Pedro Ghilardi

Aplicação da técnica de diagnóstico de defeitos de software em ferramentas de execução de testes

Florianópolis

Junho 2009

Pedro Ghilardi

Aplicação da técnica de diagnóstico de defeitos de software em ferramentas de execução de testes

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Ciências da Computação.

Orientador: Prof. Dr. Sérgio Peters

Co-Orientador: MSc. José Otávio Carlomagno Filho

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Florianópolis

Junho 2009

Monografia de graduação sob o título *"Aplicação da técnica de diagnóstico de defeitos de software em ferramentas de execução de testes"*, defendida por Pedro Ghilardi e aprovada em 02 de junho de 2009, em Florianópolis, Santa Catarina, pela banca examinadora constituída pelos avaliadores:

Prof. Dr. Sérgio Peters
Universidade Federal de Santa Catarina
Orientador

MSc. José Otávio Carlomagno Filho
Co-Orientador

Prof. Dr. Luiz Cláudio Villar dos Santos
Universidade Federal de Santa Catarina
Membro da Banca

Prof. Dr. Ricardo Pereira e Silva
Universidade Federal de Santa Catarina
Membro da Banca

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Abstract

1 Introdução p. 13

2 Programação Orientada a Aspectos p. 16

2.1 Conceitos gerais . . . . p. 17

2.2 Ponto de Junção de Execução . . . . p. 18

2.3 Um exemplo de aspecto . . . . p. 20

2.4 Composição . . . . p. 21

3 Testes p. 23

3.1 Conceitos gerais . . . . p. 23

3.2 Testes Caixa Branca . . . . p. 23

3.3 Testes Caixa Preta . . . . p. 24

3.4 Suítes de casos de teste . . . . p. 24

3.5 Testes automatizados . . . . p. 25

3.6 Ambiente de testes . . . . p. 25

3.6.1 *Test Automation Framework* (TAF) . . . . p. 25

3.6.2 TAFPlus2 . . . . p. 26

3.6.3 TAFStudio . . . . p. 27

4 Metodologia p. 29

4.1 Localização e diagnóstico de faltas pelo perfil de execução . . . . p. 29

4.2 Um exemplo de diagnóstico . . . . p. 31

4.3 Vantagens e Desvantagens . . . . p. 32

5 Análise, Projeto e Implementação da Ferramenta p. 33

5.1 Análise, Projeto e Implementação do aspecto . . . . p. 33

5.2 Análise de blocos monitorados . . . . p. 39

5.3 Análise, Projeto e Implementação do *plug-in* . . . . p. 41

5.4 Composição . . . . p. 46

6 Experimentação e Resultados p. 49

6.1 Criação de Testes . . . . p. 49

6.2 Introdução de Faltas . . . . p. 51

6.3 Execução de Testes . . . . p. 53

6.4 Avaliação . . . . p. 53

6.4.1 Avaliação da execução de uma suíte de testes . . . . p. 53

6.4.2 Coeficiente de Similaridade . . . . p. 56

6.4.3 Suíte de Testes . . . . p. 56

6.4.4 Programa Alvo . . . . p. 57

6.4.5 Granularidade de Blocos . . . . p. 58

6.5 Resultados . . . . p. 60

7 Conclusão p. 63

7.1 Trabalhos Futuros . . . . p. 64

Referências Bibliográficas p. 65

Anexo A – Código dos aspectos p. 67

A.1 Código do aspecto para o TAFStudio . . . . p. 67

A.2 Código do aspecto para o TAFPlus2 . . . . p. 68

Anexo B – Diagramas de seqüência e de classes do *plug-in* p. 70

B.1 Diagrama de Classes . . . . p. 70

B.2 Diagrama de Seqüência: Diagnosticar Casos de Teste . . . . p. 72

B.3 Diagrama de Seqüência: Controlar Rastreamento . . . . p. 77

Anexo C – Código do *plug-in* p. 80

Anexo D – Suítes de testes p. 123

D.1 Suítes de casos de teste manuais do TAFPlus2 . . . . p. 123

D.2 Suítes de casos de teste manuais do TAFStudio . . . . p. 158

Anexo E – Faltas introduzidas, Execuções de Testes e Diagnósticos p. 180

E.1 Faltas introduzidas no TAFStudio e execuções manuais de suítes de testes . . p. 180

E.2 Faltas introduzidas no TAFPlus2 e execuções manuais de suítes de testes . . . p. 184

E.3 Casos particulares na introdução de faltas, execução de testes e emissão de diagnósticos . . . . p. 188

