamento de UIDs para componentes JSF, feito pela ferramenta UID2JSF, foram definidas estratégias de armazenamento das informações de cada componente mapeado e posteriormente essas informações foram utilizadas para construir classes Managed Bean e classes de teste. As classes de teste geradas ficaram responsáveis por testar funcionalidades pré-determinadas e específicas de cada página JSF gerada pela ferramenta de UID2JSF.

Também foi desenvolvido um protótipo de uma ferramenta para automatizar o processo de geração dos testes de interface. O protótipo desenvolvido obteve um bom desempenho quanto à automatização da geração das classes e testes para cada página JSF mapeada a partir de um UID. A ferramenta ajudou a antecipar a geração dos testes da aplicação, gerando testes de interface já no momento da validação de requisitos da aplicação.

Os testes gerados permitem a execução de ações que um usuário real da aplicação poderia executar (e.g. navegações através de links), além disso, como uma forma de prototipação de testes, foram gerados também, testes de asserção básicos que podem ser modificados e evoluídos pelo programador que venha a cuidar dos testes da aplicação. O protótipo desenvolvido também eliminou um ponto negativo da aplicação UID2JSF que fazia com que o usuário final tivesse que implementar as classes Managed Beans de controle das páginas JSF para que pudesse rodar a aplicação final em um servidor web. Para isso foi automatizado o processo de geração de classes Managed Beans com dados mockados (fictícios).

A maior contribuição deste trabalho é o auxílio na etapa de projeto da interface com o usuário e teste dessas interfaces. A partir da especificação de UIDs, o projetista pode tomar como base as páginas geradas, rodar a aplicação em modo desenvolvimento e executar os casos de teste gerados para garantir a integridade parcial da aplicação. Mesmo que um desenvolvedor venha a fazer alterações nas páginas JSF geradas a partir do mapeamento de UIDs, os testes continuarão sendo válidos para a aplicação. Esses testes gerados deixarão de ser válidos caso o desenvolvedor remova algum elemento das páginas JSF ou altere os identificadores de localização de componente utilizados pela ferramenta Selenium no momento de execução dos

testes. Neste caso o desenvolvedor pode utilizar os testes gerados e adaptá-los para que voltem a funcionar com as mudanças feitas nas páginas JSF.

A Tabela 7.1 apresenta um comparativo entre a ferramenta desenvolvida neste trabalho com as apresentadas no capítulo de trabalhos relacionados. A comparação foi realizada considerando os seguintes critérios: A origem dos dados para geração dos testes; Para que tipo de interface os testes são gerados; o processo de geração da interface é automático; se a interface é gerada a partir de um modelo abstrato; se juntos com os testes é gerada um protótipo de aplicação onde os testes podem ser aplicados; e se um humano deve participar do processo de geração dos testes para informar dados a serem usados nos testes para validação.

**Tabela 7.1 – Tabela de comparação entre ferramentas.**

| | **Reverse Engineering of GUI Models for Testing** | **User Interface Testing and its Challenges in an Industrial Scenario** | **Model Based Testing Of Web Applications** | **Ferramenta Para Testes de Interface Geradas a Partir de UIDs** |
|---|---|---|---|---|
| **Origem dos Testes** | Modelos gerados através de engenharia reversa | Diagrama de Sequência | Máquina de Estados Finitos | UIDs |
| **Tipo de Interface** | Desktop | Desktop | Web | Web |
| **Testes gerados junto com protótipo da aplicação** | Não | Não | Não | Sim |
| **Dados de teste informados manualmente** | Sim | Não Informado | Sim | Não |

Para trabalhos futuros sugerimos melhorias na ferramenta implementada, incluindo:

- A implementação de regras para a geração de novos casos de teste para as interfaces.
- A geração de um Suite (grupo) de casos teste que possa, ao ser executado, executar todos os casos de teste da aplicação de uma só vez.
- A implementação de regras que permitam que os testes rodem em uma velocidade (segundos) pré-determinada pelo testador. Onde cada ação de um caso de teste terá um intervalo entre suas execuções para que as ações possam ser acompanhadas visualmente.
- Criar uma estrutura de geração de testes tal que a cada novo mapeamento de UID para JSF, as informações de teste que já tenham sido geradas não sejam sobrescritas ou então que o testador possa definir quais informações podem ser substituídas. Esta regra se aplicaria nos casos onde um UID já tenha sido mapeado e deseja-se refazer o processo de mapeamento, porém, sem que os testes já gerados sejam alterados.

# Referências Bibliográficas


ACHKAR, H. **Model Based Testing of Web Applications.** - 2010.

AGARWAL, B. B.; TAYAL, S. P. e GUPTA, M. **Software Engineering and Testing: An Introduction (Computer Science)** [Livro]. - [s.l.] : Jones and Bartlett Publishers, 2010.

BERNARDO, P. C. **Padrões de Testes Automatizados.** - Depto de Ciência da Computação, Instituto de Matemática e Estatística - Universidade de São Paulo : [s.n.], 2011.

BURNSTEIN, I. **Practical Software Testing** [Livro]. - Chicago : Springer, 2003.

BUSCHMANN, F. [et al.] **Pattern-oriented software architecture: a system of patterns.** - 1996.

**CHIANG, T. C. Product and service testing methodology and ISO 9000** [Periódico]. - 1994.

DAMIANI, F. B. **Ferramenta de Mapeamento de UIDs para JSF.** - Florianópolis : [s.n.], 2011.

GRILO, A.M.P.; PAIVA, A.C.R. e FARIA, J.P. **Reverse Engineering of GUI Models for Testing.** – Depto de Engenharia Informática, Faculdade de Engenharia da Universidade do Porto : [s.n.], 2010.

**IEEE Standard for Software Verification and Validation Plans** [Online] // IEE Xplore Digital Library. -<http://ieeexplore.ieee.org/stamp/stamp.jsp? tp=&arnumber= 26585& isnumber =1028> Acesso em: 03 de Junho de 2013.

JSF [Online] // **JavaServer Faces. - Oracle**, 2013. <http://www.oracle.com/ technetwork/java /javaee/javaserverfaces-139869.html>. Acessado em: 3 de Junho de 2013

MARAFON D. L. **Integração JavaServer Faces e Ajax, Estudo da integração das tecnologias JSF e Ajax.** - UFSC/Brasil : [s.n.], 2006.

MCCLANAHAN C., BURNS E. e KITAIN R. **JAVASERVER™ FACES SPECIFICATION.** - 2004.

MEMON A. M., SOFFA M. L. e POLLACK M. E. **Coverage criteria for GUI testing.** - 2001.

MUGRIDGE R. e CUNNINGHAM Wa. **Fit for Developin Software: Framework for Integrated Tests.** - [s.l.] : Prentice Hall, 2005.

MYERS G. J. **The Art of Software Testing** [Livro]. - [s.l.] : John Wiley & Sons, Inc., 2004. - Vol. 2.

PATTON R. **Software Testing** [Livro]. - Indianapolis : Sams, 2001.

PRADHAN L. **User Interface Test Automation and its Challenges in an Industrial Scenario.** - Mälardalen UniversitY - Sweden : [s.n.], 2011.

RAGGETT D., HORS A. L. e JACOBS I. **HTML 4.01 specification. w3c recommendation** [Online]. - 1999. http://www.w3.org/tr/html4/ Acessado em 3 de Junho de 2013.

SELENIUM [Online] // **Selenium**. - Sauce Labs, 2013. < http://code.google.com /p/selenium /wiki/PageObjects> Acessado em: 3 de Junho de 2013.

SOFTWARE TESTING HELP [Online] // **Software Testing Help**. - 2013. - <http://www.softwaretestinghelp.com /what-is-performance-testing-load-testing-stress-testing/> Acessado em: 3 de Junho de 2013.

VILAIN P. **Modelagem da Interação com o Usuário em Aplicações.** - Puc-Rio : [s.n.], 2002..

VILAIN P. **Relatório Final, Funpesquisa 2003. Implementação de um Framework para Suporte à Representação de Requisitos Funcionais no Processo de Software.** - UFSC/Brasil: [s.n.], 2003.

# Apêndice A - Testes gerados para UIDs da aplicação de venda de CDs

A seguir será apresentado um exemplo de UID (Mostrar gravações de um CD) e para este exemplo serão mostradas as páginas JSF, classes de teste e classes Managed Bean geradas a partir da ferramenta desenvolvida.

