
Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

# Validação de uma Abordagem para Testar o Comportamento Excepcional

André Estevão Beltrão Chagas

## TRABALHO DE GRADUAÇÃO

Recife, 11 de Dezembro de 2011

Universidade Federal de Pernambuco
Centro de Informática

# Validação de uma Abordagem para Testar o Comportamento Excepcional

André Estevão Beltrão Chagas

*Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.*

*Orientador: Fernando José Castor de Lima Filho*

Recife, 11 de Dezembro de 2011

Dedico este trabalho à Deus, minha Família e minha namorada.

**Agradecimentos**

Gostaria de agradecer primeiramente a Deus por estar comigo em todos os momentos, me ajudando e fortalecendo. Gostaria também de agradecer aos meus pais, Josenildo Chagas e Shirley Mary de Oliveira Beltrão Chagas que sempre me apoiaram e me motivaram no decorrer do curso. Além desses, estendo meus agradecimentos aos meus irmãos Daniel Filipe Beltrão Chagas e Shirleyne Mary Beltrão Chagas pela companhia e auxílio, assim como agradeço a minha querida namorada Íris Víegas que esteve ao meu lado me dando força e conselhos. Agradeço a Rafael Brito que me deu suporte esclarecendo eventuais dúvidas na realização deste trabalho. Gostaria de agradecer ao professor Fernando Castor que me orientou com destreza e sabedoria. E aos amigos e companheiros do CIn com quem convivi todos esses anos de graduação.

**Resumo**

Robustez e confiabilidade são qualidades de uma boa aplicação. Porém para o sistema ser robusto e confiável é necessário que eles estejam preparados para situações excepcionais.

Por isso, é necessário que quando acionado o comportamento excepcional funcione conforme o especificado. Para garantir a boa funcionalidade do comportamento excepcional é preciso testá-lo.

Para testar o comportamento excepcional foi criada uma abordagem leve e ágil. Este trabalho irá validar a abordagem de teste do comportamento excepcional. Serão mostrados os conceitos, os passos a serem seguidos para testar a aplicação, a ferramenta de apoio desta abordagem. Para validação será aplicada a abordagem em um sistema real e, mostrados os resultados obtidos com a abordagem.

**Palavras-Chave:** Validação; Abordagem; Ágil; Leve; Comportamento, Excepcional.

**Abstract**

Robustness and reliability are qualities of a good application. But to be a robust and reliable system it is required that they be prepared for exceptional situations.

Therefore it is necessary that when an exceptional behavior occurs, the system keep working as specified. To ensure the good functionality of the exceptional behavior, testing it is necessary.

To test the exceptional behavior, a light and agile approach was created. This work will validate the approach of testing exceptional behavior. It will show the concepts, the steps to be followed for testing the application and the support tool for this approach. To validate, the approach will be applied in a real system and the results obtained with the approach will be shown.

**Keywords:** validation, approach, agile, light, behavior, exceptional.

**Sumário**

# 1. Introdução

## 1.1. Contexto e Motivação

O comportamento excepcional traz mais robustez à aplicação [1] e, para tornar a criação do fluxo excepcional mais fácil, as linguagens de programação mais modernas separam os dois fluxos [2]: o normal e o excepcional. Desta forma, tornou-se uma forma explícita para o desenvolvedor o bloco de código que trataria a condição excepcional.

Apesar de todos os esforços é difícil prever o comportamento do fluxo excepcional e o único modo de analisar o comportamento excepcional é executando o trecho do código excepcional. No entanto, executar o fluxo excepcional requer habilidade visto que é necessário inserir situações de erro no programa, portanto seria mais fácil alterar o programa para entrar no fluxo excepcional do que criar caso de teste.

A mudança de versão pode ocasionar a alteração da especificação devido a modificação de métodos, nomes de atributos, herança, polimorfismo e outros casos.

Devidos a esses fatos foi proposta uma abordagem por Rafael Brito leve para testar o comportamento excepcional [3]. Com essa abordagem é possível testar o comportamento excepcional através da comparação dos caminhos. O caminho percorrido pela exceção é obtido em tempo de execução e assim é possível acompanhar o percurso da exceção desde o momento em que foi lançada até o momento da sua captura. Com a especificação é gerado o caminho esperado que a exceção percorresse. Os documentos gerados pela execução dos testes ajudam a manter a especificação do sistema sempre atualizada a cada mudança de versão. Em sua parte automatizada, a abordagem é composta por uma ferramenta. A ferramenta desenvolvida para dar suporte a abordagem é uma extensão do framework JUnit. Para acompanhar a exceção em tempo de execução foi utilizada programação orientada a aspectos [4].

Para avaliação da abordagem proposta em sua dissertação, Rafael Brito usou os sistemas jEdit, aTunes e o Health Watcher. Com a aplicação da abordagem nos sistemas foram encontrados ao todo 19 erros.

Esta abordagem foi testada com três sistemas. Este trabalho fará um estudo de caso mais detalhado da abordagem com aplicação em um sistema real.

## 1.2. Objetivos

Este trabalho tem como propósito validar a abordagem proposta para teste do comportamento excepcional. Para isso será criado um estudo de caso seguindo os passos da abordagem para observar os resultados obtidos pela abordagem bem como sua eficiência.

Os resultados obtidos com a aplicação da abordagem em um sistema real serão apresentados neste trabalho.

## 1.3. Estrutura

Além deste capítulo de introdução, este trabalho contém mais quatro capítulos que estão organizados na seguinte maneira:

No Capítulo 2 serão apresentados conceitos básicos de engenharia de software para o entendimento do trabalho e o framework JUnit.