Lista de Figuras

2.1 Área de dispersão . . . . p. 16

2.2 Ponto de junção de execução (*execution join point*) . . . . p. 19

2.3 Codificação de um ponto de junção de execução . . . . p. 19

2.4 Exemplo de aspecto para registrar reservas e liberação de quartos . . . . p. 21

2.5 Composição de aspecto com um programa . . . . p. 22

3.1 Arquitetura do *Test Automation Framework* (TAF) . . . . p. 26

5.1 Caso de uso ⟨ Executar Transação ⟩ . . . . p. 35

5.2 Ponto de extensão modelado em um diagrama de casos de uso . . . . p. 36

5.3 Diagrama de papéis . . . . p. 37

5.4 Diagrama de seqüência do rastreamento de métodos . . . . p. 37

5.5 Diagrama de seqüência do rastreamento de construtores . . . . p. 38

5.6 Fatia de caso de uso . . . . p. 39

5.7 Implementação básica do aspecto de rastreamento . . . . p. 40

5.8 Diagrama Conceitual . . . . p. 42

5.9 Diagrama inicial de casos de uso do *plug-in* . . . . p. 42

5.10 Diagrama de atividades do caso de uso Diagnosticar Casos de Teste . . . . . p. 43

5.11 Diagrama final de casos de uso do *plug-in* . . . . p. 44

5.12 Diagrama de atividades do caso de uso Controlar Rastreamento . . . . p. 44

6.1 Bloco TagConfigEditor.addTag() sem falta . . . . p. 52

6.2 Bloco TagConfigEditor.addTag() com falta . . . . p. 52

6.3 Ciclo de execução manual de suítes de testes . . . . p. 54

6.4 Bloco RecordingSession.pushGroup() com falta . . . . p. 54

6.5 Bloco AddGroupDialog.update() sem falta . . . . p. 59

6.6 Bloco AddGroupDialog.update() com falta . . . . p. 60

6.7 Bloco AddGroupDialog.update() refatorado e com falta . . . . p. 61

6.8 Resultados do TAFPlus2 . . . . p. 62

6.9 Resultados do TAFStudio . . . . p. 62

B.1 Diagrama de classes . . . . p. 71

B.2 Diagrama de sequência Diagnosticar Casos de Teste (1) . . . . p. 73

B.3 Diagrama de sequência Diagnosticar Casos de Teste (2) . . . . p. 74

B.4 Diagrama de sequência Diagnosticar Casos de Teste (3) . . . . p. 75

B.5 Diagrama de sequência Diagnosticar Casos de Teste (4) . . . . p. 76

B.6 Diagrama de sequência Controlar Rastreamento (1) . . . . p. 78

B.7 Diagrama de sequência Controlar Rastreamento (2) . . . . p. 79

Lista de Tabelas

5.1 Especificação do caso de uso ⟨ Executar Transação ⟩ . . . . p. 35

5.2 Especificação do caso de uso Gerenciar Rastreamento de Blocos . . . . p. 35

6.1 Um exemplo de caso de teste manual para o TAFPlus2 . . . . p. 50

6.2 Diagnóstico parcial da execução de uma suíte de testes . . . . p. 55

D.1 Suíte 001 do TAFPlus2: Testes 001 e 002 . . . . p. 124

D.2 Suíte 001 do TAFPlus2: Testes 003 e 004 . . . . p. 125

D.3 Suíte 001 do TAFPlus2: Teste 005 . . . . p. 126

D.4 Suíte 001 do TAFPlus2: Teste 006 . . . . p. 127

D.5 Suíte 001 do TAFPlus2: Testes 007, 008 e 009 . . . . p. 128

D.6 Suíte 002 do TAFPlus2: Teste 010 . . . . p. 129

D.7 Suíte 002 do TAFPlus2: Teste 011 . . . . p. 130

D.8 Suíte 002 do TAFPlus2: Teste 012 . . . . p. 131

D.9 Suíte 002 do TAFPlus2: Testes 013 e 014 . . . . p. 132

D.10 Suíte 002 do TAFPlus2: Testes 015 e 016 . . . . p. 133

D.11 Suíte 003 do TAFPlus2: Testes 030 e 031 . . . . p. 134

D.12 Suíte 003 do TAFPlus2: Testes 032 e 033 . . . . p. 135

D.13 Suíte 003 do TAFPlus2: Testes 034 e 035 . . . . p. 136

D.14 Suíte 003 do TAFPlus2: Teste 036 . . . . p. 137

D.15 Suíte 003 do TAFPlus2: Teste 037 . . . . p. 138

D.16 Suíte 003 do TAFPlus2: Teste 038 . . . . p. 139

D.17 Suíte 003 do TAFPlus2: Teste 039 . . . . p. 140

D.18 Suíte 003 do TAFPlus2: Teste 040 . . . . p. 141

D.19 Suíte 004 do TAFPlus2: Testes 050 e 051 . . . . p. 142

D.20 Suíte 004 do TAFPlus2: Testes 052 e 053 . . . . p. 143

D.21 Suíte 004 do TAFPlus2: Testes 054, 055 e 056 . . . . p. 144

D.22 Suíte 004 do TAFPlus2: Testes 057 e 058 . . . . p. 145

D.23 Suíte 005 do TAFPlus2: Teste 070 . . . . p. 146

D.24 Suíte 005 do TAFPlus2: Teste 071 . . . . p. 147

D.25 Suíte 005 do TAFPlus2: Teste 072 . . . . p. 148

D.26 Suíte 005 do TAFPlus2: Teste 073 . . . . p. 149

D.27 Suíte 005 do TAFPlus2: Teste 074 . . . . p. 150

D.28 Suíte 005 do TAFPlus2: Teste 075 . . . . p. 151

D.29 Suíte 005 do TAFPlus2: Teste 075 (continuação) . . . . p. 152

D.30 Suíte 005 do TAFPlus2: Teste 075 (continuação) . . . . p. 153

D.31 Suíte 005 do TAFPlus2: Teste 075 (continuação) . . . . p. 154

D.32 Suíte 005 do TAFPlus2: Teste 076 . . . . p. 155

D.33 Suíte 005 do TAFPlus2: Testes 077 e 078 . . . . p. 156

D.34 Suíte 005 do TAFPlus2: Testes 079 e 080 . . . . p. 157

D.35 Suíte 001 do TAFStudio: Testes 001 e 002 . . . . p. 159

D.36 Suíte 001 do TAFStudio: Testes 003, 004, 005 e 006 . . . . p. 160

D.37 Suíte 002 do TAFStudio: Testes 020, 021, 022 e 023 . . . . p. 161

D.38 Suíte 002 do TAFStudio: Testes 024, 025 e 026 . . . . p. 162

D.39 Suíte 002 do TAFStudio: Testes 027 e 028 . . . . p. 163

D.40 Suíte 002 do TAFStudio: Teste 029 . . . . p. 164

D.41 Suíte 003 do TAFStudio: Testes 030, 031 e 032 . . . . p. 165

D.42 Suíte 003 do TAFStudio: Testes 033, 034 e 035 . . . . p. 166

D.43 Suíte 003 do TAFStudio: Testes 036, 037 e 038 . . . . p. 167

D.44 Suíte 003 do TAFStudio: Testes 039 e 040 . . . . p. 168

D.45 Suíte 004 do TAFStudio: Teste 050 . . . . p. 169
D.46 Suíte 004 do TAFStudio: Teste 051 . . . . p. 170
D.47 Suíte 004 do TAFStudio: Teste 052 . . . . p. 171
D.48 Suíte 004 do TAFStudio: Teste 053 . . . . p. 172
D.49 Suíte 004 do TAFStudio: Teste 054 . . . . p. 173
D.50 Suíte 004 do TAFStudio: Teste 055 . . . . p. 174
D.51 Suíte 004 do TAFStudio: Teste 056 . . . . p. 175
D.52 Suíte 004 do TAFStudio: Teste 057 . . . . p. 176
D.53 Suíte 004 do TAFStudio: Teste 058 . . . . p. 177
D.54 Suíte 004 do TAFStudio: Teste 059 . . . . p. 178
D.55 Suíte 004 do TAFStudio: Teste 060 . . . . p. 179
E.1 Faltas inseridas no TAFStudio e avaliações de resultados . . . . p. 181
E.2 Execução de testes manuais do TAFStudio (1) . . . . p. 182
E.3 Execução de testes manuais do TAFStudio (2) . . . . p. 183
E.4 Faltas inseridas no TAFPlus2 e avaliações de resultados . . . . p. 185
E.5 Execução de testes manuais do TAFPlus2 (1) . . . . p. 186
E.6 Execução de testes manuais do TAFPlus2 (2) . . . . p. 187
E.7 Casos particulares do TAFStudio e TAFPlus2 . . . . p. 189
E.8 Execução de testes manuais dos casos particulares do TAFPlus2 e TAFStudio p. 190

Resumo

Com a crescente complexidade dos programas, menor ciclo de desenvolvimento e grande expectativa de qualidade pelo cliente, tornou-se necessário otimizar o processo de localização e diagnóstico de faltas. Este trabalho propõe a elaboração de uma ferramenta que automatize a localização e diagnóstico de faltas baseada no perfil de execução (*spectrum*) de suítes de casos de teste. Utiliza-se o perfil de execução por bloco (*block hit spectra*) para registrar em quais execuções um bloco foi executado, armazenando também o resultado dos casos de teste. Assim, calcula-se a similaridade entre os blocos executados e os resultados dos testes para saber quais blocos são os maiores candidatos a estarem com falta. Serão executadas diversas suítes de casos de teste com algumas faltas habilitadas. Os resultados (diagnósticos) dessas execuções serão armazenados e posteriormente avaliados quanto a quatro fatores: coeficiente de similaridade, suíte de testes, programa alvo e granularidade de blocos. Com a introdução da ferramenta em um ciclo de testes, espera-se diminuir consideravelmente o tempo gasto para se localizar faltas, pois o diagnóstico emitido pela ferramenta diminui a quantidade de blocos que precisam ser inspecionados manualmente até se encontrar o bloco com falta. Obtêm-se também um menor tempo de entrega do produto ao mercado (*time-to-market*) e uma maior confiabilidade, pois muitas faltas que passariam despercebidas com técnicas manuais, serão detectadas e corrigidas previamente.

Palavras-Chave: Teste de *software*, localização de faltas, perfil de execução de um programa, diagnóstico automatizado, aspect programming.

Abstract

With the increasing complexity of software, shorter development cycle and high expectation by the client, it has become necessary to improve the process of localization and diagnosis of faults. This work proposes the elaboration of a tool that automates the localization and diagnosis of faults based on the spectrum of test case suites. The block hit spectra is used to register in which executions a block was executed, while storing test case results. Then, the similarity between executed blocks and the results of the test cases is calculated to know which blocks are the major candidates to have the fault. Some test case suites will be executed with enabled faults. The results (diagnosis) of these executions will be stored and subsequently evaluated on four factors: similarity coefficient, test case suite, target software and block granularity. With the introduction of the tool in a test cycle, it is expected to decrease the time spent to locate the faults, because the diagnosis delivered by the tool decreases the quantity of blocks that need to be inspected manually until the faulty block is found. Additionally, short time-to-market and a higher reliability are obtained, because a lot of faults that would pass undetected with manual techniques will be detected and fixed previously.

Keywords: Software testing, fault localization, program spectra, automated debugging, aspect programming.

1 Introdução

A elaboração e execução de testes é uma atividade essencial no ciclo de desenvolvimento de um *software*. Segundo (HAILPERN; SANTHANAM, 2002), testes são de grande importância devido à crescente complexidade dos programas, menor ciclo de desenvolvimento e grande expectativa de qualidade pelo cliente.

Durante o ciclo de desenvolvimento de um programa, apenas algumas das faltas que afetam o usuário final são localizadas e diagnosticadas (ABREU; ZOETEWEIJ; GEMUND, 2007). Esse cenário implica em uma menor confiabilidade no programa, pois ele provavelmente apresentará alguns erros que não foram localizados durante o desenvolvimento. Isso ocorre devido à grande pressão por parte do cliente para que o programa seja entregue em um curto espaço de tempo. Assim, muitas faltas passam despercebidas, não sendo detectadas com as técnicas de localização e diagnóstico manuais.

Para diminuir o tempo gasto com esse ciclo de testes, pode-se utilizar alguma técnica automatizada de localização e diagnóstico de faltas. Neste trabalho pretende-se utilizar a técnica baseada no perfil de execução (*spectrum*) de algumas suítes (coleções) de testes. O perfil de execução indica quais partes (blocos) de um programa estavam ativas durante uma determinada execução (ABREU; ZOETEWEIJ; GEMUND, 2007). Observando execuções que finalizaram com sucesso e outras que falharam, é possível identificar quais partes estão correlacionadas com os erros.

Nesta monografia, a parte de monitoramento de uma execução será implementada utilizando Programação Orientada a Aspectos (POA). A POA permite resolver alguns problemas que a Programação Orientada a Objetos (POO) não consegue solucionar adequadamente (KICZALES et al., 1997). Utilizando apenas POO, para registrar a passagem por cada bloco de código seria necessário introduzir esse novo comportamento bloco por bloco, o que tornaria a implementação difícil. Utilizando aspectos, cria-se um módulo separado que adicionará essa funcionalidade em tempo de compilação, gerando um código mais elegante.

A proposta principal deste trabalho é a elaboração de uma ferramenta que faça a localização

e diagnóstico de faltas monitorando execuções manuais de casos de teste para dois programas: TAFStudio e TAFPlus2. Esses dois programas criam ou executam testes automatizados para telefones celulares da Motorola Industrial Ltda. Para criar a ferramenta, será feita a análise e projeto do novo comportamento a ser adicionado com programação por aspectos. Após a criação do aspecto será criado um *plug-in* para gerenciar e controlar o rastreamento e também emitir os diagnósticos de uma execução.

Com a ferramenta (aspecto de rastreamento e *plug-in* de controle) implementada, serão introduzidas algumas faltas e executadas algumas suítes de testes para posteriormente avaliar os resultados obtidos. Os resultados provêm de um diagnóstico calculado por um coeficiente de similaridade que informa quais partes do programa têm a maior probabilidade de estarem apresentando faltas (ABREU; ZOETEWEIJ; GEMUND, 2007). Utilizando os resultados obtidos, serão avaliados quatro fatores para verificar se eles afetam a precisão de diagnóstico da técnica. Os fatores são os seguintes:

- **Coeficiente de Similaridade:** Serão utilizados dois coeficientes: Ochiai, que é utilizado na biologia molecular (MEYER et al., 2004) e Jaccard, proveniente de algoritmos de *data clustering* (JAIN; DUBES, 1988). Avalia-se se algum dos coeficientes obtém melhores resultados.

- **Suíte de testes:** Avaliar o impacto no diagnóstico causado por testes que não levam a falhas e outros que executam mais de um bloco do programa com defeito.

- **Programa monitorado:** Serão monitorados dois programas: TAFPlus2 e TAFStudio. Será verificado se existe diferença na precisão do diagnóstico variando o programa sob análise.

- **Granularidade:** Refere-se ao tamanho dos blocos rastreados. Nesse trabalho um método ou construtor será um bloco. Verifica-se se a diminuição ou aumento da granularidade modifica a precisão da técnica.

Assim, após introduzir a ferramenta em um ciclo de desenvolvimento de testes, espera-se diminuir o tempo gasto para se localizar e diagnosticar faltas, resultando em um menor ciclo de testes, um programa com maior confiabilidade e um menor tempo de entrega ao mercado (*time-to-market*).

Na seqüência deste trabalho, tem-se no Capítulo 2 uma breve introdução sobre Programação Orientada a Aspectos (POA). No Capítulo 3 são expostos conceitos sobre testes e é contextualizado o ambiente de testes utilizado. O capítulo 4 expõe a metodologia para localização

e diagnóstico de faltas. O Capítulo 5 descreve o projeto e implementação da ferramenta e no Capítulo 6 são criadas e executadas algumas suítes de testes e avaliados os resultados em relação aos quatro fatores citados anteriormente. No Capítulo 7 resumem-se as conclusões e possibilidades para trabalhos futuros.