- UID: Mostrar gravações de um CD

O UID da Figura A.1 foi especificado para representar o caso de uso Mostrar gravações de um CD:

1. Para um dado CD, o sistema mostra um conjunto com todas as suas gravações. Para cada música, é apresentado o nome da música, tempo de duração, cantor, compositor e letra da música.
2. Se o usuário desejar, uma gravação pode ser selecionada e um trecho desta gravação pode ser escutado.

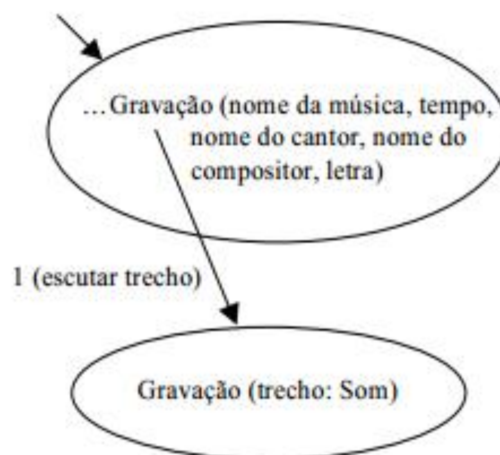



**Figura A.1 - Mostrar gravações de um CD**

A Figura A.2 apresenta o código JSF da página JSF gerada a partir do estado inicial de interação deste UID. Já a Figura A.3 representa a classe de objeto de página gerada para a página JSF apresentada na Figura A.2.

PinitialState.xhtml

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN' 'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
<html xmlns='http://www.w3.org/1999/xhtml' xmlns:h='http://java.sun.com/jsf/html' xmlns:f='http://java.sun.com/jsf/core'>
<h:head>
    <title>PinitialState</title>
</h:head>
<h:body>
    <h:form id='hidden' prependId='false'>
        <h:inputHidden id='nloEzY' />
    </h:form>
    <h:form id='PinitialState_forminitialState'>
        <h:dataTable id='dTable_Gravacao_0' value='#{pinitialStateMB.valueGravacao}'
            rendered='#{pinitialStateMB.renderTableValuedTable_Gravacao_0}' var='var'
            styleClass='dTable_Gravacao_0' border='1'>
            <f:facet name='header'>
                <h:outputText id='outText_dTable_Gravacao_0_Header_0'
                    styleClass='outText_dTable_Gravacao_0_Header_0'
                    value='#{pinitialStateMB.cabecalhoGravacao}' />

            </f:facet>
            <h:column>
                <f:facet name='header'>
                    <h:outputText id='outText_col_Header_2' styleClass='outText_col_Header_2'
                        value='nomeDaMusica' />
                </f:facet>
                <h:outputText id='outText_nomeDaMusica_1' styleClass='outText_nomeDaMusica_1'
                    value='#{var.nomeDaMusica}' />
            </h:column>

            <h:column>
                <f:facet name='header'>
                    <h:outputText id='outText_col_Header_4' styleClass='outText_col_Header_4'
                        value='tempo' />
                </f:facet>
                <h:outputText id='outText_tempo_3' styleClass='outText_tempo_3' value='#{var.tempo}' />
            </h:column>

            <h:column>
                <f:facet name='header'>
                    <h:outputText id='outText_col_Header_6' styleClass='outText_col_Header_6'
                        value='nomeDoCantor' />
                </f:facet>
                <h:outputText id='outText_nomeDoCantor_5' styleClass='outText_nomeDoCantor_5'
                    value='#{var.nomeDoCantor}' />
            </h:column>

            <h:column>
                <f:facet name='header'>
                    <h:outputText id='outText_col_Header_8' styleClass='outText_col_Header_8'
                        value='nomeDoCompositor' />
                </f:facet>
                <h:outputText id='outText_nomeDoCompositor_7' styleClass='outText_nomeDoCompositor_7'
                    value='#{var.nomeDoCompositor}' />
            </h:column>

            <h:column>
                <f:facet name='header'>
                    <h:outputText id='outText_col_Header_10' styleClass='outText_col_Header_10'
                        value='letra' />
                </f:facet>
                <h:outputText id='outText_letra_9' styleClass='outText_letra_9' value='#{var.letra}' />
            </h:column>

            <h:column>
                <f:facet name='header'>
                    <h:outputText id='outText_col_Header_11' styleClass='outText_col_Header_11'
                        value='escutarTrecho' />
                </f:facet>
                <h:commandLink id='cLink_Gravacao0_0' value='escutarTrecho' action='P0'
                    styleClass='cLink_Gravacao0_0' />
            </h:column>
        </h:dataTable>
    </h:form>
</h:body>
</html>
```

**Figura A.2 - Página JSF referente ao estado inicial de interação do UID apresentado.**

Para apresentar um exemplo de relacionamento entre a página JSF e a classe de objeto de página, será utilizado o elemento OutputText do JSF que está representado na linha 16 da Figura A.2. Este elemento possui os seguintes campos:

- id – este atributo está representado na Figura A.3, linha 16, como um atributo de classe com o mesmo valor definido na página JSF.

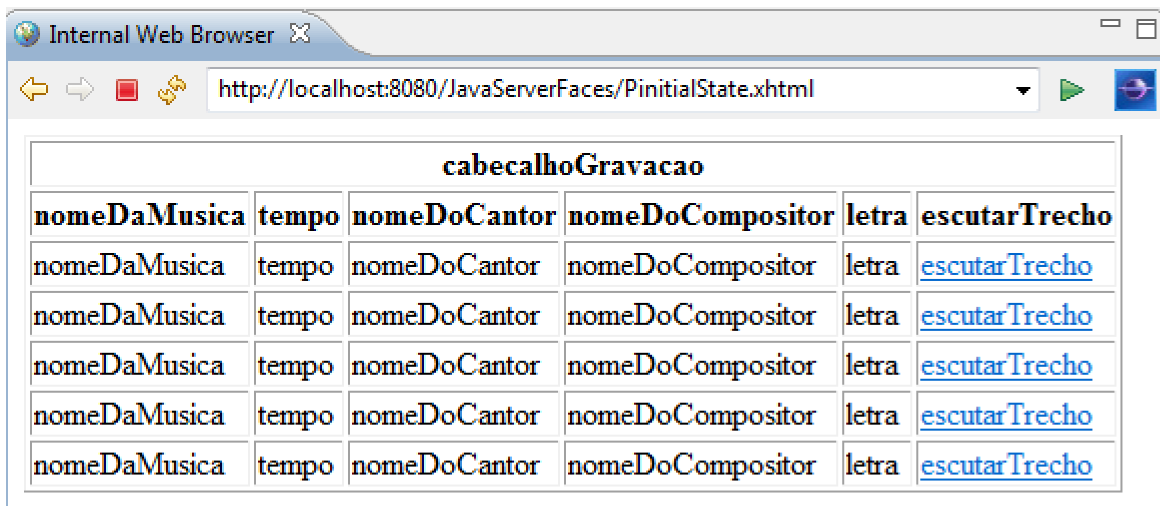
- *styleClass* – este atributo está representado na Figura A.3, linha 15, dentro da anotação @FindBy(name="*outText_dTable_Gravacao_0_Header_0*"). O campo *name* dentro da anotação, faz com o que framework Selenium busque, na página JSF, por elementos que possuam o atributo *styleClass* com valor igual ao valor a este campo.

- *value* – este valor tem por objetivo definir o valor que o componente deve assumir. Este valor pode ser definido explicitamente na página JSF ou então ser obtido através de uma classe tipo Managed Bean.

PinitialState.xhtml | PinitialStatePageObject.java

```java
package pageobject.gen;

import junit.framework.Assert;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

import pageobject.AbstractPageObject;

public class PinitialStatePageObject extends AbstractPageObject {
    // OUTPUTTEXT
    @FindBy(className = "outText_dTable_Gravacao_0_Header_0")
    private WebElement outText_dTable_Gravacao_0_Header_0;
    @FindBy(className = "outText_col_Header_2")
    private WebElement outText_col_Header_2;
    @FindBy(className = "outText_col_Header_4")
    private WebElement outText_col_Header_4;
    @FindBy(className = "outText_col_Header_6")
    private WebElement outText_col_Header_6;
    @FindBy(className = "outText_col_Header_8")
    private WebElement outText_col_Header_8;
    @FindBy(className = "outText_col_Header_10")
    private WebElement outText_col_Header_10;
    @FindBy(className = "outText_col_Header_11")
    private WebElement outText_col_Header_11;
    // COMMANDLINK
    @FindBy(className = "cLink_Gravacao0_0")
    private WebElement cLink_Gravacao0_0;