No Capítulo 3 será apresentada a abordagem para o teste do comportamento excepcional. Este capítulo detalhará a abordagem em fases e, mostrará o arcabouço responsável pela a automatização dos testes.

O Capítulo 4 mostrará a aplicação da abordagem em um programa real, o estudo de caso apresentará o sistema utilizado para validar a abordagem e os casos de teste criados para testar o comportamento excepcional.

O último capítulo será composto da conclusão do trabalho. Neste capítulo será feito a conclusão sobre a validação da abordagem.

## 1.4. Trabalhos Relacionados

Este trabalho testa a abordagem apresentada na dissertação relacionada:

- Uma Abordagem Leve para testar o comportamento excepcional [3]: Nessa dissertação é apresentada a abordagem e avaliada com três sistemas: aTunes, jEdit e o Health Watcher. Para validar essa abordagem será feita nesta monografia um estudo de caso mais detalhado com um sistema real.

# 2. Fundamentos

Este capítulo foi criado para mostrar conhecimentos que são necessários para a continuidade deste projeto. Para isso, serão apresentados fundamentos de tolerância de falhas e teste de softwares, tratamento de exceções, teste de software e o framework JUnit.

## 2.1. Tolerância a falhas

Para Avizienis (1967) "Um sistema é tolerante a falhas se seus programas podem ser executados corretamente, apesar da ocorrência de falhas de lógica". Sistemas que exigem confiabilidade precisam continuar executando corretamente mesmo após a ocorrência de falhas [5]. Existem técnicas de tolerância de falhas para que o sistema não pare após a ocorrência da falha.

### 2.1.1. Falhas, erros e defeitos

Defeito (failure) é um desvio da especificação do projeto. Sistemas não devem apresentar defeitos. Um sistema está em estado de erro quando o seu próximo estado conduzir a um defeito, ou seja, quando o seu processamento posterior a partir do estado atual ocasionar um defeito. Falha (fault) como uma causa física ou algorítmica do erro.

### 2.1.2. Falhas e suas classificações

Falhas possuem geralmente duas classificações [6]:

Falhas Físicas: São aquelas causadas por componentes físicos, tais como: componentes defeituosos, variações ambientais e, distúrbios externos como radiação e interferência eletromagnética.

Falhas Humanas: São de dois tipos. Falhas de interação e falhas de projeto. O primeiro tipo engloba as violações de procedimento e manutenção e o segundo problemas de especificação e implementação.

Um sistema tolerante a falhas deve está atento para erros provocados por interações humanas que visam propositalmente provocar o erro.

### 2.1.3. Técnicas de tolerância a falhas

Nenhum sistema está totalmente livre de falhas, existem técnicas de tolerância delas. Vários autores classificam essas técnicas e, a mais conhecida é a que coloca em quatro fases de aplicação. São elas: detecção, confinamento, recuperação e tratamento.

**Detecção**

Uma falha se manifesta primeiramente como um erro para depois ser detectada. Antes da manifestação do erro a falha está latente e não pode ser detectada. Desta forma, a falha pode estar contida em um sistema por toda sua vida útil sem se manifestar.

**Confinamento**

Por enquanto que a falha não se manifesta ela pode espalhar dados errôneos pelo sistema. A técnica de confinamento estabelece limites para a propagação dos dados errôneos, porém, seus limites dependem do projetista. Durante o projeto devem ser previstas e implementadas, restrições ao fluxo de informações para evitar fluxos acidentais e estabelecer interfaces de verificação para detecção de erros.

**Recuperação**

A recuperação ocorre assim que o erro é detectado. A recuperação é a substituição do estado atual para um estado livre de erros. A recuperação pode ocorrer de duas formas:

Recuperação por retrocesso: É a condução do sistema para um estado anterior já ocorrido. Esse tipo de recuperação é especifica para cada sistema.

Recuperação por avanço: Condução a um estado novo ainda não ocorrido. É uma recuperação de alto custo, porém genérica.

**Tratamento**

O tratamento consiste em localizar a origem do erro (falha), localizá-la precisamente, repará-la e recuperar o restante do sistema. A localização da falha é feita em duas etapas: A localização grosseira (módulo ou subsistema) e localização fina onde o componente falho é localizado. Nos dois tipos de localização são usados os testes diagnósticos (comparação dos resultados previstos com os resultados gerados).

- Diagnostico manual é realizado por um operador local ou remoto.
- Diagnostico automático é realizado por componentes livres de falha.

Despois da localização o componente falho é retirado e o sistema é restaurado de formar manual ou automática.

## 2.2. Tratamento de Exceções

O tratamento de exceções permite os programas de capturar e tratar exceções [7]. O tratamento de exceções é utilizado para recuperar o programa de um mau funcionamento que causou a exceção.

No momento em que o método detecta um erro e não possui capacidade de lidar com o erro o ele dispara uma exceção. Se houver um tratador para o tipo de exceção levantada a exceção será capturada e tratada.