2 *Programação Orientada a Aspectos*

A Programação Orientada a Aspectos (POA) é um paradigma que pertence ao grupo de paradigmas *Post-Object Programming* (POP). Esses paradigmas pretendem superar as limitações da Programação Orientada a Objetos (POO).

A POO não possibilita uma maneira eficiente de separar áreas de interesse (*concerns*) que ficam dispersas em vários componentes. Segundo (JACOBSON; NG, 2005), uma área de interesse pode ser definida como algo que é de interesse do cliente, usuário final, desenvolvedor ou patrocinador do projeto. Já um componente, no contexto de POO, pode ser uma classe. Essas áreas dispostas em vários componentes são chamadas de áreas de dispersão (*crosscutting concerns*). A POA é utilizada como um complemento à POO para implementar áreas de dispersão gerando um código separado em módulos, reusável e elegante.

Segundo (COLYER et al., 2004), uma área de dispersão necessita de certo comportamento para ocorrer em um ou mais pontos de um fluxo de execução. Mas, o foco principal do programa não é completar o comportamento requerido nesses pontos. A figura 2.1 mostra um exemplo de uma área de dispersão. A elaboração deste trecho de código foi baseada em um exemplo do livro **Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools** (COLYER et al., 2004)

O método **addTax(Tax tax)** tem como comportamento principal que uma taxa passada por parâmetro seja adicionada à lista de taxas existentes. Logo, nesse ponto, o foco principal do programa é adicionar uma taxa. Agora, observe a chamada de método destacada **notifyListeners()**.

```
public void addTax(Tax tax){
        if (tax == null){
                throw new IllegalArgumentException("A invalid tax was received!");
        }

        this.taxes.add(tag);
        notifyListeners();
}
```

Figura 2.1: Área de dispersão

Essa chamada não é o foco principal do método. Sendo assim, essa chamada é considerada uma área de dispersão que necessita de algum comportamento (nesse caso a adição de uma taxa) para ocorrer. Essa área de dispersão pode aparecer em diversos componentes e para programá-la elegantemente pode-se utilizar a Programação Orientada a Aspectos. Em um programa sem aspectos, uma mudança em áreas de dispersão impacta na modificação de todos componentes os quais ela está presente.

Ao programar uma área de dispersão utilizando aspectos, três benefícios chaves serão obtidos: **modularidade**, **encapsulamento** e **abstração**. A modularidade será obtida porque essa funcionalidade estará implementada em um único local: o aspecto. Além disso, os detalhes da implementação estarão encapsulados dentro desse aspecto. Nomeando o aspecto, obtêm-se a abstração desse conceito no sistema. Esses três benefícios diminuem a complexidade de um programa, facilitam a manutenção e possibilitam a reusabilidade de componentes.

2.1 Conceitos gerais

Nesta seção serão apresentados alguns conceitos importantes de programação por aspectos para a linguagem **AspectJ**, que é uma extensão da linguagem Java para se trabalhar com aspectos. O conteúdo desta seção é baseado no livro **Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools** (COLYER et al., 2004) e também no guia de programação do AspectJ (XEROX, 1998).

Um conceito importante é o de **pontos de junção** (*join points*). Pontos de junção são eventos no fluxo de execução de um programa. Um exemplo é a instanciação de uma classe ou a execução de um método. Assim, a linguagem AspectJ define o modelo de pontos de junção, o qual define quais eventos serão expostos como pontos de junção e poderão ser capturados pelo desenvolvedor.

A linguagem também permite a seleção de pontos de junção através de **pontos de corte** (*pointcuts*). Um ponto de corte atua como um filtro aceitando os pontos de junção desejados e bloqueando todos os outros. Uma boa prática é nomear um ponto de corte para proporcionar uma melhor compreensão de código. Além disso, existem três categorias de ponto de corte. A primeira engloba os pontos de junção baseada no tipo do ponto de junção (execução de um método, inicialização de uma classe, execução do tratador de uma exceção, etc.). A segunda diz respeito ao escopo do ponto de junção. Por exemplo, se o ponto de junção está dentro de certo pacote, ou de certa classe, etc. E a última refere-se ao contexto, que verifica, por exemplo, se o objeto que está executando é de um determinado tipo.

Na elaboração de pontos de corte utiliza-se também ***wildcards***. Um *wildcard* funciona como uma expressão regular. Ele permite uma maior expressividade na definição de pontos de corte, geralmente representando uma seqüência de caracteres capturada pelo ponto de corte. Dois exemplos de *wildcards* muito utilizados são: **"*"** e **".."**. O primeiro captura qualquer identificador com qualquer seqüência de caracteres, menos o separador de pacote ".". O segundo captura qualquer identificador com qualquer seqüência de caracteres que comece e termine com um separador de pacote ".".

Depois de capturar os eventos necessários através de pontos de corte, deve-se definir o que fazer nesses pontos. Para definir o comportamento executado antes, depois ou durante os pontos de junção agrupados pelo ponto de corte, a linguagem disponibiliza os **avisos** (*advices*). Existem três tipos de avisos:

1. **Antes** (*before*): O comportamento executará antes dos pontos de junção agrupados pelo ponto de corte.
2. **Durante** (*around*): O comportamento executará antes e depois dos pontos de junção agrupados. É o tipo de aviso mais poderoso disponibilizado pela linguagem.
3. **Depois** (*after*): O código poderá executar depois dos pontos de junção agrupados. Esse tipo divide-se em três subtipos: depois de retornar (*after returning*), depois de lançar uma exceção (*after throwing*) e depois do retorno ou lançamento de uma exceção (*after*).

Outro elemento disponibilizado pela linguagem são as **declarações inter-tipos** (*inter-type declarations*), que são utilizadas para introduzir um novo comportamento a componentes já existentes. Por exemplo, a introdução de um novo método em uma classe.

Utilizando essas definições da linguagem define-se um aspecto, que pode ser utilizado para programar áreas de dispersão modularizadas.

2.2 Ponto de Junção de Execução

Um ponto de junção é um evento no fluxo de execução de um programa. Para entender melhor sobre pontos de junção de execução, observe a figura 2.2 extraída do livro **Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools** (COLYER et al., 2004).

Pode-se observar na figura 2.2 que um ponto de junção de execução (*execution join point*) está sendo capturado toda vez que o método **aMethod()** executa.

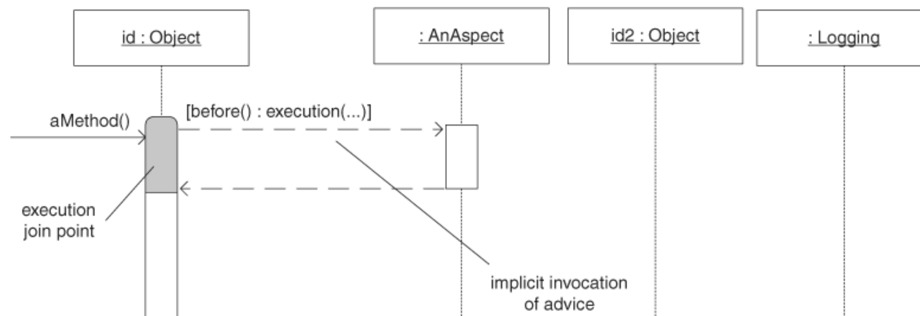


Figura 2.2: Ponto de junção de execução (*execution join point*)

A implementação do comportamento do diagrama de sequência da figura 2.2 está sendo mostrada na figura 2.3.

```
public aspect ExecutionJoinPointExample {

        // Ponte de Corte
        pointcut methodExecution() : execution(* aMethod(..));

        // Aviso
        before() : methodExecution() {
                // faça algo
        }

}
```

Figura 2.3: Codificação de um ponto de junção de execução

Nessa implementação foram utilizados um ponto de corte para abstrair o conceito de captura de execução de um método e filtrar os pontos de junção que serão capturados, e um aviso que executa algum comportamento antes da execução de um método.

A definição formal de um ponto de junção de execução pela linguagem AspectJ é a seguinte:

- **execution(*MethodSignature*)**: Especifica um ponto de junção particular de execução de método cuja assinatura é compatível com *MethodSignature*.

A definição completa da assinatura de um método (*MethodSignature*) é a seguinte:

- **[modifiers][returnTypePattern][DeclaredTypePattern.]methodName([Parameters])[throws TypePattern]**

No início dessa definição, **[modifiers]** representa características particulares de um método. As possíveis opções para esse campo são: *public*, *protected*, *private*, *final* e *static*. Na seqüência, tem-se o padrão **[returnTypePattern]** que representa o tipo retornado pelo método. **[DeclaredTypePattern.]** especifica que apenas os métodos declarados nesses tipos serão capturados. Logo após, é definido o nome do método e os parâmetros que o método recebe através

da definição **methodName([Parameters])**. Finalmente, **[throws TypePattern]** especifica que apenas os métodos que lançarem exceções de determinados tipos serão capturados.

Nesta monografia, utiliza-se o ponto de junção de execução para capturar a execução dos métodos e construtores rastreados. Assim, ao final de uma execução, obtêm-se os dados necessários para se diagnosticar qual parte do programa tem a maior probabilidade de estar com falta.

2.3 Um exemplo de aspecto

Esta seção pretende aplicar alguns conceitos explicados até agora no desenvolvimento de um aspecto. O programa deste exemplo é um sistema de gerenciamento de um hotel. Nesse sistema deseja-se registrar toda vez que um cliente reserva ou libera um quarto. Esse comportamento "corta" por vários componentes, logo é uma área de dispersão que pode ser implementada com aspectos elegantemente. Nomeia-se esse aspecto de **RoomLogging**.

Analisando o novo comportamento, verifica-se que ele impactará mudanças em duas classes do sistema: **ReserveRoomHandler** e **ReleaseRoomHandler**. Essas classes são responsáveis por reservar e liberar quartos respectivamente. Os métodos que fazem essas tarefas são: **reserveRoom()** na classe ReserveRoomHandler e **releaseRoom()** na classe ReleaseRoomHandler. Se esse comportamento fosse implementado apenas com POO seria necessário modificar cada uma dessas classes nesses dois métodos. Implementando com POA utilizando a linguagem AspectJ, pode-se criar um aspecto que irá inserir o novo comportamento em tempo de compilação.