    /**
     * CONSTRUCTOR
     **/
    public PinitialStatePageObject(WebDriver driver) {
        super(driver);
    }

    @Override
    protected By getUniqueElement() {
        return By.id("nloEzY");
    }

    /**
     * ASSERTIONS - TEST METHODS
     **/
    public void assertPresence() {
        Assert.assertTrue(outText_dTable_Gravacao_0_Header_0.isDisplayed());
        Assert.assertTrue(outText_col_Header_2.isDisplayed());
        Assert.assertTrue(outText_col_Header_4.isDisplayed());
        Assert.assertTrue(outText_col_Header_6.isDisplayed());
        Assert.assertTrue(outText_col_Header_8.isDisplayed());
        Assert.assertTrue(outText_col_Header_10.isDisplayed());
        Assert.assertTrue(outText_col_Header_11.isDisplayed());
        Assert.assertTrue(cLink_Gravacao0_0.isDisplayed());

    }

    public void assertCLinksTexts() {
        Assert.assertEquals("escutarTrecho", cLink_Gravacao0_0.getText());

    }

    /**
     * TRANSACTIONS - TEST METHODS
     **/
    public P0PageObject transition_From_PinitialStatePageObject_to_P0PageObject_cLink_Gravacao0_0() {
        this.cLink_Gravacao0_0.click();
        return PageFactory.initElements(driver, P0PageObject.class);
    }
}
```

**Figura A.3 – Objeto de página referente a página JSF apresentada na Figura A.2.**




| cabecalhoGravacao | | | | | |
|---|---|---|---|---|---|
| **nomeDaMusica** | **tempo** | **nomeDoCantor** | **nomeDoCompositor** | **letra** | **escutarTrecho** |
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |

**Figura A.4 – Página web referente ao estado inicial do UID Mostrar gravações de um CD.**

As Figuras A.5 e A.6 representam, respectivamente, as classes Managed Bean e Managed Bean Helper utilizadas pela página JSF apresentada na Figura A.2. No momento em que a página JSF fizer a invocação de algum método presente na classe Managed Bean referente a ela, esta classe Managed Bean irá invocar o método da classe Managed Bean Helper que tenha a mesma assinatura da que foi invocada pela página JSF, porém a classe Helper se encarregará em gerar um valor para ser retornado à página web.

A geração de valores dentro das classes Managed Bean Helper se dará através das anotações do tipo @Podam presente sobre os atributos de classe. Para o exemplo na Figura A6, os seguintes valores serão gerados:

- Linhas 11 – será gerada uma coleção com cinco elementos do tipo Gravacao que serão utilizados para popular a tabela apresentada na Figura A.4.
- Linha 14 – será gerado um valor do tipo verdadeiro ou falso utilizado para determinar se a tabela será renderizada ou não na tela.
- Linha 15 – será gerado um valor responsável por preencher o título da tabela apresentada na Figura A.4

```java
PinitialStatePageObject.java   P0.xhtml   PinitialStateMB.java

1  package br.ufsc.common.mbeans;
2
3⊕import java.util.List;
11
12 @ManagedBean
13 @SessionScoped
14 public class PinitialStateMB {
15
16     private PinitialStateMBHelper mBeanHelper = new PinitialStateMBHelper();
17
18⊖    @PostConstruct
19     protected void initMBHelper() {
20         PodamFactoryImpl f = new PodamFactoryImpl();
21         this.mBeanHelper = f.manufacturePojo(PinitialStateMBHelper.class, PinitialStateMBHelper.class);
22     }
23
24⊖    public List<Gravacao> getValueGravacao() {
25         return mBeanHelper.getValueGravacao();
26     }
27
28⊖    public void setValueGravacao(List<Gravacao> valueGravacao) {
29         mBeanHelper.setValueGravacao(valueGravacao);
30     }
31
32⊖    public boolean isRenderTableValuedTable_Gravacao_0() {
33         return mBeanHelper.getRenderTableValuedTable_Gravacao_0();
34     }
35
36⊖    public String getCabecalhoGravacao() {
37         return mBeanHelper.getCabecalhoGravacao();
38     }
39
40⊖    public void setCabecalhoGravacao(String cabecalhoGravacao) {
41         mBeanHelper.setCabecalhoGravacao(cabecalhoGravacao);
42     }
43
44 }
45
```

**Figura A.5 – Classe Managed Bean referente a página JSF gerada para o estado inicial do UID apresentado.**

```
PinitialStatePageObject.java   P0.xhtml   PinitialStateMB.java   *PinitialStateMBHelper.java
1  package br.ufsc.common.mbeans;
2⊕ import java.util.List;
8
9  public class PinitialStateMBHelper {
10
11⊖     @PodamCollection(nbrElements = 5)
12     private List<Gravacao> valueGravacao;
13⊖     @PodamBooleanValue(boolValue = true)
14     private boolean renderTableValuedTable_Gravacao_0;
15⊖     @PodamStringValue(strValue = "cabecalhoGravacao")
16     private String cabecalhoGravacao;
17
18⊖     public void setValueGravacao(List<Gravacao> valueGravacao) {
19         this.valueGravacao = valueGravacao;
20     }
21
22⊖     public List<Gravacao> getValueGravacao() {
23         return valueGravacao;
24     }
25
26⊖     public void setRenderTableValuedTable_Gravacao_0(boolean renderTableValuedTable_Gravacao_0) {
27         this.renderTableValuedTable_Gravacao_0 = renderTableValuedTable_Gravacao_0;
28     }
29
30⊖     public boolean getRenderTableValuedTable_Gravacao_0() {
31         return renderTableValuedTable_Gravacao_0;
32     }
33
34⊖     public void setCabecalhoGravacao(String cabecalhoGravacao) {
35         this.cabecalhoGravacao = cabecalhoGravacao;
36     }
37
38⊖     public String getCabecalhoGravacao() {
39         return cabecalhoGravacao;
40     }
41
42 }
43
```

**Figura A.6 - Classe Managed Bean Helper referente a página JSF gerada para o estado inicial do UID apresentado.**

A Figura A.7 apresenta a página JSF referente ao segundo estado de interação deste UID.

```
PinitialStatePageObject.java   P0.xhtml   PinitialStateMB.java   *PinitialStateMBHelper.java
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC '-//W3C//DTD XHTML 1.0 Transitional//EN' 'http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd'>
3⊖ <html xmlns='http://www.w3.org/1999/xhtml'
4  xmlns:h='http://java.sun.com/jsf/html'
5  xmlns:f='http://java.sun.com/jsf/core'>
6  <h:head><title>P0</title></h:head>
7⊖ <h:body>
8⊖ <h:form id ='P0_form0'>
9⊖ <h:outputLabel id='outLbl_Gravacao_0' styleClass='outLbl_Gravacao_0' value='#{p0MB.labelGravacao}'>
10⊖ <h:panelGrid columns='1'>
11 <h:outputText id='outText_trecho_12' styleClass='outText_trecho_12' value='trecho'/>
12
13 </h:panelGrid>
14 </h:outputLabel>
15
16 </h:form>
17 <h:form id='hidden' prependId='false'> <h:inputHidden id='xSEZky'/></h:form></h:body>
18 </html>
```

**Figura A. 7 - página JSF referente ao segundo estado de interação do UID apresentado.**

A Figura A.8 apresenta o objeto de página referente à página JSF gerada para o segundo estado de interação deste UID.