## 2.3. Tratamento de Exceções em Java

O tratamento da exceção em Java é iniciado pelo `bloco try`. Dentro do `bloco try` são colocados os códigos que podem gerar uma exceção e também os códigos que não devem ser executados após o levantamento da exceção. Após ser levantada no `bloco try` a exceção é capturada no `bloco catch`. Em cada `bloco catch` é especificado o tipo da exceção capturada e dentro do bloco é desenvolvido o tratamento para a exceção. O último bloco de tratamento de exceções é o `finally`. A utilização do `finally` é opcional e, sua execução é independente da ocorrência de exceção. A estrutura do tratamento de exceções em Java é a seguinte:

```
Try{

//Codigos que podem gerar exceção.

//Codigos que não devem ser executados após o levantamento
//da excecao

}Catch (TipoDeExcecao referencia){

//Tratamento da exceção.
```

```
}Finally{

// Código executado mesmo sem ocorrência de exceção.

}
```

O funcionamento do tratamento da exceção ocorre da seguinte maneira: após uma exceção ser levantada no `bloco try` o controle de programa sai do bloco e vai para o primeiro `bloco catch`. O programa procura o tratador apropriado para o tipo de exceção levantada passando por cada `bloco catch` na ordem implementada. Ao encontrar o tratador adequado o `bloco catch` correspondente é executado. Se houver um `bloco finally` após o `bloco catch`, ele é executado, e o controle volta à execução normal do programa.

## 2.4. Testes de software

Não é possível garantir o funcionamento de um software sem erros, no entanto é possível aplicar testes para ver se o mesmo adere as suas especificações. O seu objetivo é encontrar as falhas e corrigi-las imediatamente antes da entrega do software para o usuário final. Esta área da engenharia de software visa aumentar a confiabilidade do produto.

### 2.4.1. Estratégia de teste

Existem quatro tipos de testes que são: teste de unidade, teste de componentes, teste de integração e teste de sistemas [8].

**Teste de Unidade**

A unidade é a menor parte de um software. O seu teste se limita a avaliar uma porção de código, por isso, normalmente é preciso à utilização de stubs e drivers. Os casos de testes são aplicados nos drivers, os drivers são responsáveis por colocar dados na entrada e mostrar aos usuários os dados da saída. O stub é a parte que simula o comportamento dos componentes que faltam.

**Teste de Componente**

Componentes são interfaces com um conjunto de funcionalidades. Seu teste utiliza stubs e drivers.

**Teste de Integração**

A fase de integração é a junção dos componentes. Os testes de integração são executados para encontrar os erros decorrentes da junção de módulos do sistema. Eles devem ser feitos da forma incremental, ou seja, suas unidades devem ser incrementadas de pequenos segmentos aplicando seus testes. Os principais tipos de teste de integração são o bottom-up e top down.

- Bottom-up: São integrados os componentes de alto nível primeiro até os de nível mais baixo. Os stubs são responsáveis por representar os componentes de nível mais baixo nos testes.
- Top-Down: São integrados os componentes de nível mais baixo subindo no nível de hierarquia dos componentes até o último ser testado.

**Teste de Sistemas**

E por último o teste do sistema. Esse teste é aplicado antes da entrega do produto ao usuário final. Esse teste é feito em condições similares ao do usuário onde são percorridas as funcionalidades do sistema em busca de erro.

## 2.5. JUnit

É um framework de testes automatizados criado por Eric Gamma e Kent Beck [9]. É uma ferramenta amplamente aceita em projetos open source e também em softwares comerciais. Com esta ferramenta é possível testar 30 linguagens de programação diferentes. Os teste criados para rodar em JUnit são chamados de caso de teste.

No JUnit 4, podemos utilizar a anotação `@Test`[10] para criar um caso de teste para o método responsável por testar uma parte do código. Como no teste abaixo:

```
public class MeuTeste {

   @Test

     public void deposito(){
```

```
        Conta depositada = new Conta(15);
        Conta total = new Conta(30);
        conta.depositar(15);
           assertTrue (total.equals(conta));
        }
}
```

O teste verifica se o valor das contas depositada e total são iguais.

O JUnit também oferece a possibilidade de agrupar um conjunto de teste. O método responsável é o `runClasses` da classe `JUnitCore` seu parâmetro são as classes de teste como é mostrado abaixo:

```
org.junit.runner.JUnitCore.runClasses(TestClass1.class,
TestClass2.class,…);
```

É possível que seja preciso inicializar alguma variável ou executar um bloco de código antes da execução do caso de teste para isso o framework possui a anotação `@Before`. Para executar blocos após a execução dos testes existe a anotação `@After`. Como mostrado abaixo:

```
public class MeuTeste {

   @Before
     public void inicializar(){
        Conta conta = new Conta(15);
        Conta total = new Conta(30);
     }
```

```
    @Test
      public void deposito(){
         Conta conta = new Conta(15);
         Conta total = new Conta(30);
         conta.depositar(15);
            assertTrue (total.equals(conta));
         }

    @After
         public void zerar(){
         conta.zerar();
         total.zerar();
      }
}
```

Antes da execução do caso de teste é executado o método inicializar e após a execução do método é executado o método zerar.

# 3. Abordagem proposta para testar o comportamento excepcional

O comportamento excepcional foi criado para tornar mais robusto o comportamento normal. Se solicitado, é esperado que o comportamento excepcional execute sem erros para levar o sistema de um estado errôneo para um estado livre de erros. Por ser uma parte do código pouco executada dificulta ainda mais o seu teste.

Em sua dissertação Rafael Brito propõe uma abordagem leve para testar o comportamento excepcional. Com ela é possível especificar o caminho esperado que a exceção percorra e o caminho percorrido pela exceção em tempo de execução desde o seu lançamento até o último método ao qual a exceção foi propagada. A comparação destes caminhos indica o resultado do teste. A figura 1 ilustra as etapas desta abordagem. Na figura, os retângulos arredondados são as atividades, os losangos representam as alternativas e, as caixas restantes são os artefatos. As linhas contínuas representa a ordenação das atividades e as linhas tracejadas indicam a dependência dos fatos. Esta abordagem será detalhada neste capítulo.

Os três pontos fundamentais da abordagem são: especificação, monitoramento e verificação do comportamento excepcional.