O aspecto para inserir esse novo comportamento terá três pontos de corte (*pointcuts*): um referindo-se ao escopo (**classeAlvo()**) e outros dois aos tipos dos pontos de junção (**reservaDe Quarto()** e **liberacaoDeQuarto()**). O ponto de corte que define o escopo diz que as execuções de método dentro das classes ReserveRoomHandler e ReleaseRoomHandler serão monitoradas. Já os pontos de corte que definem os tipos dizem que apenas as execuções de métodos que reservam ou liberam um quarto serão monitoradas.

Assim, combinando os três pontos de corte, obtêm-se o código de aspecto que pode ser conferido na figura 2.4. Observa-se nesse aspecto a presença de um aviso (*advice*) que executará antes (*before*) da reserva ou liberação de um quarto. Assim, resta apenas codificar o comportamento que executará no corpo do aviso. Esse comportamento será provavelmente registrar a reserva ou liberação de um quarto em um repositório ou um arquivo.

```
public aspect RoomLogging {

        pointcut classeAlvo() : within(ReserveRoomHandler) || within(ReleaseRoomHandler);

        pointcut reservaDeQuarto() : classeAlvo() && execution(reserveRoom());

        pointcut liberacaoDeQuarto() : classeAlvo() && execution(releaseRoom());

        before() : reservaDeQuarto() || liberacaoDeQuarto() {
                // Registrar uma reserva ou liberação de quarto em algum repositório ou arquivo.
        }

}
```

Figura 2.4: Exemplo de aspecto para registrar reservas e liberação de quartos

2.4 Composição

A composição dos aspectos com o código da aplicação é feita por um procedimento de compilação denominado *weaving* (KICZALES et al., 1997). Existem três diferentes formas de compor os aspectos com classes de um programa:

- **Composição em tempo de compilação (Compile-time weaving)**: A composição é feita em tempo de compilação. Nesse caso o desenvolvedor tem os arquivos fonte do programa e o aspecto está em formato fonte (.java) ou binário (.class).
- **Composição depois da compilação (Post-compile weaving)**: Abordagem parecida com a anterior, mas agora o desenvolvedor tem o aspecto em formato fonte ou binário, mas as classes do programa estão em formato binário ou dentro de um arquivo JAR.
- **Composição em tempo de carregamento (Load-time weaving)**: Essa técnica só compõe os aspectos com as classes quando elas forem carregadas por um Carregador de Classes (*Class Loader*). Para que essa composição funcione, devem-se ter carregadores de classes específicos para fazer a composição.

Neste trabalho, a composição é feita em tempo de compilação, pois se tem acesso aos arquivos fonte do programa e do aspecto. A figura 2.5 mostra esquematicamente como funciona a composição executada nesta monografia.

Para facilitar o desenvolvimento e a composição de aspectos neste trabalho, utiliza-se o projeto **AspectJ Development Tools (AJDT)**. Esse projeto provê ferramentas para facilitar a programação de aspectos utilizando a **IDE Eclipse** (ECLIPSE, 2009a). Serão dadas maiores informações sobre esse projeto na seção 5.4.

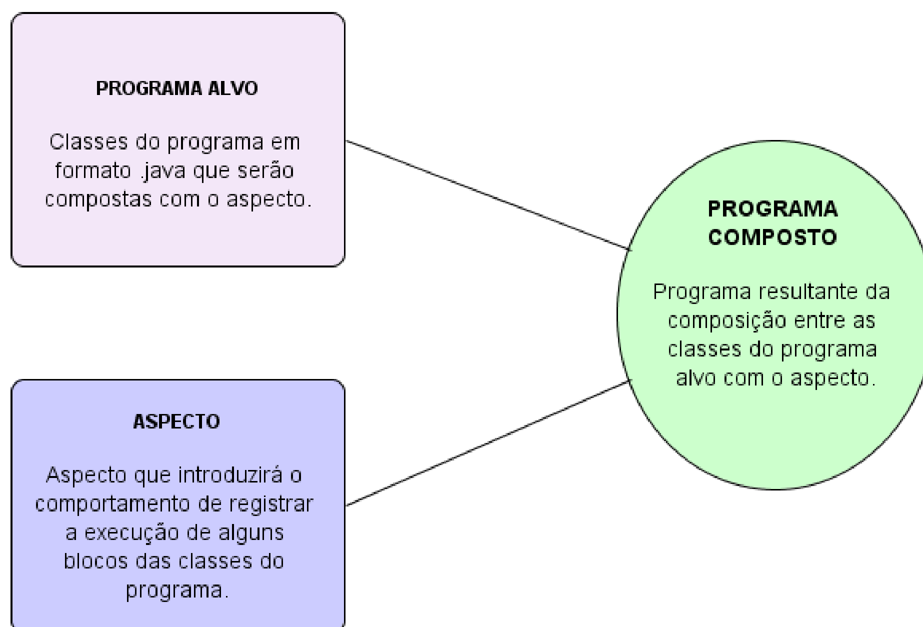


Figura 2.5: Composição de aspecto com um programa

3 *Testes*

3.1 Conceitos gerais

Segundo (MYERS, 2004), testar é o processo de executar um programa com a intenção de localizar erros. Nos itens abaixo, serão expostos alguns conceitos básicos retirados de (AVIZIENIS et al., 2004) e que são indispensáveis para a compreensão deste trabalho.

- **Falha**: É um evento que ocorre quando um serviço obtido difere do serviço esperado.
- **Erro**: É um estado do sistema que pode causar uma falha.
- **Falta**: É a causa de um erro no sistema. Algumas vezes a palavra defeito é utilizada como sinônimo de falta.

Uma das grandes dúvidas ao se elaborar testes é saber quando se pode parar, pois já foram cobertas todas as situações possíveis e existe uma garantia que o programa não apresentará erros. O grande problema é a dificuldade de garantir que um programa não tem erros, pois encontrar todos esses erros é na maior parte das vezes impraticável (MYERS, 2004).

Para se detectar a maior parte dos erros, existem algumas estratégias para criação de testes. A seguir serão descritas duas das mais importantes delas.

3.2 Testes Caixa Branca

A estratégia chamada de **Caixa Branca** (*White-Box*) permite que o testador examine a estrutura interna de um programa, isto é, a lógica do código. Os dados a serem utilizados no teste são derivados da lógica do programa.

Uma das idéias dessa técnica é o teste de todos os possíveis caminhos no fluxo de execução de um programa (teste exaustivo). Assim, se todos os testes passarem, o programa não contêm erros. Essa abordagem é inadequada para maior parte dos programas, pois existe uma grande

quantidade de caminhos possíveis a serem testados. Logo, o tempo gasto para se testar todos os caminhos atrasaria a entrega do programa, gerando insatisfação para o cliente.

3.3 Testes Caixa Preta

A estratégia chamada de **Caixa Preta** (*Black-Box*) não considera a estrutura interna de um programa, mas busca encontrar circunstâncias pelas quais o programa não se comporta de acordo com sua especificação. Essa estratégia de teste deriva os dados de entrada somente da especificação do programa.

Para se garantir que um programa não tem erros, deve-se testar todas as entradas possíveis para o mesmo. Logicamente, para maior parte dos programas, é muito difícil testar todas essas possibilidades. Imagine testar todas as possíveis entradas de uma função que recebe três números inteiros como parâmetro. Seriam necessários milhões, talvez bilhões de casos de teste variando as entradas até o maior número inteiro possível de ser representado em uma dada linguagem. Assim, geralmente são criados testes para algumas entradas prováveis e outras improváveis. Mas não há nenhuma garantia que um determinado valor de entrada não possa causar um erro no programa.

Neste trabalho, os casos de teste foram elaborados utilizando essa técnica, pois se pretende verificar que o programa está de acordo com sua especificação. Obviamente, não foram elaborados testes que verificassem todas as entradas possíveis, mas apenas as prováveis, improváveis e alguns casos genéricos.

3.4 Suítes de casos de teste

Uma suíte de casos de teste é uma coleção de casos de teste. Uma suíte pode conter também anotações e informações importantes para cada teste. Já um caso de teste contém uma série de passos com resultados esperados. Um caso de teste não precisa obrigatoriamente ter um resultado esperado. O objetivo de um caso de teste é verificar se uma parte de um programa está funcionando corretamente. Alguns testes também contêm pré-condições que precisam ser garantidas para que ele possa executar.

Os casos de teste foram agrupados em suítes de acordo com a funcionalidade que verificam. Por exemplo, existe uma suíte de testes que verifica o comportamento do controle de uma execução. Outra suíte pode testar as funcionalidades de criação, edição e remoção de recursos. Essa separação foi feita para previnir que existisse mais de um defeito por suíte de testes. Serão

dados mais detalhes sobre a criação de testes e a necessidade dessa separação no Capítulo 6.

3.5 Testes automatizados

Como o tempo de entrega de um produto ao mercado (*time-to-market*) está cada vez menor, deve-se buscar soluções que possam diminuir o tempo gasto no ciclo de testes. A automatização de testes é uma alternativa de solução para esse problema. Testes automatizados podem ser reaproveitados, têm uma maior precisão e diminuem os custos de um ciclo de testes a longo prazo. A grande desvantagem da automatização são os custos iniciais e a não apresentação de resultados imediatos, pois é preciso implementar algum programa para criar e executar os casos de teste.

É importante esclarecer que os casos de teste criados nesta monografia serão executados manualmente, pois testam funcionalidades dos programas monitorados. Os casos de teste automatizados são para telefones celulares e são criados ou executados pelos dois programas monitorados: TAFStudio e TAFPlus2. A automatização dos casos de teste para os programas monitorados é uma tarefa que foge do escopo desse trabalho.

3.6 Ambiente de testes

Os dois programas que serão monitorados pela ferramenta de localização e diagnóstico de faltas foram criados para possibilitar a automatização de casos de teste para telefones celulares produzidos pela Motorola Industrial Ltda. Nesta seção será contextualizado o ambiente de testes e dada uma visão geral da arquitetura e das funcionalidades desses programas.

3.6.1 *Test Automation Framework* (TAF)

O *Test Automation Framework* (TAF) é um *framework* criado para automatizar casos de teste dos *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola Industrial Ltda (KAWAKAMI et al., 2007). Esse *framework* permite a reutilização de testes e a execução automatizada dos mesmos, diminuindo o tempo gasto no ciclo de desenvolvimento. Assim, não é necessário criar um novo teste para cada modelo de telefone a ser testado. Os testes podem ser reaproveitados e apenas algumas modificações são necessárias.