```
P0.xhtml | PinitialStateMB.java | *PinitialStateMBHelp | P0MB.java | P0MBHelper.java | P0PageObject.java

1  package pageobject.gen;
2
3⊕ import junit.framework.Assert;
12 public class P0PageObject extends AbstractPageObject {
13     // OUTPUTTEXT
14⊖    @FindBy(className = "outText_trecho_12")
15     private WebElement outText_trecho_12;
16     // OUTPUTLABEL
17⊖    @FindBy(className = "outLbl_Gravacao_0")
18     private WebElement outLbl_Gravacao_0;
19
20⊖    /**
21      * CONSTRUCTOR
22      **/
23⊖    public P0PageObject(WebDriver driver) {
24         super(driver);
25     }
26
27⊖    @Override
28     protected By getUniqueElement() {
29         return By.id("xSEZky");
30     }
31
32⊖    /**
33      * ASSERTIONS - TEST METHODS
34      **/
35⊖    public void assertPresence() {
36         Assert.assertTrue(outText_trecho_12.isDisplayed());
37         Assert.assertTrue(outLbl_Gravacao_0.isDisplayed());
38
39     }
40
41 }
```

**Figura A.8 - Objeto de página referente a página JSF gerada para o segundo estado de interação deste UID.**

A Figura A.9 apresenta a página JSF referente ao segundo estado de interação deste UID.

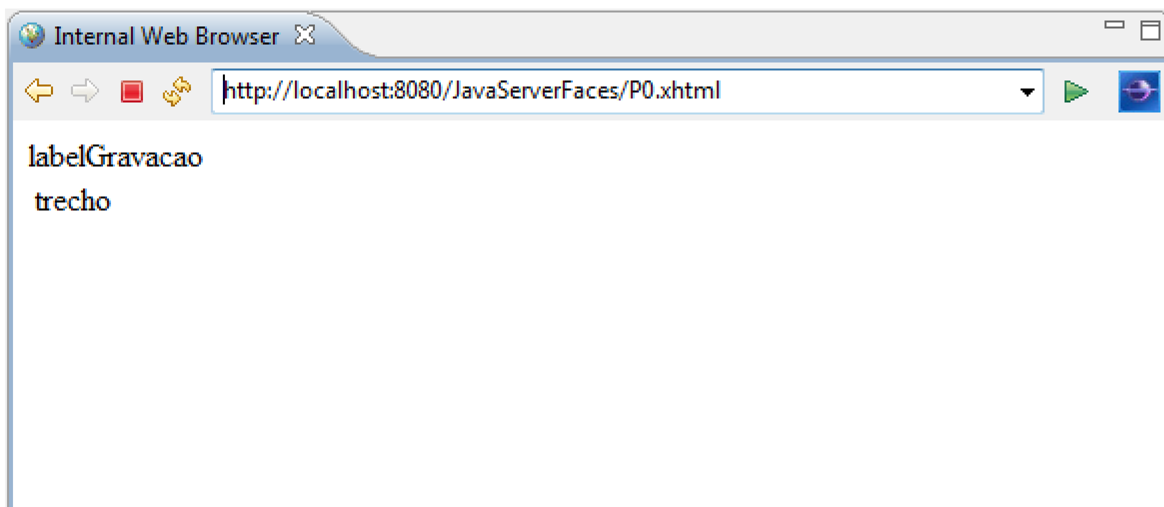



**Figura A.9 - Página referente ao segundo estado de iteração do UID Mostrar Gravação.**

A Figura A.10 apresenta a classe Managed Bean referente ao segundo estado de interação deste UID.

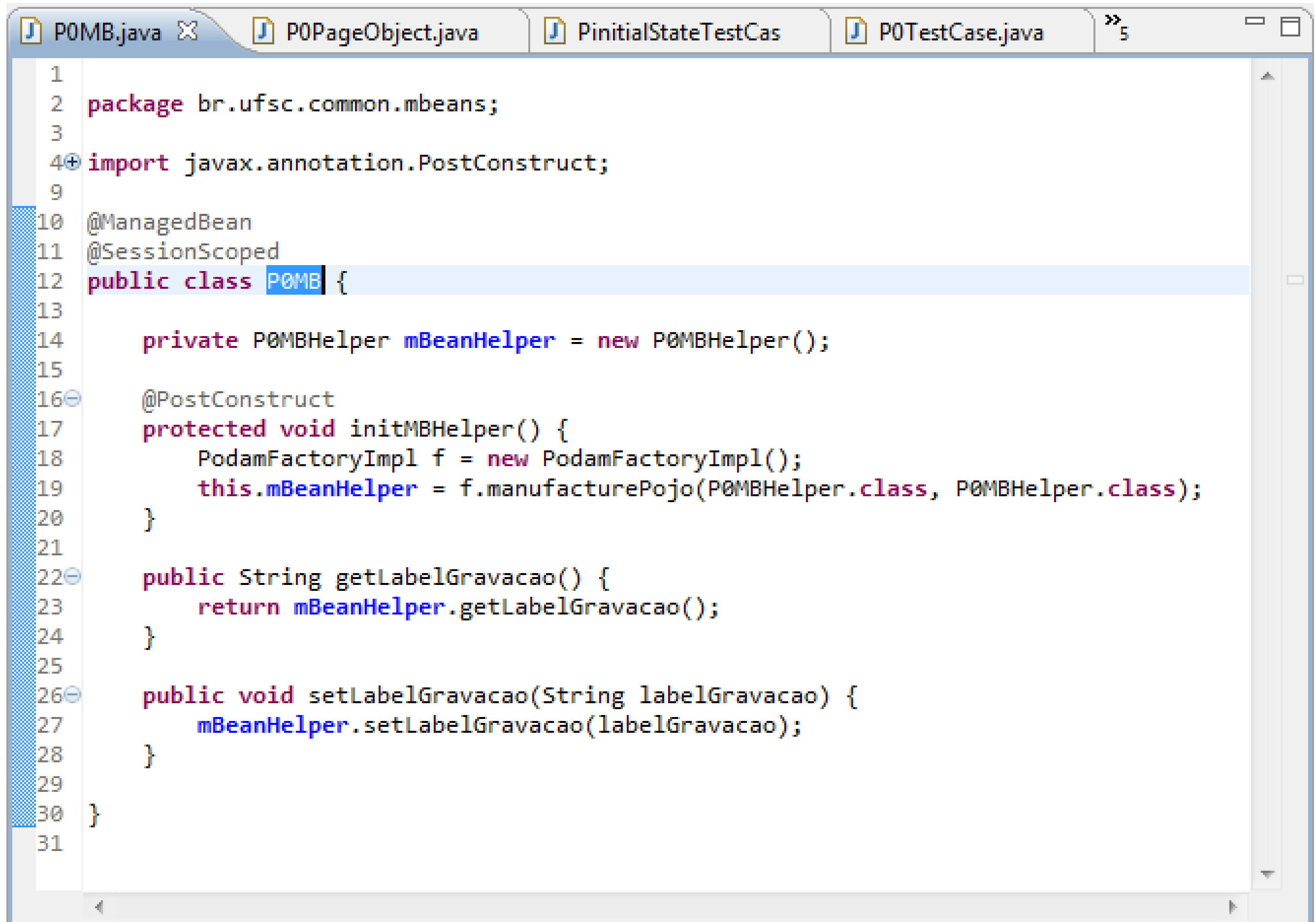



**Figura A.10 – Classe Managed Bean referente a página JSF gerada para o segundo estado de intereção do UID apresentado.**

A Figura A.11 apresenta a classe Managed Bean Helper referente à página JSF gerada para o segundo estado de interação do UID apresentado.

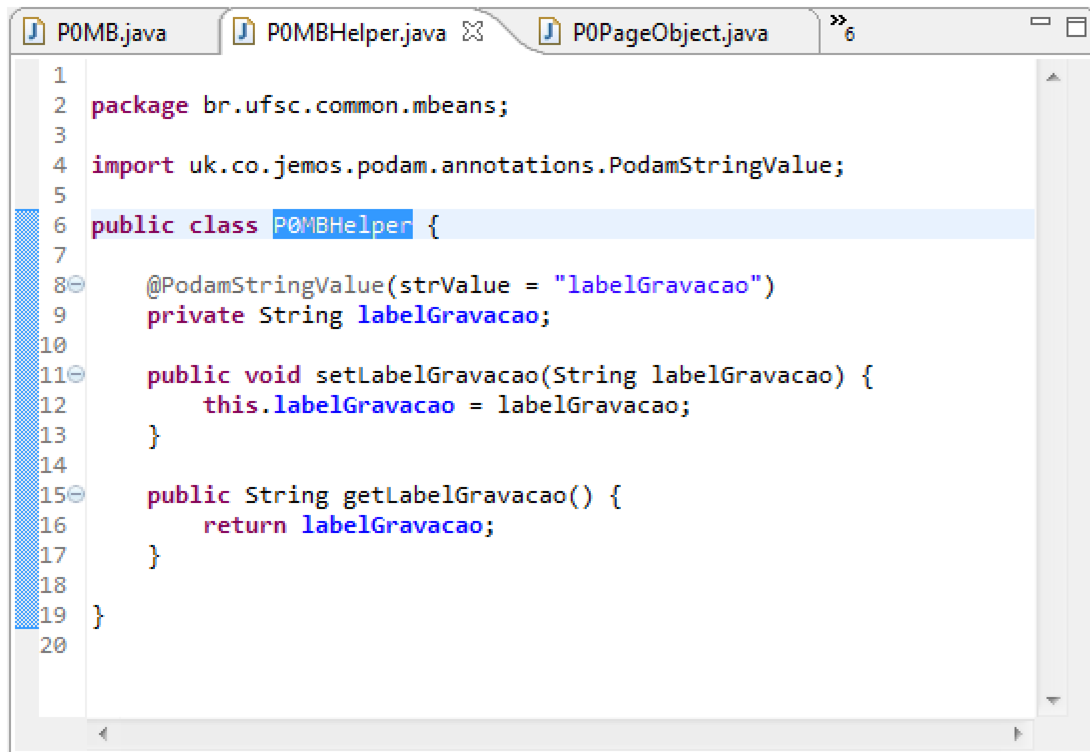



**Figura A.11 – Classe Managed Bean Helper referente à página JSF gerada para o segundo estado de interação do UID apresentado.**

Na sequência serão apresentadas as classes de caso de teste geradas para validar os testes automatizados propostos por essa ferramenta. A Figura A.12 apresenta a classe de teste que tem por responsabilidade executar os testes relacionados à página JSF gerada para o estado inicial de interação do UID sendo apresentado.

Para a execução correta dos casos de teste abaixo apresentados, faz-se necessário que o servidor de aplicação web, o qual disponibiliza as páginas JSF na web, esteja ativo. No momento em que esta classe for executada (utilizando o framework JUnit), os métodos serão executados na ordem que estão apresentados, com exceção dos métodos que possuem a anotação @Test. Estes serão executados na ordem em que o framework JUnit julgar correto.:

1. Método *init()* - (Linha 22): este método será executado apenas uma vez, antes de todos os outros métodos e tem por responsabilidade inicializar uma instância do navegador Firefox e também inicializar  o driver do Selenium o qual fará a comunicação com as páginas JSF testadas.

2. Método *startPage()* - (Linha 30): este método será executado anteriormente a cada método de teste que possua a anotação @Test. Ele tem por objetivo garantir que os testes sempre serão inicializados a partir da página principal sendo testada.

3. Método *testLinksTransitions0()* - (Linha 35): este método tem por funcionalidade testar o elemento do tipo CommandLink apresentado na Figura A.2, linha 69. O teste se dará através da simulação de utilizando o método de transição apresentado na linha 68 da Figura A.3. Caso o link navegue para a página correta, o resultado do teste será igual a sucesso. Caso ocorra um erro ao navegar, o teste será considerado como falho.

4. Método *assertPresence()* - (Linha 41): este método tem por funcionalidade, através do objeto de página gerado para a página JSF referente ao estado inicial de interação do UID apresentado, verificar os elementos JSF estão visíveis na interface.

5. Método *assertCLinksText()* - (Linha 47): este método visa garantir que o texto de um elemento do tipo CommandLink, o qual tenha sido definido já na especificação do UID, tenha sido renderizado na tela com o texto idêntico ao usado pelo projetista durante a modelagem do UID.

6. Método *end()* - (Linha 52): este método será executado apenas após o momento em que todos os demais métodos tenham sido executados. Sua funcionalidade é a de solicitar que seja fechado o navegador que estava sendo usado para testes.

A Figura A.12 apresenta a classe de casos de teste referente à página JSF gerada para o segundo estado de interação do UID apresentado.

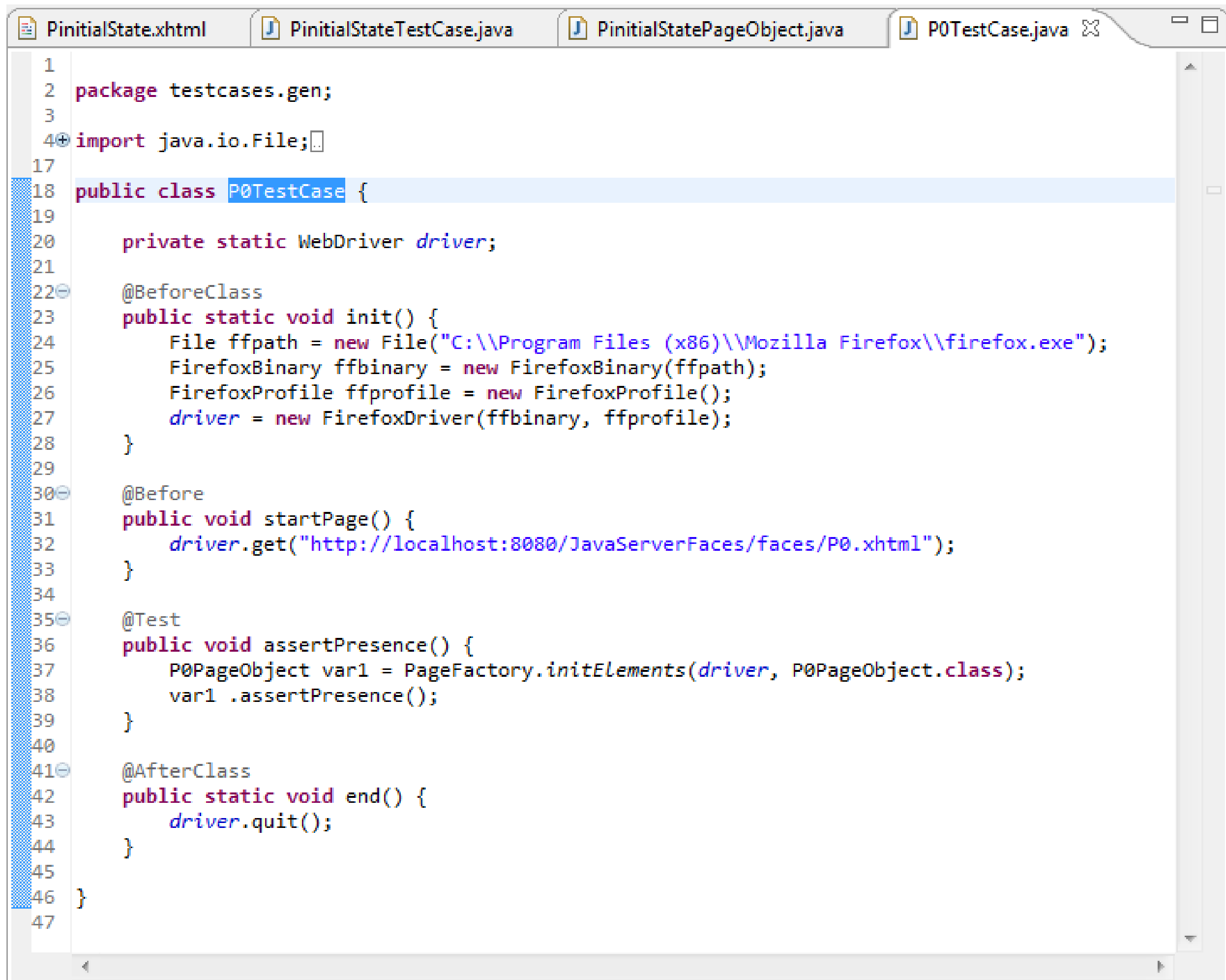



**Figura A. 12 - Classe de casos de teste referente à página JSF gerada para o segundo estado de interação do UID apresentado**

Como pode ser visto na figura A.12, os métodos de teste de transição e teste de asserção de nomes de links não foram gerados pelo fato de não existirem elementos do tipo CommandLink no conteúdo da página JSF gerada para o segundo estado de interação do UID apresentado.

# Apêndice B – Código Fonte

O Código Fonte das regras para geração de classes de teste de interface para páginas JSF encontra-se anexado em CD.

# Apêndice C - Artigo

## Ferramenta para Teste de Páginas JSF Geradas a partir de UID

**Éric Felipe Barboza**[1]

[1] Departamento de Informática e Estatística - INE – Universidade Federal de Santa Catarina (UFSC)

Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brasil

ericfb1@gmail.com

***Abstract.*** *This paper aims to present and describe a tool for generating test cases for automated JSF pages from the mapping of UIDs (User Interaction Diagram) to JavaServer Faces (JSF) pages. The process of generating this tool involved the use of tests automation supporting tools and Java dynamic code generation tools. The tools used were: Selenium 2 (test) JUnit (testing), Velocity (generate Java code), Code Model (generate Java code), prune (generate dummy data to Java objects).*

***Resumo.*** *Este trabalho tem por objetivo apresentar e descrever uma ferramenta para geração de casos de teste automatizados para páginas JSF a partir do mapeamento de UIDs (User Interaction Diagram) em páginas JSF (JavaServer Faces). O processo de geração da ferramenta envolveu a utilização de ferramentas de suporte a automatização de testes e ferramentas de geração dinâmica de código Java. As ferramentas utilizadas foram: Selenium 2 (teste) JUnit (teste) , Velocity (gerar código Java ) , Code Model (gerar código Java ) , Podam (gerar dados fictícios para objetos Java).*

## 1. Introdução

Processos de testes são de extrema importância no ciclo de desenvolvimento de um projeto, seja este na área de TI ou em qualquer outra área de desenvolvimento de produtos [CHIANG, 1994]. Um interesse comum de todas as empresas é o de poder entregar seu produto final ao cliente sem que este esteja com erros ou defeitos. Porém, devido à falta de tempo e à intensa cobrança de seus clientes, estas empresas acabam entregando seus produtos sem que estes tenham passado pelos testes necessários.