Na fase de especificação será definido o caminho que as exceções devem percorrer. Depois de definidos os caminhos segue-se para a monitoração dos comportamentos excepcionais através da execução dos casos de teste e, finalmente, os resultados da especificação e o resultado da monitoração são comparados.

## 3.1. Especificação

Na fase de especificação são definidos os caminhos esperados e os casos de testes. É custoso testar todos os fluxos excepcionais em busca de erro em sistemas de grande porte, porém para diminuir os custos é possível analisar características que criam maiores oportunidades de falhas de projetos serem introduzidas no sistema [11].

As características são seguintes:

- Blocos que não levantam exceção;

- Blocos que re-mapeiam exceções mais genéricas, ou seja, a exceção mais genérica é capturada no bloco e outra exceção é levantada para propagação;
- Blocos que levantam diferentes tipos de exceção;
- Blocos que pegam mais de um tipo de exceção.

Após a escolha dos caminhos são criados os casos de teste. Nesta atividade são relatados os caminhos esperados para cada caso de teste: qual exceção é levantada, a classe que levantou a exceção, as classes intermediárias e por último a classe que trata a exceção.

## 3.2. Monitoração das exceções

Nesta fase são reunidos todos os casos de teste e agrupados de acordo com suas características em suítes de teste. As suítes de teste são executadas e, os caminhos gerados por cada exceção devem ser guardados para a próxima atividade.

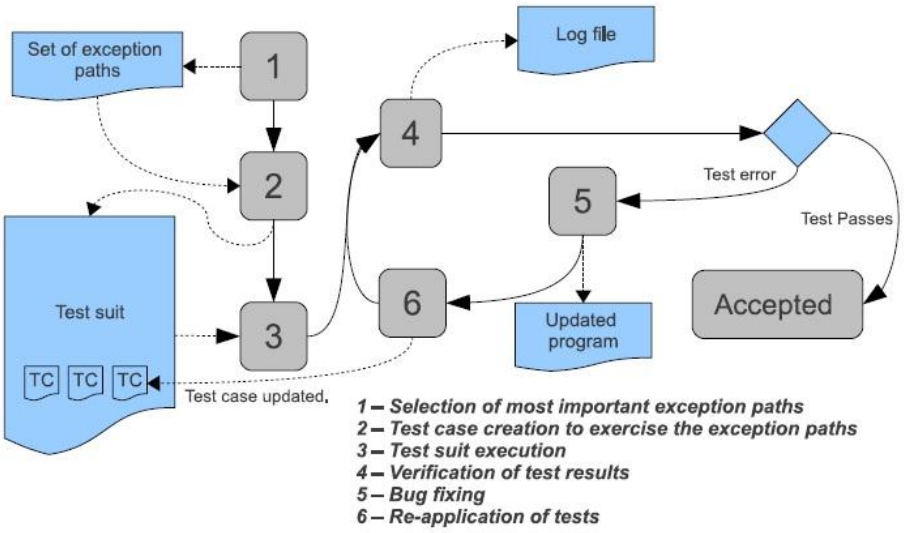


*Figura 1 - Esquema Geral da Abordagem*

## 3.3. Comparação dos caminhos gerados

Com os caminhos criados nas fases anteriores, ambos serão comparados e se todos passarem, então, o comportamento excepcional está agregado a sua especificação. Caso contrário foi encontrado inconsistência. As inconsistências podem ser de dois

tipos: por manutenção ou fluxos incompatíveis. Inconsistência por manutenção é devido alteração em nome de código, nome de classe, adição de método. Essas incoerências não são consideradas erros. Os erros são as divergências do fluxo excepcional.

Os testes que não passaram por motivo de alteração no projeto terão seus testes refeitos executados novamente. Nas situações em que houve incoerência de comportamento o componente deve ser corrigido e aplicado os testes novamente para a comparação dos comportamentos.

## 3.4. JUnitE

Para dar suporte à abordagem proposta foi desenvolvida a ferramenta JUnitE. Esta ferramenta é uma extensão do framework JUnit. O JUnitE é responsável por executar os testes, monitorar as exceções, criar os logs com os caminhos percorridos e comparar os resultados.

### 3.4.1. Monitoramento de Exceções

Para monitorar as exceções foi utilizada programação orientada a aspectos. Esta parte da ferramenta é responsável por fornecer informações como: exceção levantada, métodos percorridos, método que capturou a exceção, se houve re-mapeamento e para qual exceção a exceção originalmente lançada foi re-mapeada.

### 3.4.2. Logs de saída dos casos de teste

Após a execução dos testes, criados na fase de especificação, são fabricados logs automaticamente pelo arcabouço JUnitE. Esses logs são os caminhos percorridos pela exceção em tempo de execução e, o conhecimento do conteúdo do log de saída é importante para acompanhar a mudança de versão.

O exemplo a seguir é um caminho percorrido pela exceção em um dos testes do estudo de caso:

```
#com.eteks.sweethome3d.model.RecorderException

com.eteks.sweethome3d.io.FileUserPreferences.writeCatalog

com.eteks.sweethome3d.io.FileUserPreferences.write
```

**@com.eteks.sweethome3d.swing.HomeController.updateUserPreferencesRecentHomes**

A exceção levantada vem com o símbolo # antes do nome e, logo após vem o método que lançou a exceção. No caso o nome da exceção é `com.eteks.sweethome3d.model.RecorderException` e o nome do método lançador é `com.eteks.sweethome3d.io.FileUserPreferences.writeCatalog`. O nome do método que captura e trata a exceção é antecedido pelo símbolo @ consequentemente, `com.eteks.sweethome3d.swing.HomeController.updateUserPreferencesRecentHomes` é o nome do método que captura a exceção. Entre o método que lança a exceção e o método que captura estão os métodos intermediários.