A comunicação entre o TAF e o telefone celular é obtida por uma biblioteca proprietária denominada *Phone Test Framework* (PTF). Essa biblioteca se comunica com o telefone pela

interface USB (*Universal Serial Bus*) e provê funções como retornar uma foto do que está sendo mostrado no telefone, pressionar teclas do telefone ou verificar o aparecimento de uma determinada tela, dentre outras funções (ESIPCHUK; VALIDOV, 2006).

Arquitetura

A arquitetura resumida do TAF pode ser visualizada na figura 3.1 (PETROSKY, 2006). Nessa arquitetura observam-se quatro camadas: *Test Case*, *Feature Toolkit*, *Utility Function* e *Phone Test Framework*. A camada inferior (PTF) é a responsável pela comunicação com o telefone celular. A camada *Utility Function* (UF) encapsula as chamadas feitas ao PTF. Uma UF é um passo (*Step*) de um caso de teste em alto nível. Um exemplo de UF é a composição de uma mensagem. A camada *Feature Toolkit* é responsável por criar instâncias de UF, passando os argumentos necessários, e acrescentar essas UFs para serem executadas por um caso de teste. A camada *Test Case* representa um caso de teste em alto nível. Segundo (RECHIA et al., 2007), um caso de teste é uma lista de *Steps*, onde tudo que um caso de teste faz é invocar o método *execute()* definido na interface *Step* e implementado nas UFs concretas.

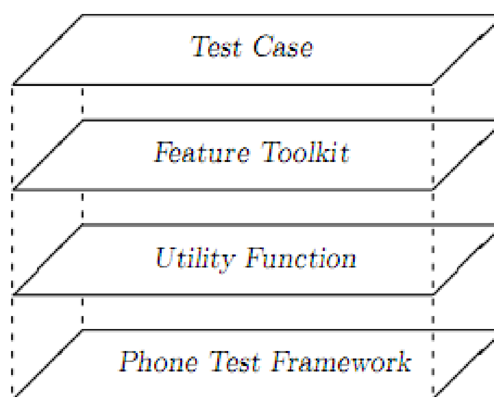


Figura 3.1: Arquitetura do *Test Automation Framework* (TAF)

3.6.2 TAFPlus2

O **TAFPlus2** é um programa para facilitar a execução de casos de teste automatizados criados pelo TAF. Esse programa dispõe de uma interface gráfica e permite controlar a execução de vários casos de teste simultaneamente. O TAFPlus2 foi criado pois a execução de casos de teste utilizando apenas o TAF é um pouco complexa e necessita de um certo conhecimento do *framework* e de linguagens de programação. Logo, o objetivo do TAFPlus2 é permitir que usuários sem conhecimento do TAF possam executar casos de testes criados no TAF. Por isso, uma preocupação importante do TAFPlus2 é a usabilidade.

O TAFPlus2 provê muitas funcionalidades interessantes além da possibilidade de executar suítes de casos de teste. As principais funcionalidades são:

- Controle da ordem de execução de testes e também de passos de casos de teste.
- Envio automático de informações do caso de teste para um repositório de testes.
- Fácil configuração dos telefones onde os testes serão executados.
- Armazenamento de informações de cada execução de caso de teste para posterior análise.

Arquitetura

As características fundamentais da arquitetura do TAFPlus2 são:

- **Distribuída e baseada em serviços**: Toda comunicação é feita através de serviços. Qualquer execução é um serviço.
- **Baseada em eventos**: Eventos são utilizados para sincronização entre classes.
- ***Model-View-Controller* (MVC)**: Padrão de arquitetura de software que separa o modelo de negócios (*model*) da apresentação (*view*). Já o controle (*controller*) recebe os eventos e os repassa para o modelo.
- **Repositório**: Um repositório provê uma interface fácil para acessar e armazenar elementos do TAFPlus2 no disco.

O TAFPlus2 é um dos programas que será monitorado pela ferramenta para se localizar e diagnosticar faltas em seus componentes.

3.6.3 TAFStudio

Com o TAFPlus2, a execução de casos de teste automatizados tornou-se mais simples. Mas os testes ainda precisam ser criados utilizando o TAF, o que demanda um conhecimento detalhado do *framework*. O objetivo do **TAFStudio** é permitir que usuários sem nenhum conhecimento em linguagens de programação possam criar casos de teste simplesmente executando os passos diretamente no telefone celular. Assim, o TAFStudio grava os passos executados manualmente e essa sequência de passos pode ser salva como um caso de teste que depois poderá ser executado.

As funcionalidades principais que o TAFStudio provê são:

- Gravação dos passos de um teste executados manualmente em um telefone celular para posterior execução automatizada.
- Criação de suítes de testes para agrupar casos de teste.
- Pontos de verificação (*checkpoints*) podem ser definidos para verificar se determinados itens estão na tela do telefone em um determinado momento.
- Configuração automática do telefone.
- Controle de execução de casos de teste.
- Controle na gravação de passos para um caso de teste.

Arquitetura

Para gravar os passos de um caso de teste para futura execução, o TAFStudio monitora dois eventos fundamentais:

- **Pressionamento das teclas do telefone**: Identifica-se a tecla pressionada, o tempo de pressionamento e o tempo decorrido desde o último pressionamento.
- **Telas capturadas do telefone**: Identifica-se o posicionamento e tamanho de cada item de tela.

Assim, os casos de teste criados no TAFStudio são compostos por passos que representam eventos ocorridos no telefone. A grande diferença entre um passo do TAF e um passo do TAFStudio é o nível de abstração. O passo do TAFStudio é de baixo nível, pois representa um evento dependente do modelo do telefone executado. Logo, a reutilização de casos de teste torna-se difícil.

O TAFStudio é o outro programa que será monitorado pela ferramenta de localização e diagnóstico de faltas.

4 Metodologia

4.1 Localização e diagnóstico de faltas pelo perfil de execução

A localização e diagnóstico de faltas manual é uma tarefa que demanda um bom tempo para o testador. Após executar um caso de teste ou uma suíte de testes, o testador deve analisar o erro obtido e procurar exaustivamente em cada trecho de código pela falta que causou esse erro. Um programa com muitas linhas de código torna essa tarefa muito custosa, pois examinar cada linha de código demanda muito tempo.

Para diminuir o tempo gasto na localização e diagnóstico manual, pode-se utilizar alguma técnica automatizada. Nesta monografia se utiliza a técnica de **localização e diagnóstico de faltas baseada no perfil de execução** (*Spectrum-Based Fault Localization*). Segundo (REPS et al., 1997), o **perfil de execução** (*spectrum*) é uma coleção de dados que representa o comportamento dinâmico de um programa. Esses dados são coletados em tempo de execução e posteriormente podem ser utilizados para fins de diagnóstico.

Neste trabalho, utiliza-se o **perfil de execução por bloco** (*block hit spectra*). Para cada bloco rastreado se tem um valor associado ao mesmo, indicando se uma execução passou ou não por esse bloco. Um bloco que foi executado terá um valor um, enquanto um bloco que não for executado terá o valor zero. Esses valores serão coletados para um número de suítes de testes executadas e serão armazenados em uma matriz do seguinte formato:

$$MS_{m,n} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix} VE_{m,1} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}$$

Nessa matriz, as colunas representam os diferentes blocos dos programa monitorados. Já as linhas representam as diferentes execuções de testes. Além dessa matriz, existe um vetor de erro, que registra quando uma execução falhou. Se uma execução falhar existirá um número um

nesse vetor. Caso contrário, existirá um número zero. Uma execução será considerada falha se o resultado esperado for diferente do resultado obtido. Essa detecção de erros baseada em falhas é uma forma rudimentar de detecção de erros, pois muitas vezes alguns erros nunca levam a uma falha e permanecerão indetectáveis. Neste trabalho, cada caso de teste será considerado uma execução e cada método ou construtor será um bloco.

Após coletar esses dados, a técnica de localização e diagnóstico de faltas tentará encontrar qual bloco mais se assemelha com o vetor de erro. Para achar a semelhança entre a matriz e o vetor de erro pode-se utilizar **coeficientes de similaridade** (JAIN; DUBES, 1988). Neste trabalho, serão utilizados dois coeficientes: **Jaccard e Ochiai**. Segundo (ABREU; ZOETEWEIJ; GEMUND, 2006), o coeficiente de Ochiai apresenta melhores resultados que o coeficiente de Jaccard. Essa conclusão foi obtida submetendo os mesmos blocos defeituosos e testes executados para quatro coeficientes diferentes: Jaccard, Tarantula (JONES; HARROLD; STASKO, 2002), Ample (DALLMEIER; LINDIG; ZELLER, 2005) e Ochiai. Assim, ao avaliar os resultados, concluiu-se que o coeficiente de Ochiai é de 2,4% a 10% mais preciso que o coeficiente de Jaccard (o segundo coeficiente com maior precisão). Em geral, o coeficiente de Ochiai diminui o percentual de blocos que precisam ser inspecionados em até 5% .

Mesmo assim, optou-se por utilizar os dois coeficientes, pois assim pode-se verificar se no âmbito deste trabalho a diferença entre eles é significativa. Mas, para uma dada execução, sempre será considerado o coeficiente de Ochiai como resultado com maior confiabilidade.

A seguir estão sendo mostradas as fórmulas para cálculo dos coeficientes:

$$s_J(j) = \frac{a_{11}(j)}{a_{11}(j)+a_{10}(j)+a_{01}(j)} \quad (1)$$

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j)+a_{01}(j))*(a_{11}(j)+a_{10}(j))}} \quad (2)$$

A fórmula (1) refere-se ao coeficiente de Jaccard, utilizado nos algoritmos de *data clustering*. Enquanto a segunda (2), refere-se ao coeficiente de Ochiai, proveniente da biologia molecular. A seguir serão explicados os elementos da fórmula. O elemento $a_{11}(j)$ indica que no bloco j para uma determinada execução, o bloco foi executado e a execução falhou. Já o elemento $a_{01}(j)$ indica que o bloco j não foi executado em uma execução que falhou. Por fim, o elemento $a_{10}(j)$ indica que o bloco j foi executado e a execução passou. Nota-se que para um determinado bloco ter um *spectrum* diferente de zero, o elemento $a_{11}(j)$ deve ser diferente de zero. Isso significa que deve existir pelo menos alguma situação em que o bloco foi executado e o teste que o executou falhou.