A confiabilidade de um software é algo bastante complexo de se discutir, tanto por não ser algo mensurável como por não ser algo possível de garantir. Por isso, para aumentar confiabilidade de um software, a confiabilidade deve ser tratada em nível de sistema e não

somente em nível de componente. Isso porque em um sistema os componentes são interdependentes, e uma falha em um componente por mais simples que este seja pode se propagar pelo sistema e afetar a funcionalidade de outros componentes [AGARWAL, TAYAL e GUPTA, 2010].

Agarwal, Tayal e Gupta (2010) apresentaram três definições para confiabilidade:

*Confiabilidade de Software é definida como a probabilidade que um programa de computador tem de operar livre de falhas em um determinado ambiente durante um determinado tempo.*

*OU*

*Confiabilidade de um produto de software denota essencialmente sua confiabilidade ou segurança.*

*OU*

*Confiabilidade de um produto de software também pode ser definida como a probabilidade do produto funcionar corretamente durante um dado período de tempo.*

Hoje em dia ainda é possível ver inúmeras empresas que ao final do desenvolvimento de um módulo específico ou de um sistema inteiro, contam apenas com testes manuais para garantir a integridade e eficiência de seu produto [BERNARDO, 2011].

Deve-se saber que o processo de testes, independente da ferramenta ou produto que venha a ser testado, pode ser iniciado logo que os requisitos sejam estabelecidos pelo cliente. Uma área até então não muito explorada é a da geração de interfaces a partir de modelagens diagramáticas, tendo a ferramenta de mapeamento de UIDs para JSF, chamada UID2JSF, como um dos recentes projetos nessa área e que servirá de base para este trabalho [DAMIANI, 2011]. O UID (Diagrama de Interação do Usuário) é uma ferramenta diagramática que representa a interação do usuário com o sistema. A elaboração dos UIDs é feita pelo analista durante a tarefa de levantamento de requisitos de uma aplicação, juntamente com os casos de uso. Os UIDs contribuem com a tarefa de levantamento de requisitos, e ainda podem ser utilizados nas etapas de projeto conceitual, projeto navegacional e projeto da interface com o usuário [VILAIN, 2002].

Já o Java Server Faces (JSF) é um framework para desenvolvimento de aplicações Web que utilizam tecnologia Java. Este trabalha principalmente nas camadas de Visão e Controle do modelo MVC [BUSCHMANN, MEUNIER, ROHNERT, SOMMERLAD e STAL 1996]. Seu principal objetivo é facilitar o trabalho dos desenvolvedores na tarefa de construção das páginas das aplicações Web [MARAFON 2006].

Visto que a ferramenta de mapeamento UID2JSF irá gerar páginas JSF a partir de regras e fluxos estabelecidos nos UIDs, se torna de grande valia a aplicação de testes de interface baseados em UIDs. A criação de testes junto com o processo de análise de requisitos garante que os futuros desenvolvimentos da aplicação, já estarão sendo cobertos por pelo menos alguns testes. Além disso, os testes gerados poderão ser modificados ou incrementados, de acordo com a necessidade dos testadores.

# 2. Ferramenta de Mapeamento UID2JSF

A ferramenta UID2JSF tem por objetivo efetuar o mapeamento dos UIDs (User Interaction Diagrams) para páginas JSF (Java Server Faces). Para que este mapeamento possa ocorrer com sucesso, Damiani (2011) desenvolveu um protótipo capaz de ler arquivos XML que representam os UIDs e transformar as informações obtidas em páginas JSF.

Vilain (2003), visando facilitar o armazenamento e também intercâmbio das instâncias dos UIDs entre aplicações, desenvolveu a especificação de uma DTD (Document Type Definition) para que os UIDs pudessem ser representados através de arquivos XML (Extensible Markup Language). É a partir desses arquivos XML que as interfaces serão geradas.

Durante o desenvolvimento da ferramenta UID2JSF, foram definidas regras para mapear cada componente dos UIDs em componentes das páginas JSF [DAMIANI, 2011]. A Tabela 1.1 apresenta de uma forma resumida as regras de mapeamento de cada elemento de um UID para componentes JSF utilizadas pela ferramenta UID2JSF.

**Tabela 1.1 - Tabela de mapeamento dos elementos do UID para componentes JSF**

| **Entradas do Usuário** | |
|---|---|
| **UID** | **JSF** |

| | |
|---|---|
| Item de Dado | Input Text |
| Estrutura | Input Text com um Output Label<br>Panel Grid com um Output Label |
| Conjunto de Itens de Dado | Um Input Text para cada item de dado do conjunto<br><br>Apenas um Input Text |
| Conjunto de Estruturas | Data Table com uma coluna do tipo Input<br><br>Text para cada elemento da estrutura<br>Apenas uma Estrutura de entrada de usuário |
| Entrada de usuário Enumerada | Select One Radio<br><br>Select Many Checkbox |
| Seleção dentre dois Itens de Dados (OR) | Select Many Checkbox |
| Seleção de um Item de Dado (XOR) | Select One Radio |
| **Saídas do Sistema** | |
| **UID** | **JSF** |
| Texto | Output Text |
| Item de Dado | Output Text<br><br>Command Link |
| Estrutura | Output Text<br><br>Command Link<br>Panel Grid com Output Label |
| Conjunto de Itens de Dado | Data Table com apenas uma coluna. |
| Conjunto de Estruturas | Data Table com uma coluna para cada<br>elemento da estrutura. |
| **Demais Elementos** | |

| UID | JSF |
|---|---|
| Estado da Interação, Estado Inicial da Interação | Página XHTML com Form |
| Sub-estado de um Estado de Interação | Form |
| Transição com Seleção de N Elementos | Command Link<br>Select Many Checkbox |
| Transição com Seleção da Opção X | Command Link |

## 2.1. Exemplo de mapeamento UID2JSF

Nesta seção será apresentado um exemplo de um UID e em seguida o seu mapeamento deste UID para páginas JSF utilizando a ferramenta UID2JSF.