### 3.4.3. Comparar resultados

O JunitE oferece suporte para a comparação dos caminhos. Para a comparação é preciso descrever o caminho esperado da seguinte maneira. O caminho é um array de string, com exceção levantada, método lançador, métodos intermediários e o método capturador que são invocados em Java pelos respectivos métodos `exception`, `raiseSite`, `intermediateSite` e `handlingSite`.

Para demonstrar como fazer a comparação de caminhos foi retirado do estudo de caso o exemplo a seguir:

```
1-@Test
2-public void testando(){
3-          String[] trace = new String[]{
4-           exception("java.awt.print.PrinterException"),
5-          raiseSite("com.eteks.sweethome3d.swing.
HomePrintableComponent.print"),
6-          raiseSite("com.eteks.sweethome3d.swing.
HomePrintableComponent.getPageCount),
```

```
7-          catchSite("com.eteks.sweethome3d.test.
MeuTeste.testando")};
8-          setExceptionPath(trace);}
```

No exemplo acima o caminho esperado é especificado para testar o comportamento excepcional do sistema no momento em que são impressas as características em pdf dos objetos inseridos na casa projetada pelo sistema. Na linha 1 temos a anotação do JUnit para teste, na linha 3 temos a criação do caminho. Dentro do caminho temos a exceção que é passada na linha 3 temos o método lançador na linha 5 e na linha 6 temos o método intermediário por ultimo temos na linha 7 o método capturador da exceção. Na linha 8 o array com o caminho é passado como parâmetro para o método `setExceptionPath`.

### 3.4.4. Forçar Exceções

Criar situações de erro é uma tarefa bastante difícil e de custo elevado. Com o intuito de melhorar essa situação foi criada no JUnitE a anotação `@ForceException`. Essa anotação permite forçar o lançamento de exceções sem ser preciso modificar o projeto. Para forçar a exceção existe um pré-processador que verifica todas as anotações `@ForceException` do caso de teste e gera automaticamente o aspecto responsável por forçar a exceção.

O exemplo foi retirado do estudo de caso e está forçando o levantamento da exceção no momento da impressão das características dos objetos incluídos na casa projetada pelo sistema:

```
1-@ForceException(
2    exception="java.awt.print.PrinterException",
3-method=
"com.eteks.sweethome3d.swing.HomePrintableComponent.print"
4-   methodParType="java.awt.Graphics,
java.awt.print.PageFormat, int",
5-   methodReturnType="int")
```

Na linha 1 é passada a anotação para `@ForceException` do JUnitE na linha 1. Como parâmetros da anotação são passados a exceção a ser lançada na linha 2, o método lançador como na linha 3, os parâmetros do método como na linha 4 e o tipo do retorno do método na linha 5.

# 4. Estudo de Caso

Sistemas com tratamento de exceções são mais robustos. O tratamento de exceções é responsável de capturar a exceção e tratá-la, porém o tratamento de exceções do sistema pode não estar funcionando conforme o especificado.

Uma abordagem para testar se o comportamento excepcional de um sistema funciona conforme o especificado foi proposta por Rafael Brito e, testada com apenas três sistemas reais este capítulo criará um estudo de caso com um sistema real para validar mais detalhadamente a abordagem. Os casos de teste criados para validar a abordagem e a tabela com todos os resultados serão apresentados.

## 4.1. Sweet Home 3D

O sistema escolhido foi o Sweet Home 3D, desenvolvido por Emmanuel Puybarete na linguagem Java. O aplicativo é open source e está atualmente na versão 3.3. Ao todo foram desenvolvidas 37 versões.

O programa foi criado para projetar o interior de uma casa em terceira dimensão. É possível delimitar áreas desenhando cômodos e paredes. Para não exigir do usuário técnicas de desenho o software dispõe de um catalogo com portas, janelas e mobiliários já incluídos que só basta serem arrastados para serem incluídos na casa virtual. Os objetos do catálogo ainda podem ter seu tamanho, cor, textura e outros atributos alterados.

Sua principal característica é ver em qualquer ponto de vista o objeto simultaneamente em terceira dimensão contemplando o usuário com a sensação mais realista, além de criar imagens e vídeos com várias fontes de luz.

Por fim, existe uma funcionalidade que oferece ao usuário incluir mais objetos em terceira dimensão para o projeto, em sua página oficial é possível baixar novos modelos incluído por contribuintes.

Para a escolha do aplicativo foram levados em conta os seguintes atributos: quantidade de versões e quantidade de linhas de código. A quantidade de versões é importante para ser possível testar varias vezes o mesmo fluxo excepcional. A

consequência de testar várias versões é a modificação que sofre o código testado essa alteração tanto pode consertar um erro no bloco quanto pode trazer um erro inesperado.

É importante para mensurar o tamanho do projeto. Para descobrir a quantidade de linhas foi utilizada a ferramenta open source Metrics e, na primeira versão testada são ao todo 20.702 linhas de código já na sua ultima versão testada são 64.969 linhas código.