Após o cálculo do coeficiente de similaridade, os blocos que obtiverem o maior resultado são os candidatos com a maior probabilidade de estarem apresentando uma falta. Para entender melhor como funciona o cálculo do coeficiente, na próxima seção será mostrado um simples exemplo de localização e diagnóstico de uma falta.

4.2 Um exemplo de diagnóstico

Neste exemplo, está sendo simulada a execução de quatro casos de teste. Estão sendo monitorados quatro blocos de um programa. Logo, a ordem da matriz de *spectrum* (MS) é 4x4. Cada linha da matriz representa a execução de um caso de teste. Cada coluna representa um bloco do programa. Existe também o vetor de erros (VE) que armazena o resultado dos testes. A ordem do vetor de erros é 4x1.

Pode-se utilizar POA para implementar o rastreamento da execução dos casos de teste. Após implementar o rastreamento, deve-se então introduzir uma falta e executar os testes para verificar a precisão dos diagnósticos emitidos pelos coeficientes de similaridade (Jaccard e Ochiai). Foi introduzida uma falta no bloco 2 (segunda coluna da matriz de *spectrum*).

Após a introdução da falta, executam-se os casos de teste com o rastreamento habilitado. Esses testes foram pré-selecionados, pois executam funções que passam pelo bloco com falta em seu fluxo de execução. Os testes 1, 3 e 4 falharam (tem o número um no vetor de erros), enquanto o teste 2 passou (tem o número zero no vetor de erros). O registro de quais blocos foram executados foi armazenado na matriz de *spectrum* (o número um representa que o bloco foi executado, zero caso contrário), enquanto o resultado dos testes foi armazenado no vetor de erros. As duas matrizes podem ser visualizadas abaixo:

$$MS_{4,4} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} VE_{4,1} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Ao analisar a matriz de spectrum e o vetor de erros em relação ao bloco com falta (bloco 2), nota-se que o caso de teste 2 não falhou e não executou o bloco. É importante observar também que o bloco defeituoso foi executado em dois testes que falharam (testes 1 e 3). O teste 4 falhou mas não executou o bloco com falta. Utilizando o coeficiente de Ochiai da fórmula (2), a probabilidade de falta do bloco 2 é calculada com a seguinte fórmula:

$$s_O(j) = \frac{2}{\sqrt{(2+1)*(2+0)}} = 0,33$$

O elemento $a_{11}(j)$ tem o valor dois, pois em dois casos de teste o bloco foi executado e o teste falhou. Já $a_{01}(j)$ tem o valor um, pois em um caso de teste o bloco não foi executado e o teste falhou. O elemento $a_{10}(j)$ tem o valor zero, pois em nenhuma ocasião o bloco foi executado e o teste passou. Logo, a probabilidade de falta do bloco 2 é de 0.33.

Ao calcular o *spectrum* para os blocos 1, 3 e 4 são obtidas as seguintes probabilidades de falta respectivamente: 0.22 , 0.33 e zero. Assim, ordenando os blocos pela maior probabilidade de falta obtêm-se o seguinte resultado: bloco 2, bloco 3, bloco 1 e bloco 4. É importante observar que não existe diferenciação entre blocos com mesmo valor de *spectrum*. Os blocos 2 e 3 têm a mesma probabilidade de falta. Logo, dentre 4 blocos monitorados, 2 foram reportados com a maior probabilidade de falta. Assim, 50% dos blocos necessitam ser inspecionados manualmente até se encontrar o bloco defeituoso (bloco 2).

4.3 Vantagens e Desvantagens

Uma das vantagens da técnica utilizada nesta monografia é que ela pode ser facilmente integrada em qualquer programa, e não causa perda de desempenho mesmo com recursos limitados de processamento e memória. Outra vantagem é que não é necessário elaborar um modelo do programa, pois ela é uma estratégia de teste do tipo caixa preta. Como dito anteriormente, uma estratégia desse tipo não se concentra na implementação interna do programa, mas sim em encontrar circunstâncias nas quais o programa não se comporta como o esperado (MYERS, 2004).

Segundo (ABREU; ZOETEWEIJ; GEMUND, 2006), essa técnica é muito boa para ser integrada em *software*. Para se localizar faltas em *hardware* deve-se utilizar alguma técnica baseada em modelo. No âmbito de *software*, não se pode utilizar uma técnica baseada em modelo, pois elaborar o modelo de um programa é uma tarefa extremamente complexa.

A grande desvantagem da técnica baseada no perfil de execução é que algumas vezes blocos são classificados com grande probabilidade de estarem com falta, quando na verdade não estão. Isso pode acontecer quando um erro não leva a uma falha. Existem outros pontos importantes a serem avaliados a respeito da técnica, os quais serão explicados em detalhes no Capítulo 6 e Capítulo 7.

5 *Análise, Projeto e Implementação da Ferramenta*

Este capítulo trata da análise, projeto e implementação da ferramenta para localização e diagnóstico de faltas. Essa ferramenta é composta por um **aspecto de rastreamento** (desenvolvido utilizando a IDE Eclipse (ECLIPSE, 2009a) e AspectJ (KICZALES et al., 1997)) e de um ***plug-in* de controle** (desenvolvido utilizando a IDE Eclipse).

Primeiramente será feita a análise, projeto e implementação do aspecto de rastreamento na seção 5.1. Na seção 5.2, os blocos dos programas monitorados serão analisados para verificar se alguns blocos podem ser retirados do rastreamento sem impactar no diagnóstico. A seção 5.3 trata da análise, projeto e implementação do *plug-in*. Ao final, na seção 5.4 mostra-se como foi feita a composição entre o aspecto e os programas monitorados. Após a composição, a ferramenta está pronta e já pode ser utilizada para localizar e diagnosticar faltas.

5.1 Análise, Projeto e Implementação do aspecto

A análise e projeto do aspecto a ser desenvolvido nesta monografia são baseados no livro **Aspect-Oriented Software Development with Use Cases** (JACOBSON; NG, 2005). Nesse livro, Jacobson explica como modelar aspectos de uma maneira modular utilizando casos de uso, além de separar as áreas de dispersão (*crosscuting concerns*), resultando em um código de fácil manutenção e reuso.

O primeiro passo ao se analisar um novo comportamento, é compreender o que o cliente deseja inserir no sistema. Ao conversar com o cliente, conclui-se que ele deseja inserir o registro de toda chamada de método ou construtor em determinadas classes do programa. Assim, esse é um **requisito não funcional**. Segundo (JACOBSON; NG, 2005), um requisito não funcional representa um atributo de qualidade do sistema. Já um **requisito funcional**, representa algo que o usuário pode fazer no sistema.

Atualmente, muitos engenheiros de *software* não modelam requisitos não funcionais com

casos de uso, mas apenas referenciam esses requisitos nos casos de uso dos requisitos funcionais. A metodologia utilizada neste trabalho pretende modelar um caso de uso para os requisitos não funcionais que demandam algum processamento. No entanto, existem alguns requisitos não funcionais que não podem ser modelados com casos de uso. O requisito não funcional "O programa deve ser implementado na linguagem Java" não pode ser modelado com um caso de uso, pois não demanda nenhum processamento do sistema e é apenas uma característica desejada pelo usuário. Um **caso de uso** ajuda a entender o comportamento do sistema, pois demonstra o que o mesmo deve fazer como resposta as ações do usuário.

Ainda segundo (JACOBSON; NG, 2005), requisitos não funcionais geralmente necessitam do suporte de algum mecanismo de infra-estrutura para serem realizados. Por isso, os casos de uso para requisitos não funcionais que demandam certo processamento são chamados de **casos de uso de infra-estrutura**. Os casos de uso para requisitos funcionais são nomeados **casos de uso de aplicação**.

Como os requisitos não funcionais geralmente afetam cada passo de um caso de uso de aplicação, pode-se criar um caso de uso genérico que representa o que acontece durante um determinado passo. Esse caso de uso é um padrão, representando a requisição de um ator para posteriormente receber a resposta do sistema. A esse caso de uso é dado o nome de ⟨ **Executar Transação** ⟩. O nome do caso de uso está entre parênteses angulares, pois o ator e o caso de uso são elementos parametrizados. Eles podem variar dependendo do caso de uso de aplicação. A especificação desse caso de uso de infra-estrutura foi extraída do livro **Aspect-Oriented Software Development with Use Cases** (JACOBSON; NG, 2005) e está sendo mostrada na tabela 5.1.

Nessa especificação, foram omitidos os fluxos alternativos e requisitos especiais, pois seria necessário fazer um levantamento das funcionalidades dos programas alvo e fugiria do escopo deste trabalho. Um exemplo de requisito especial seria que nenhuma transação poderia demorar mais que 2 segundos. Já os fluxos alternativos representam os diversos caminhos que um caso de uso pode tomar.

É importante observar o ponto de extensão (*extension point*) E1. Um **ponto de extensão** no caminho de execução de um caso de uso define onde os comportamentos adicionais poderão ser inseridos. Em POA esses pontos são conhecidos como pontos de junção. Para entender melhor o contexto dos mecanismos de infra-estrutura e também localizar facilmente pontos de extensão, recomenda-se modelar os outros requisitos não funcionais como extensões do caso de uso ⟨ Executar Transação ⟩. A figura 5.1 mostra o diagrama de casos de uso de ⟨ Executar Transação ⟩.

Especificação do caso de uso: ⟨ **Executar Transação** ⟩
⟨ Executar Transação ⟩ **Fluxo Principal**
O caso de uso começa quando um ator executa uma transação para ver os valores de uma entidade.
1.O sistema pede ao ator que ele identifique qual é a entidade desejada.
2.O ator entra os valores e submete a requisição.
3.O sistema retorna a entidade e mostra os valores.
4.O caso de uso termina.
Fluxos Alternativos
...
Requisitos Especiais
...
Pontos de Extensão
E1. Executar Blocos
O ponto de extensão Executar Blocos ocorre em todos os passos do fluxo principal.

Tabela 5.1: Especificação do caso de uso ⟨ Executar Transação ⟩

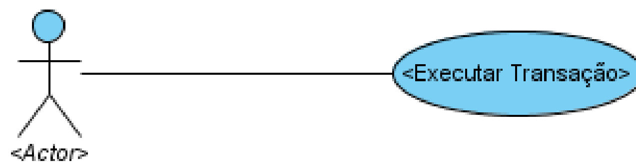


Figura 5.1: Caso de uso ⟨ Executar Transação ⟩

A tabela 5.2 mostra a especificação do caso de uso **Gerenciar Rastreamento de Blocos**. Esse caso de uso estenderá ⟨ Executar Transação ⟩ pois é um requisito não funcional do sistema que exige algum processamento.