O UID mapeado é correspondente ao caso de uso Mostrar gravações de um CD, mostrado a seguir.

- Caso de Uso: Mostrar gravações de um CD

  1. Para um dado CD, o sistema mostra um conjunto com todas as suas gravações. Para cada música, é apresentado o nome da música, tempo de duração, cantor, compositor e letra.

  2. Se o usuário desejar escutar um trecho de uma música, uma gravação pode ser selecionada.

  A Figura 2.1 apresenta o UID "*Mostrar gravações de um CD*".

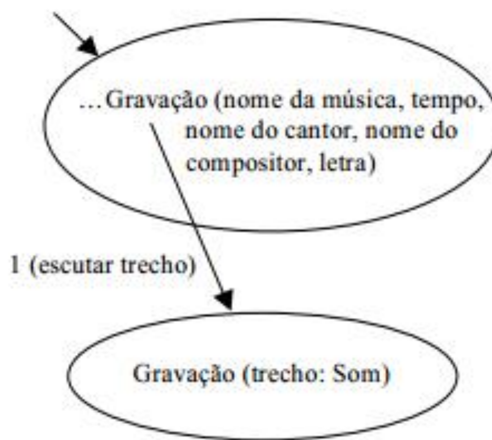



**Figura 2.1 - Mostrar gravações de um CD.**

A Figura 2.2 apresenta a página JSF gerada a partir do estado inicial de interação do UID apresentado anteriormente. O estado está representado por um componente *"form"* que não pode ser visualizado na imagem. O conjunto de estruturas originou a tabela e cada item de dado da estrutura "Gravação" ficou representado por uma coluna da tabela, assim como a transição "escutar trecho" que é representada pela última coluna. Já os elementos do conjunto, isto é, as estruturas do tipo gravação, são as linhas da tabela.

http://localhost:8080/tcc.paginas/mostrarGravacoesDeUmCD/pagina0.jsf

| Gravacao | | | | | |
|---|---|---|---|---|---|
| nomeDaMusica | tempo | nomeDoCantor | nomeDoCompositor | letra | escutarTrecho |
| Rock N Roll Train | 4:21 | AC/DC | Brendan O´Brien | One hot angel, one cool devil... | escutarTrecho |
| Skies on Fire | 3:34 | AC/DC | Brendan O´Brien | Why don't you hang up,... | escutarTrecho |
| Big Jack | 3:57 | AC/DC | Brendan O´Brien | The steam is a burning,... | escutarTrecho |
| Anything Goes | 3:22 | AC/DC | Brendan O´Brien | Got a taste of a rocking band... | escutarTrecho |
| War Machine | 3:09 | AC/DC | Brendan O´Brien | Push your foot to the floor... | escutarTrecho |
| Smash N Grab | 4:06 | AC/DC | Brendan O´Brien | Come on and blow your mind,... | escutarTrecho |
| Spoilin´ for a Fight | 3:17 | AC/DC | Brendan O´Brien | I see trouble coming man,... | escutarTrecho |
| Wheels | 3:28 | AC/DC | Brendan O´Brien | She was a danger,... | escutarTrecho |
| Decibel | 3:34 | AC/DC | Brendan O´Brien | Take up all your time,... | escutarTrecho |
| Storm May Day | 3:10 | AC/DC | Brendan O´Brien | The storm is ragin',... | escutarTrecho |
| She Likes Rock N Roll | 3:53 | AC/DC | Brendan O´Brien | A little game of falling down,... | escutarTrecho |
| Money Made | 4:15 | AC/DC | Brendan O´Brien | Work work money made,... | escutarTrecho |
| Rock N Roll Dream | 4:41 | AC/DC | Brendan O´Brien | Deep water all around me,... | escutarTrecho |
| Rocking All the Way | 3:22 | AC/DC | Brendan O´Brien | Well, one mad shuffle,... | escutarTrecho |
| Black Ice | 3:25 | AC/DC | Brendan O´Brien | Well the devil may care,... | escutarTrecho |

**Figura 2.2 - Página referente ao estado inicial do UID Mostrar gravações de um CD**

# 3. Criação de testes para páginas JSF

Seguindo definições de melhores práticas de programação, foi utilizado no desenvolvimento deste trabalho o padrão de projeto nomeado Page Objects (Objetos de página). De acordo com esse padrão, cada página JSF da aplicação deve possuir um objeto de página referente a ela.

Um objeto de página funciona como uma ligação entre os casos de teste e as interfaces a serem testadas. Esses objetos permitem que os casos de teste acessem elementos e informações presentes em uma página JSF.

Para que os objetos de página possam ser construídos, faz-se necessário obter as informações, características e funcionalidades que cada página possua. Neste trabalho essas informações foram obtidas a partir dos elementos JSF oriundos das transformações de UIDs para páginas JSF. Processo de extração de informações dos componentes JSF

A ideia de extração das informações necessárias para geração do projeto de teste é relativamente simples. No momento da criação do componente JSF, este é registrado em uma classe do projeto, chamada **ProjectInfoHolder**. Esta classe, que foi definida neste trabalho, terá por função guardar a lista de componentes criados para cada página. Cada elemento armazenado terá consigo informações úteis no momento da criação dos testes, como por exemplo o identicador de localização e a maneira pela qual o Selenium deverá localizar cada elemento

### 3.1. Processo de criação de objetos de páginas

Para cada página JSF mapeada pela ferramenta UID2JSF, será criado um Objeto de página com o seguinte padrão de nomenclatura: ***P*** + *nome da página* + ***PageObject.*** Cada objeto de página gerado conterá atributos de classe do tipo WebElement, os quais farão referência a objetos presentes nas páginas JSF. Além disso, os objetos de página também conterão métodos de validação que serão executados de acordo com os casos de teste criados.

### 3.2. Localização de elementos JSF dentro de objetos de página

Neste trabalho, a forma de localização padrão de um elemento JSF pela ferramenta Selenium se dará através do atributo “styleClass” de cada componente JSF. Para isso, elementos que foram mapeados pela ferramenta UID2JSF e armazenados na classe ProjectInfoHolder, serão declarados como atributos de classe do tipo WebElement e receberão a anotação @FindBy(className=”*ID componente*”), onde o valor entre aspas corresponderá ao ID gerado para o componente JSF. Porém, o componente do tipo SelectOneRadio, responsável por renderizar *radio buttons* na tela, não suporta a definição do atributo “styleClass” para seus componentes filhos, portanto, quando for necessário trabalhar com esse tipo de campo o mesmo terá de ser localizado através do driver do Selenium, o qual consegue encontrar elementos pais e seus descendentes através do método *findElements(By)*.

A Figura 3.1 apresenta um exemplo de uma página JSF gerada a partir da ferramenta UID2JSF. Esta tela é composta por três componentes visíveis que, em ordem de apresentação, são: Outputlabel, InputText, CommandLink.

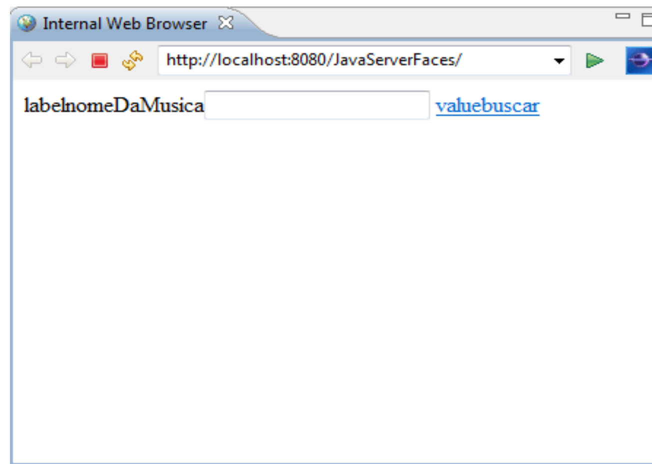



**Figura 3.1 - Exemplo de tela mapeada a partir da ferramenta UID2JSF**

Na sequência, a Figura 3.2 apresenta o objeto de página gerado para esta tela. Este objeto possui duas instâncias de objetos do tipo WebElement que fazem referência aos dois objetos presentes na Figura 3.1.