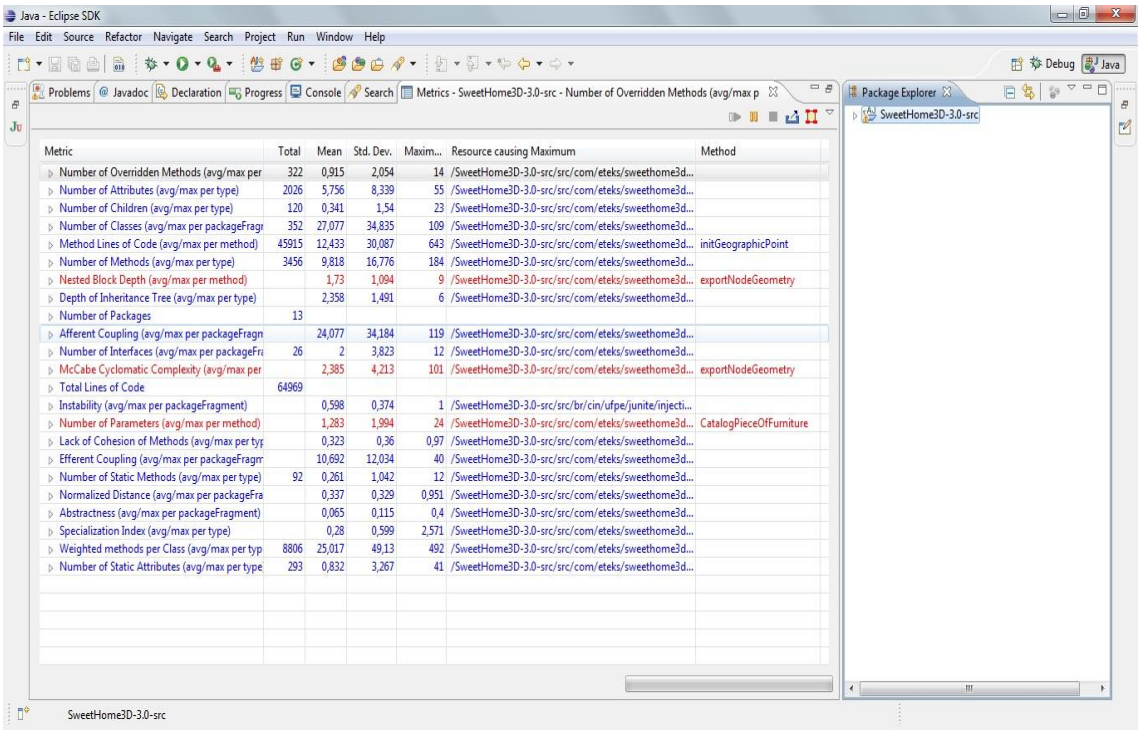


*Figura 2 - Métricas Sweet Home 3D*

## 4.2. Casos de Teste

Antes da criação dos casos de teste foram especificados os caminhos esperados a partir da versão 1.0 do Sweet Home 3D. Após a escolha da versão inicial foram escolhidos os comportamentos excepcionais que possuíam características de maior ocorrência de erro, conforme a seção 3.1, e posteriormente criados os casos de teste. Ao todo são 10 casos de teste a seguir e serão mostradas quais características pertence a cada fluxo, seus caminhos esperados e percorridos.

### 4.2.1. Teste 01

O teste 01 é do tipo que abrange uma exceção mais genérica e levanta outra para a sua propagação. Ao tentar salvar um arquivo, era possível que fosse levantado a exceção de entrada e saída e essa exceção seria mapeada para outra exceção.

Portanto a sua saída esperada era:

```
#java.io.IOException

com.eteks.sweethome3d.io.HomeFileRecorder$HomeOutputStream.
writeHome

@com.eteks.sweethome3d.io.HomeFileRecorder.writeHome

#com.eteks.sweethome3d.model.RecorderException

com.eteks.sweethome3d.io.HomeFileRecorder.writeHome

@com.eteks.sweethome3d.swing.HomeController.save
```

Através dessa saída podemos verificar que a exceção originária foi o `java.io.IOException` seu método lançador foi o `writeHome` da classe `HomeOutputStream`. Na classe `HomeFileRecorder` no método `writeHome` a exceção propagada deixa de ser `java.io.IOException` e é levantada outra em exceção o `com.eteks.sweethome3d.model.RecorderException`.

Após a alteração na versão 1.4 do Sweet Home o `bloco try catch` deixou de ser somente do tipo de re-mapeamento e passou a ser também do tipo que capturam mais de um tipo de exceção. A nova exceção adicionada no `bloco try-catch` deu origem ao caso de teste 02.

A alteração no `bloco try catch` provocou uma mudança no seu caminho percorrido pela exceção que passou a ser a seguinte.

```
#java.io.IOException

com.eteks.sweethome3d.io.HomeFileRecorder$HomeOutputStream.
writeHome

@com.eteks.sweethome3d.io.HomeFileRecorder.writeHome
```

**#java.util.concurrent.ExecutionException**

**java.util.concurrent.FutureTask$Sync.innerGet**

**java.util.concurrent.FutureTask.get**

**@com.eteks.sweethome3d.swing.ThreadedTaskController$1.done**

Após a versão 1.4 não houve mudanças no código do método `writeHome` e essa saída permaneceu até o fim dos testes.

### 4.2.2. Teste 02

Esse caso de teste foi originado devido alteração no comportamento excepcional do caso de teste anterior então o mesmo não se encontra desde a primeira versão testada. O teste 02 foi incluído a partir da versão 1.4 para verificar o caminho percorrido da exceção `java.io.InterruptedIOException`. Esta exceção é levantada se a thread do sistema for interrompida. A verificação de interrupção da thread é executa no inicio do método `writeHome`.