Especificação do caso de uso: **Gerenciar Rastreamento de Blocos**
Fluxo Principal
Sem fluxo principal.
Fluxos Alternativos
Sem fluxos alternativos.
Requisitos Especiais
Sem requisitos especiais.
Fluxos de Extensão
EF1. Registrar Passagem por Bloco
Esse fluxo de extensão ocorre no ponto de extensão Executar Blocos (E1) do caso de uso ⟨ Executar Transação ⟩ antes de um bloco ser executado.
1. O sistema registra a execução do bloco atual.
2. O controle é retornado ao caso de uso base.

Tabela 5.2: Especificação do caso de uso Gerenciar Rastreamento de Blocos

O fluxo de extensão EF1 indica aonde o fluxo será inserido no caso de uso base. Em POA, seria o equivalente a um ponto de corte que indica aonde extensões de operações seriam

inseridas em operações já existentes. Uma informação importante para casos de uso de extensão é que nunca se deve fazer referência direta a um passo do caso de uso base. Isso tornaria a referência inválida se um novo passo fosse introduzido no caso de uso base.

Como sugerido pela **Unified Modeling Language (UML)**, modela-se pontos de extensão através de uma nota no relacionamento de extensão entre o caso de uso base e o caso de uso de extensão. A modelagem do caso de uso Gerenciar Rastreamento de Blocos pode ser conferida na figura 5.2.

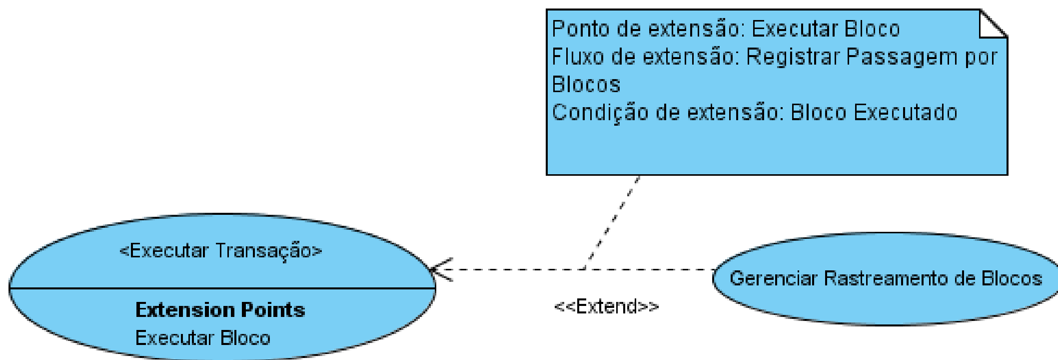


Figura 5.2: Ponto de extensão modelado em um diagrama de casos de uso

Pode-se observar nessa figura, que o caso de uso Gerenciar Rastreamento de Blocos estende ⟨ Executar Transação ⟩ no ponto de extensão Executar Bloco. Quando um bloco for executado será registrada a passagem pelo mesmo. Observa-se também que esse caso de uso não tem nenhum fluxo principal. Isso significa que ele é um caso de uso puramente extensivo e não pode ser instanciado independentemente. Um caso de uso puramente extensivo sempre executará no contexto de um outro caso de uso base.

Avançando um pouco mais e pensando em um nível de projeto, deve-se tomar cuidado para continuar mantendo as áreas de interesse separadas. Com técnicas tradicionais de modelagem, a introdução de casos de uso de extensão em casos de uso base causa emaranhamento de código na fase de implementação. Utilizando-se aspectos, **fatias de caso de uso** podem ser utilizadas para separar o novo comportamento. Uma fatia de caso de uso tem esse nome pois “corta” o modelo de projeto.

Segundo (JACOBSON; NG, 2005), uma fatia de caso de uso é composta por:

- Uma colaboração que descreve a realização do caso de uso.
- Classes específicas para realização do caso de uso.
- Extensões de classes existentes para realização do caso de uso.

Cada caso de uso tem uma fatia associada ao mesmo no modelo de projeto. Essa fatia mantém os comportamentos específicos de uma realização de caso de uso. Por exemplo, na realização do caso de uso Gerenciar Rastreamento de Blocos, precisam-se adicionar comportamentos em operações já existentes em determinados pontos. Para alcançar esse objetivo, será criada uma fatia de caso de uso para esse caso de uso de extensão.

Antes de criar a fatia do caso de uso Gerenciar Rastreamento de Blocos, serão elaborados outros artefatos que ajudam o compreender o novo comportamento. Um artefato importante é o diagrama de papéis, que mostra a separação dos papéis de cada classe. O primeiro papel é o de **Rastreamento**. Esse papel é cumprido pela classe responsável pelo rastreamento. Essa classe foi denominada de **TraceManager**. Ela registrará as execuções de blocos (métodos ou construtores). As outras classes envolvidas são as classes monitoradas do programa alvo. Essas classes têm o papel de **Alvo** do novo comportamento. Conclui-se então que o registro de execução de blocos de uma classe monitorada será feito pela classe TraceManager. O diagrama de papéis está sendo mostrado na figura 5.3.

Figura 5.3: Diagrama de papéis

Com o intuito de visualizar onde a extensão de operação será inserida, foram modelados dois diagramas de seqüência que podem ser conferidos nas figuras 5.4 e 5.5. Esses diagramas demonstram que o método **traceEntry()** será executado antes da execução de um método ou de um construtor de alguma das classes monitoradas do programa alvo. Este é o novo com-

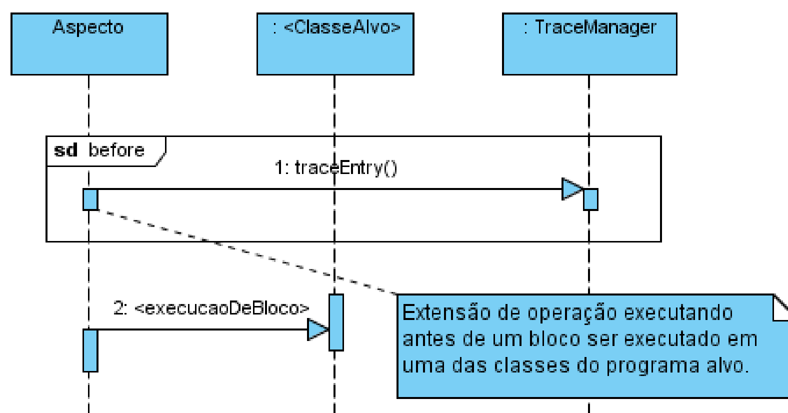


Figura 5.4: Diagrama de seqüência do rastreamento de métodos

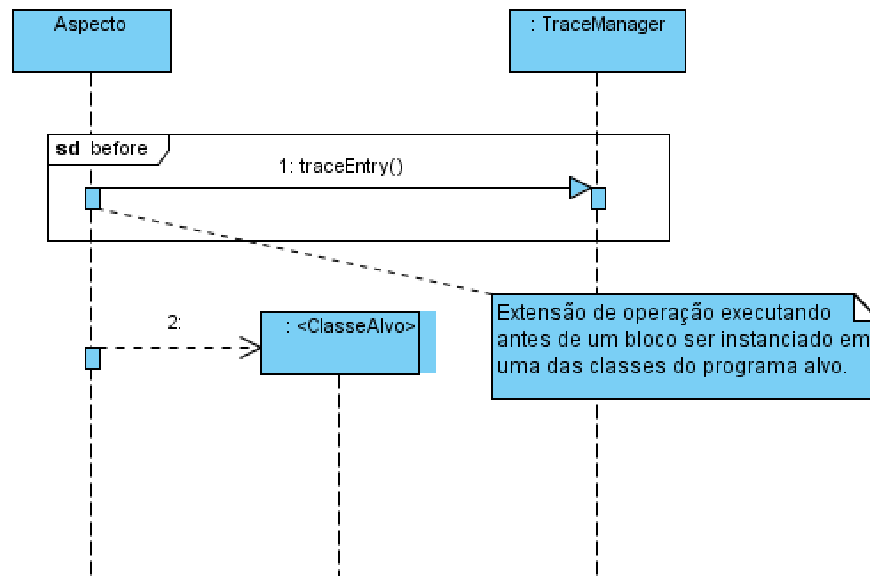


Figura 5.5: Diagrama de seqüência do rastreamento de construtores

portamento que será inserido. Um ponto interessante do diagrama de métodos é a operação ⟨ execucaoDeBloco ⟩. Essa chamada foi parametrizada, pois se pretende estender múltiplas operações de uma única vez. A POA facilita a definição de operações parametrizadas através de pontos de corte, que foram explicados na seção 2.1 e serão demonstrados em maiores detalhes na parte de implementação.

É importante ressaltar também que existem dois contextos para identificar em quais pontos a operação de extensão será executada. O contexto **estrutural** define em qual pacote, qual classe e qual operação nós estaremos executando. No diagrama acima, a extensão de operação será executada na ⟨ ClasseAlvo ⟩ que também está parametrizada. Uma ⟨ ClasseAlvo ⟩ representa uma classe monitorada do TAFStudio ou do TAFPlus2. O contexto **comportamental** define em qual ponto do fluxo de execução será executada a nova operação. Nesta monografia, esse ponto é a execução de um bloco (método ou construtor). Mais especificamente, antes da execução de um bloco.

Após a elaboração desses artefatos, pode-se criar a fatia de caso de uso para o caso de uso Gerenciar Rastreamento de Blocos. Essa fatia pode ser conferida na figura 5.6. Nessa fatia de caso de uso, o aspecto de Rastreamento contém uma extensão de classe para ⟨ ClasseAlvo ⟩. Uma classe alvo executa métodos e construtores. A execução de um construtor da classe é rastreada. A execução de métodos é rastreada, exceto os métodos que começam com *get* ou *set*. Finalmente, o novo comportamento aparece na seção de extensões de classe. Ele define que **antes** (*before*) de uma execução de método ou execução de construtor, será registrada a execução do mesmo com a chamada ao método **traceEntry()** da classe TraceManager.