```
public class PinitialStatePageObject extends AbstractPageObject {
    // OUTPUTLABEL
    @FindBy(className = "outLbl_nomeDaMusicalabel0")
    private WebElement outLbl_nomeDaMusicalabel0;
    // COMMANDLINK
    @FindBy(className = "cLink_00")
    private WebElement cLink_00;

    /**
     * CONSTRUCTOR
     **/
    public PinitialStatePageObject(WebDriver driver) {
        super(driver);
    }
```

**Figura 3.2 - Exemplo de objeto de página**

## 3.3. Métodos de teste para objetos de página

Métodos de página são métodos que auxiliam no processo dos testes de interface. Esses métodos são responsáveis por prover acesso a elementos de uma interface e ações que ela possua, como por exemplo, a validação de textos em componentes, as possíveis transições que a página possa fazer de uma interface para outra, além de outras validações que o testador necessite fazer, bastando implementá-las.

Os métodos de testes a serem gerados automaticamente dentro dos Objetos de página serão os métodos utilizados pelas classes de caso de teste no momento em que forem executadas. Cada método será invocado uma única vez no momento em que um for executada a classe de casos de teste referente a esta página.

A Figura 3.3 apresenta os quatro tipos de teste de página desenvolvidos neste trabalho. O primeiro método funciona como um identificador para validação de página. Este método é executado sempre que uma nova instância de objeto de página for criada. Isso ocorre com o objetivo de garantir que o identificador único que o objeto de página possui é o mesmo presente na página JSF que está rodando no browser no momento. Caso a asserção falhe, o teste também falhará. Os dois métodos seguintes serão "invocados" pelos casos de teste gerados. Esses métodos servem, respectivamente, para garantir que os elementos gerados pela ferramenta UID2JSF foram renderizados corretamente na tela, e que os textos de links definidos nos UIDs estão associados corretamente aos seus links. Por fim, o último método faz referência à navegação entre páginas. Para cada link presente na tela, haverá um método de transição para testar se a navegação feita pelo link está de acordo com o que foi definido no UID.