O seu caminho foi o seguinte:

**#java.io.InterruptedIOException**

**com.eteks.sweethome3d.io.HomeFileRecorder.checkCurrentThreadIsntInterrupted**

**com.eteks.sweethome3d.io.HomeFileRecorder.access$0**

**com.eteks.sweethome3d.io.HomeFileRecorder$HomeOutputStream.writeHome**

**@com.eteks.sweethome3d.io.HomeFileRecorder.writeHome**

**#java.util.concurrent.ExecutionException**

**java.util.concurrent.FutureTask$Sync.innerGet**

**java.util.concurrent.FutureTask.get**

**@com.eteks.sweethome3d.swing.ThreadedTaskController$1.done**

Após passar pelo writeHome a exceção levantada foi a mesma do teste 01 continuando assim até a ultima versão testada.

### 4.2.3. Teste 03

Para abrir os arquivos salvos no formato do sistema é executado o comportamento excepcional do teste 03. Esse caso de teste tem característica de blocos que pegam mais de um tipo de exceção e, que capturam uma exceção mais genérica e propagam outra exceção. Ao chegar ao método `readHome` da classe `HomeFileRecorder` era esperado que a exceção `java.io.IOException` fosse capturada e levantasse a exceção com.eteks.sweethome3d.model.RecorderException.

O seu caminho esperado era o seguinte:

**#java.io.IOException**

**com.eteks.sweethome3d.io.HomeFileRecorder$HomeInputStream.readHome**

**@com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**#com.eteks.sweethome3d.model.RecorderException**

**com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**@com.eteks.sweethome3d.swing.HomeController.open**

Após a versão 1.4 do Sweet Home foi alterado o método `readHome` e houve a inclusão de mais uma exceção no `bloco try-catch` a `java.io.InterruptedIOException`. Os caminhos permaneceram os mesmos.

Houve mudanças na versão 1.5 e o caminho percorrido da exceção mudou para:

**#java.io.IOException**

**com.eteks.sweethome3d.io.DefaultHomeInputStream.readHome**

**@com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**#java.util.concurrent.ExecutionException**

**java.util.concurrent.FutureTask$Sync.innerGet**

**java.util.concurrent.FutureTask.get**

**@com.eteks.sweethome3d.viewcontroller.ThreadedTaskController$1.done**

### 4.2.4. Teste 04

O caso de teste 04 faz parte do mesmo `bloco try-catch` anterior. As características desse caso de teste também são as mesmas do caso anterior: mais de uma exceção era capturada. Era esperado que o bloco capturasse a exceção `java.lang.ClassNotFoundException` e propagasse a exceção `com.eteks.sweethome3d.model.RecorderException`.

O caminho esperado dessa exceção era:

**#java.lang.ClassNotFoundException**

**com.eteks.sweethome3d.io.HomeFileRecorder$HomeInputStream.readHome**

**@com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**#com.eteks.sweethome3d.model.RecorderException**

**com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**@com.eteks.sweethome3d.swing.HomeController.open**

Porém após as alterações no método na versão 1.5 o caminho passou a ser o seguinte:

**#java.lang.ClassNotFoundException**

**com.eteks.sweethome3d.io.DefaultHomeInputStream.readHome**

**@com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**#java.util.concurrent.ExecutionException**

**java.util.concurrent.FutureTask$Sync.innerGet**

**java.util.concurrent.FutureTask.get**

**@com.eteks.sweethome3d.viewcontroller.ThreadedTaskController$1.done**

### 4.2.5. Teste 05

Este caso de teste está no mesmo `bloco try-catch` do caso de teste 03 suas características são as mesmas do caso de teste 03 e 04. Esse caso de teste foi criado a partir da versão 1.4 após a inclusão da captura de uma exceção no `bloco try-catch`.

O seu caminho foi o seguinte:

**#java.io.InterruptedIOException**

**com.eteks.sweethome3d.io.DefaultHomeInputStream.checkCurrentThreadIsntInterrupted**

**com.eteks.sweethome3d.io.DefaultHomeInputStream.readHome**

**@com.eteks.sweethome3d.io.HomeFileRecorder.readHome**

**#java.util.concurrent.ExecutionException**

**java.util.concurrent.FutureTask$Sync.innerGet**

**java.util.concurrent.FutureTask.get**

**@com.eteks.sweethome3d.viewcontroller.ThreadedTaskController$1.done**

### 4.2.6. Teste 06

Antes de salvar ou abrir um arquivo é feita uma verificação para saber se o nome passado para o arquivo já existe na pasta de destino. A verificação da existência do nome possui um bloco try catch que tem como característica não levantar a exceção.

### 4.2.7. Teste 07

Todos os objetos do catalogo possui um ícone para sua representação esse `bloco try-catch` testado é executado no momento de carregar o ícone se não for possível carregar o ícone é mostrado um ícone de erro. O teste é executado em um bloco que não

levanta exceção. Ao chegar a exceção `java.io.IOException` na classe `IconManager` e no método `createIcon` a sua propagação termina.

### 4.2.8. Teste 08

No momento de salvar ou abrir o arquivo para saber qual pasta o sistema está instalado é aberto um arquivo com as preferencias do usuário. O comportamento excepcional para tratar abertura do arquivo de preferencias possui a característica de capturar uma exceção e propagar outra. A exceção `java.io.IOException` é levantada no método lançador `getApplicationFolder` da classe `FileUserPreferences` e, ao chegar no método `writeCatalog` muda a exceção propagada para `com.eteks.sweethome3d.model.RecorderException`.

Na versão 1.4 o caminho percorrido pela exceção mudou devido à alteração de manutenção então seguindo a abordagem proposta o teste foi refeito e executado novamente, ou seja, o seu caminho esperado foi alterado e após executar o teste o caminho percorrido foi igual já que foi incluído mais um método intermediário. Já na versão 1.5 e 1.6 mudou o caminho percorrido devido à manutenção seguindo os mesmo passos anteriores.

### 4.2.9. Teste 09

Cada janela que aparece na ajuda é um html. No momento em que se abre a janela de ajuda ou troca-se de janela é carregado um arquivo html. Existe um tratamento de exceção para a abertura desse arquivo. O tratamento de exceção possui a característica de capturar uma exceção genérica e propagar outra exceção. Porém no teste não é possível verificar isto já que não é possível forçar exceções em métodos pertencentes a bibliotecas. O caso de teste levanta a exceção `java.lang.RuntimeException` dentro do método `setPage` da classe `HelpPage`.

### 4.2.10. Teste 10

O Sweet Home 3D oferece a opção de imprimir em PDF as lista de objetos inseridos na casa projetada pelo programa. Existe um tratamento de exceção para o momento da impressão da lista. O tratamento de exceção possui características de capturar uma exceção mais genérica e propagar outra. Na Classe

`HomePrintableComponent` no método `print` é levantado a exceção `java.awt.print.PrinterException` que posteriormente no método `getPageCount` da mesma classe do método de origem será capturada e no seu lugar será propagada a exceção `java.lang.RuntimeException`.

## 4.3. Tabela de Resultado dos Testes

Para visualizar os resultados dos casos de teste foi criada uma tabela com os testes e as versões testadas. Para os casos de teste que obtiveram sucesso foi colocado o valor PASSOU, e para os testes que não foram existiam na versão o seu valor atribuído foi INDEFINIDA e, para os casos de teste que passaram mas que por alguma eventualidade sofreram alteração no seu caminho o seu valor PASSOU-CAMINHO. Os casos de teste que não passaram obtiveram o valor de NÃO PASSOU.

| | 1.0 | 1.1. | 1.2 | 1.2.1 | 1.3 | 1.4 | 1.5 | 1.5.1 | 1.6 | 1.7 | 1.8 | 2.0 | 2.1 | 2.2 | 2.3 | 2.4 | 2.5 | 2.6 | 3.0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Teste 01 | Passou | Passou | Passou | Passou | Passou | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 02 | Indefinida | Indefinida | Indefinida | Indefinida | Indefinida | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 03 | Passou | Passou | Passou | Passou | Passou | Passou | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 04 | Passou | Passou | Passou | Passou | Passou | Passou | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 05 | Indefinida | Indefinida | Indefinida | Indefinida | Indefinida | Passou | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 06 | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 07 | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 08 | Passou | Passou | Passou | Passou | Passou | Passou-Caminho | Passou-Caminho | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 09 | Passou | Passou | Passou | Passou | Passou | Passou | Passou-Caminho | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |
| Teste 10 | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou | Passou |

Figura 3 Tabela de Resultados

# 5. Conclusão

A abordagem proposta por Rafael Brito auxilia ao desenvolvedor a encontrar divergências entre a especificação e a execução do comportamento excepcional. Desta forma é possível verificar se as diferenças ocorreram por atualização do sistema ou a divergência é algum erro do comportamento excepcional. Os artefatos criados ajudam a documentar os testes realizados em cada versão.

A cargo do desenvolvedor fica somente a tarefa de especificar e criar os casos de teste. O framework é responsável por executar os testes, comparar os resultados, e criar os arquivos de saída com os caminhos percorridos pela exceção facilitando o trabalho de teste do desenvolvedor.

A abordagem mostrou-se bastante eficaz em testar e especificar o comportamento excepcional do sistema.

# 6. Referência


[1] Zhang, Pingyu; Elbaum, Sebastian; Amplifying Tests to Validate Exception Handling Code. Lincoln.

[2] Robillard, Martin P.; Murphy, Gail, C.; Static analysis to support the evolution of exception structure in object oriented systems. ACM Transactions on Software Engineering and Methodology:191-221, 2003.

[3] Bernardo, R. Uma abordagem leve e ágil para testar o comportamento excepcional. 2011. Recife. Dissertação (Mestrado em Ciência da Computação), Centro de Informática-Universidade Federal de Pernambuco, Recife.

[4]- Lee, K. An Introduction to Aspect-Oriented Programming. Hong Kong. 2002.

[5]- Algirdas Avizienis. Toward systematic design of fault-tolerant systems. Computer,

30:51–58, 1997.

[6] Weber, T. S. Tolerância a Falhas: Conceitos e Exemplos. 2006. Programa de Pós Graduação em Computação- Instituto de Informática. Universidade Federal do Rio Grande do Sul.

[7] Deitel, H.; Deitel, P.; Tratamento de Exceções In: Deitel, H.; Deitel, P.; Java Como Programar: Apresentando Projeto orientado a objetos com a UML e Padrões de projeto, Porto Alegre, Bookman, 2004, p 739-767

[8]- Maldonado, J; Barbosa, E; Vincenzi, A; Delamaro M; Souza, S; Jino M; Introdução ao Teste de Software. São Carlos. 2004.

[9]- Rainsberger, J; Stirling S. Fundamentals: What is Programmer Testing? In Rainsberger, J; J; Stirling S. JUnit Recipes: Practical Methods for Programmer Testing.Greenwich, Manning , 2005, p 4-9

[10] Eric Gamma e Kent Beck. Junit. Acessado em Dezembro de 2011. http://junit.sourceforge.net/doc/cookbook/cookbook.htm.

[11] Bernardo, Rafael; Sales, Ricardo, Jr.; Castor, Fernando; Coelho, Roberta; Cacho, Nelio; Soares, Sergio; AgileTesting of Excepcional Behavior. In Proceedings of the 25th Brazilian Symposium on Software Engineering. São Paulo. Setembro 2011.