```
    @Override
    protected By getUniqueElement() {
        return By.id("IDOUO0");
    }

    public void assertPresence() {
        Assert.assertTrue(outLbl_CD3.isDisplayed());
        Assert.assertTrue(cLink_CD44.isDisplayed());
        Assert.assertTrue(outText_014.isDisplayed());
    }

    public void assertCLinksTexts() {
        Assert.assertEquals("comprar", cLink_CD44.getText());
    }

    public PinitialStatePageObject transition_From_P1PageObject_to_PinitialStatePageObject_cLink_CD44() {
        this.cLink_CD44.click();
        return PageFactory.initElements(driver, PinitialStatePageObject.class);
    }
}
```

**Figura 3.3 - Exemplo de métodos de teste em PO**

## 3.4. Processo de criação de casos de teste

Assim como ocorre com os objetos de página e classes Managed Bean, será gerada uma classe de caso de teste para cada página JSF. Essa classe terá por responsabilidade efetuar a invocação de métodos presentes no objeto de página referente à página JSF.

A Figura 3.4 apresenta um exemplo de classe de caso de teste gerado automaticamente. Essa classe utiliza comandos da biblioteca JUnit visando estruturar o processo de execução dos testes. Assim como na Figura abaixo, todas as classes de teste que forem geradas neste trabalho possuirão a seguinte estrutura:

- Método init() utilizando a anotação @BeforeClass – tem como funcionalidade a abertura do navegador antes de iniciar a execução geral dos testes. Neste trabalho os testes acontecerão utilizando apenas o Browser Firefox.

- Método startPage() utilizando a anotação @Before – tem como funcionalidade a navegação para a página testada. Esse método é executado antes da execução de cada método de testes.

- Métodos com a anotação @Test – serão os métodos de teste que farão a invocação dos métodos definidos nos objetos de página.

- Método end() utilizando a anotação @AfterClass – tem como funcionalidade fechar o Browser após a execução de todos os casos de teste.

```java
public class PinitialStateTestCase {

    private static WebDriver driver;
    @BeforeClass
    public static void init() {
        File ffpath = new File("C:\\Program Files (x86)\\Mozilla Firefox\\firefox.exe");
        FirefoxBinary ffbinary = new FirefoxBinary(ffpath);
        FirefoxProfile ffprofile = new FirefoxProfile();
        driver = new FirefoxDriver(ffbinary, ffprofile);
    }

    @Before
    public void startPage() {
        driver.get("http://localhost:8080/JavaServerFaces/faces/PinitialState.xhtml");
    }

    @Test
    public void testLinksTransitions0() {
        PinitialStatePageObject var1 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var1.transition_From_PinitialStatePageObject_to_PinitialStatePageObject_cLink_00();
    }

    @Test
    public void assertPresence() {
        PinitialStatePageObject var3 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var3.assertPresence();
    }

    @Test
    public void assertCLinksTexts() {
        PinitialStatePageObject var4 = PageFactory.initElements(driver, PinitialStatePageObject.class);
        var4.assertCLinksTexts();
    }

    @AfterClass
    public static void end() {
        driver.quit();
    }
```

**Figura 3.4 - Exemplo de classe de caso de teste**

## 3.5. Processo de execução das classes de teste geradas.

Após o processo de geração de testes de interfaces para as páginas JSF oriundas do mapeamento de UID, é possível executar as classes de teste geradas para que os testes sejam aplicados sobre as interfaces. Para isso faz-se necessário que o usuário testador tenha configurado o ambiente onde a aplicação web estará instalada. Feito esta instalação, baste que, utilizando a IDE eclipse, siga- se os passos apresentados a seguir:

4. Selecionar a classe a ser testada.

5. Ir até o menu Run do eclipse (Figura 3.5)

6. Escolher a opção Run As > JUnit Test.

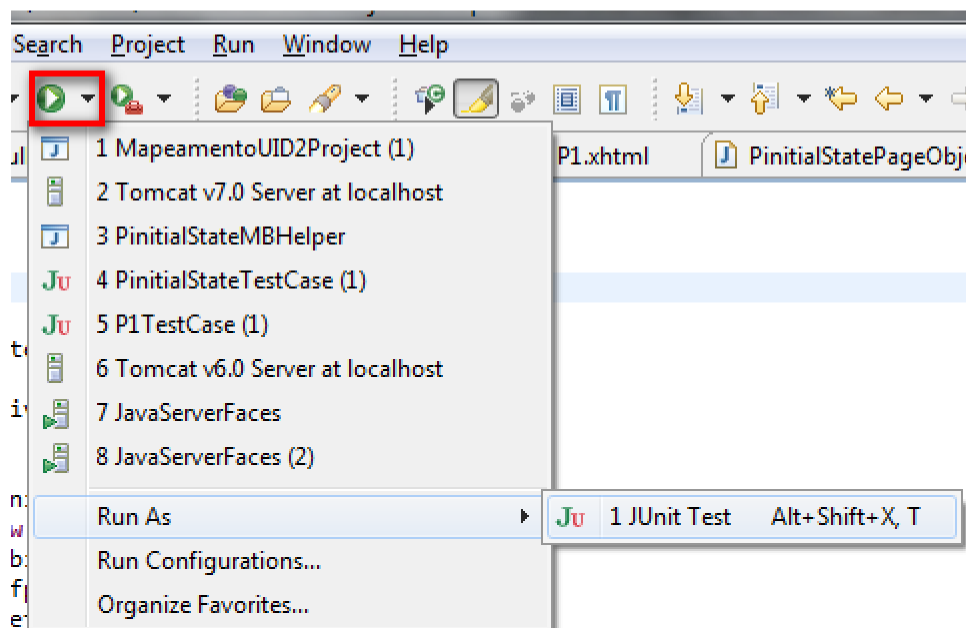



**Figura 3.5 - Exemplo de execução de classe de teste.**

# Conclusão

Com o crescimento do número de pessoas conectadas à internet no mundo todo, grandes e pequenas empresas estão voltando seus esforços para disponibilizar seus serviços na web. Porém, junto com essa necessidade das empresas em de disponibilizar seus serviços, vem a pressa, esse, além do desenvolvimento de má qualidade, é um fator que faz com que sistemas não sejam desenvolvidos nem testado da melhor forma. Portanto, a presença de ferramentas que automatizem e agilizem o processo de testes é extremante importante no ciclo de desenvolvimento de sistemas. A automatização, além de outras vantagens, permite que menos dinheiro seja gasto com correções de problemas e sobre mais tempo para planejamentos e desenvolvimentos com qualidade.

A ferramenta de geração de testes para páginas JSF geradas a partir do mapeamento de UIDs, garante que o processo de teste seja iniciado já no momento da obtenção de requisitos junto ao cliente. Os testes gerados pela ferramenta poderão ser reutilizados e adaptados pelos

testadores da futura aplicação web criada, e as classes e projetos gerados servirão como um protótipo inicial de estrutura de organização dos testes da aplicação.

# Referências Bibliográficas


**AGARWAL B B, TAYAL S P e GUPTA M** Software Engineering and Testing: An Introduction (Computer Science) [Livro]. - [s.l.] : Jones and Bartlett Publishers, 2010. - pp. 107,89.

**BERNARDO Paulo Cheque** Padrões de Testes Automatizados. - Depto de Ciência da Computação, Instituto de Matemática e Estatística - Universidade de São Paulo : [s.n.], 2011.

**BUSCHMANN F. [et al.]** Pattern-oriented software architecture: a system of patterns. - 1996.

**CHIANG T. C.** Product and service testing methodology and ISO 9000 [Periódico]. - 1994.

**DAMIANI FILIPE BIANCHI** Ferramenta de Mapeamento de UIDs para JSF. - Florianópolis : [s.n.], 2011.

**JSF** [Online] // JavaServer Faces. - Oracle, 2013. - 3 de Junho de 2013. - http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html.

**MARAFON Diego Luiz** Integração JavaServer Faces e Ajax, Estudo da integração das tecnologias JSF e Ajax. - UFSC/Brasil : [s.n.], 2006.

**MYERS Glenford J.** The Art of Software Testing [Livro]. - [s.l.] : John Wiley & Sons, Inc., 2004. - Vol. 2.

**Pradhan Ligaj** User Interface Test Automation and its Challenges in an Industrial Scenario. - Mälardalen UniversitY - Sweden : [s.n.], 2011.

**SELENIUM** [Online] // Selenium. - Sauce Labs, 2013. - 3 de Junho de 2013. - http://code.google.com/p/selenium/wiki/PageObjects.

**VILAIN Patrícia** Modelagem da Interação com o Usuário em Aplicações. - Puc-Rio : [s.n.], 2002..

**VILAIN Patrícia** Relatório Final, Funpesquisa 2003. Implementação de um . - UFSC/Brasil : [s.n.], 2003.