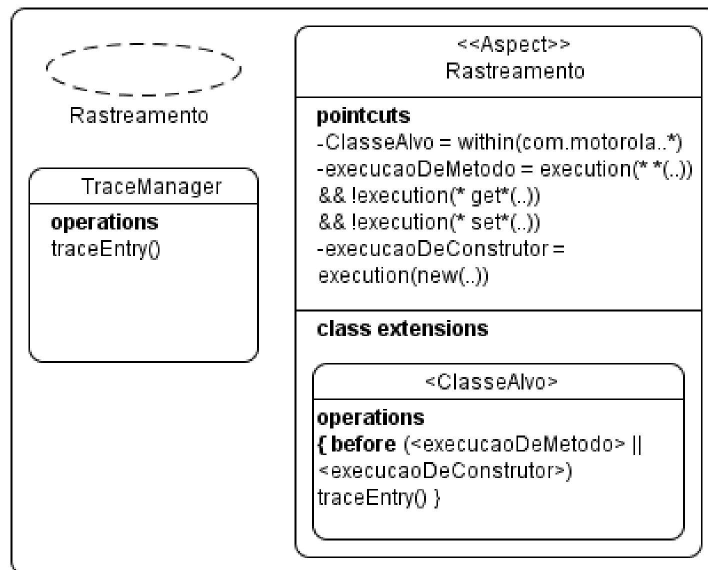


Figura 5.6: Fatia de caso de uso

A primeira implementação do aspecto pode ser visualizada na figura 5.7. Nesse código, nota-se que os pontos de corte ⟨ ClasseAlvo ⟩, ⟨ execucaoDeConstrutor ⟩ e ⟨ execucaoDeMetodo ⟩ foram implementados. O primeiro deles é do tipo estrutural e os outros dois são do tipo comportamental. Observa-se também a implementação do aviso, que introduz o novo comportamento toda vez que um método ou construtor for executado em uma classe alvo. O novo comportamento é a execução do método **traceEntry()** da classe TraceManager. Esse método recebe a assinatura do bloco (método ou construtor) que acabou de ser executado. Nota-se também que existe a possibilidade de desabilitar o rastreamento, pois a assinatura de um bloco só será registrada se o método **isTraceEnabled()** retornar *true*.

Após o projeto do *plug-in* de controle, o aspecto será atualizado para não monitorar blocos do mesmo. Os aspectos finais dos programas monitorados podem ser visualizados no Anexo A.

5.2 Análise de blocos monitorados

Durante a análise e projeto do aspecto, foi feita uma primeira definição de quais blocos seriam monitorados. Esses blocos pertencem aos dois programas que auxiliam a criação e execução de testes automatizados: TAFStudio e TAFPlus2. Informações detalhadas sobre as funcionalidades e a arquitetura de cada um desses programas podem ser encontradas nas seções 3.6.2 e 3.6.3.

Após analisar os componentes do TAFPlus2, decidiu-se eliminar o rastreamento de blocos de eventos. Esses blocos são executados na maior parte das execuções, logo, poderiam ter

```
public aspect Tracing
{
    /**
     * The trace manager.
     */
    private TraceManager traceManager = TraceManager.getInstance();

    /**
     * The scope of this aspect.
     */
    pointcut classeAlvo() : within(com.motorola..*);

    /**
     * A point cut grouping some join points.
     */
    pointcut execucaoDeMetodo(Object t) : classeAlvo()
        && execution(* *(..))
        && !execution(* get*(..))
        && !execution(* set*(..))
        && target(t);

    /**
     * A point cut grouping some join points.
     */
    pointcut execucaoDeConstrutor(Object t) : classeAlvo()
        && execution(new(..))
        && target(t);

    /**
     * Before the execution flow of a method or constructor verify if trace
     * is enabled, and if is trace this entry (Signature).
     */
    before(Object t) : execucaoDeMetodo(t) || execucaoDeConstrutor(t) {
        if (traceManager.isTraceEnabled())
        {
            traceManager.traceEntry(thisJoinPoint.getSignature());
        }
    }
}
```

Figura 5.7: Implementação básica do aspecto de rastreamento

um *spectrum* elevado, gerando diagnósticos incorretos. Além disso, esses blocos dificilmente apresentarão erros pois tem uma complexidade de código baixa. O aspecto do TAFStudio não teve nenhuma modificação nesta fase e apenas será modificado após o projeto do *plug-in*. Após o projeto do *plug-in*, precisa-se atualizar também o aspecto do TAFPlus2.

5.3 Análise, Projeto e Implementação do *plug-in*

Na seção 5.1 foi demonstrada a análise, projeto e implementação do aspecto de rastreamento. Além do aspecto, é necessário analisar, projetar e implementar outras classes para finalizar a ferramenta. Essas classes serão implementadas em um projeto de *plug-in* (programa que adicionará novas funções aos programas existentes) do Eclipse (ECLIPSE, 2009a). Para começar a análise do *plug-in*, foram elaborados alguns artefatos utilizando UML. O primeiro artefato elaborado foi um **diagrama conceitual** para compreender conceitos fundamentais. Esse diagrama será refinado e transformado em um diagrama de classes, à medida que novos artefatos forem elaborados. O diagrama conceitual pode ser conferido na figura 5.8.

Nesse diagrama pode-se observar as classes principais que serão necessárias para o *plug-in* de controle e emissão de diagnósticos. A classe **TraceManager** é o componente mais importante, pois ela controla o rastreamento e é a interface de comunicação com os aspectos de rastreamento. Nesse diagrama o aspecto de rastreamento está sendo representando pelo nome **Tracing**. O desenvolvedor pode ativar ações na interface que modificam o estado (executando, pausado, finalizado, etc) do rastreamento. Essas ações serão ativadas por diversas **Actions** que se comunicam com a classe TraceManager. A classe **SpectrumMatrix** é responsável por armazenar a matriz de *spectrum*, contendo os casos de teste e blocos que foram executados. Essa matriz contém também o resultado de cada caso de teste. Com os dados da matriz de *spectrum*, pode-se calcular um diagnóstico através de um coeficiente de similaridade. Essa é a responsabilidade da classe **SimilarityCoefficientJob**. Após calcular o diagnóstico, a classe **SpectrumTextWriter** emitirá o resultado em um arquivo no formato texto. Finalizando, a classe **TestCase** encapsula informações de um caso de teste manual.

Após a elaboração do diagrama conceitual, foi elaborado o caso de uso de aplicação **Diagnosticar Casos de Testes**. Esse caso de uso demonstra as ações de um desenvolvedor ao diagnosticar faltas executando alguns casos de teste. Para compreender os passos desse caso de uso, foi elaborado um **diagrama de atividades** também em alto nível, sem entrar em detalhes de programação. Segundo (SILVA, 2007), um diagrama de atividades pode ser utilizado para descrever algum comportamento do sistema. Essa descrição pode ser elaborada em baixo ou

alto nível de detalhamento.

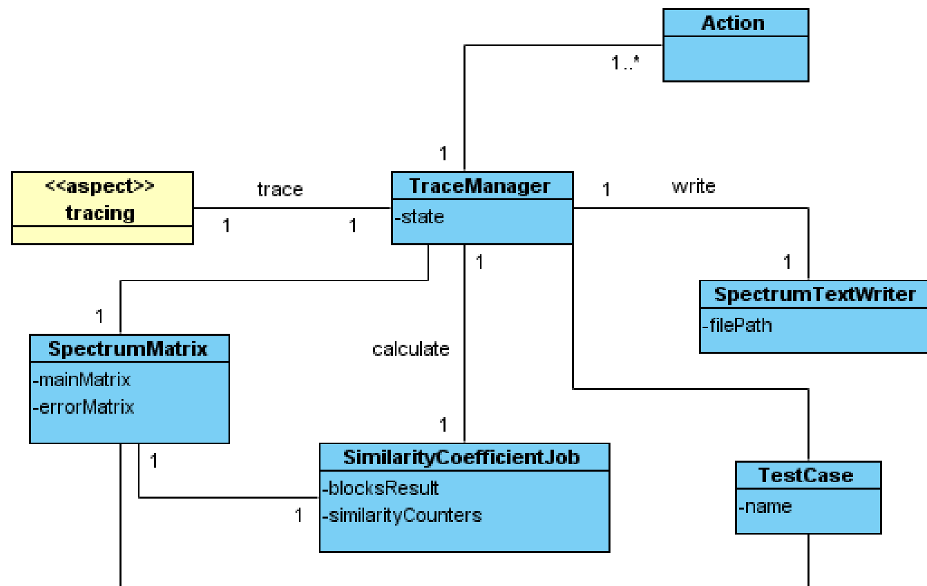


Figura 5.8: Diagrama Conceitual

O diagrama de casos de uso do sistema é mostrado na figura 5.9. O diagrama de atividades demonstrando os passos do caso de uso Diagnosticar Casos de Teste pode ser visualizado na figura 5.10. Através desse último diagrama, mostra-se como funcionará o novo comportamento através de ações e transições. O ponto preto com preenchimento representa o início da execução, e o ponto preto com uma parte sem preenchimento representa o final da execução. Foram colocadas algumas notas para fornecer maiores informações sobre algumas ações. Os losangos representam diferentes caminhos que podem ser tomados dependendo de alguma condição. Por exemplo, quando a ação “Verifique o resultado do passo” estiver sendo executada, pode-se executar a ação “Informe o resultado” se algum passo falhar, ou se não existir mais nenhum passo a ser executado. Mas, se o último passo não falhar e existirem mais passos, a ação executada será “Execute um passo de caso de teste manualmente”. Esse diagrama de atividades descreve o comportamento de um usuário ao diagnosticar casos de teste utilizando a ferramenta proposta neste trabalho.

Para adicionar um novo comportamento ao caso de uso **Diagnosticar Casos de Teste**, foi elaborado outro caso de uso denominado **Controlar Rastreamento**. Esse caso de uso estende

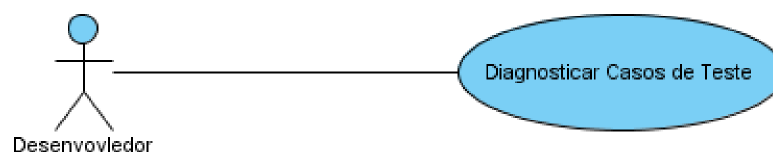


Figura 5.9: Diagrama inicial de casos de uso do *plug-in*

o caso de uso principal e controla as ações de iniciar, pausar e finalizar o rastreamento de uma execução de caso de teste manual. O diagrama de casos de uso do sistema atualizado pode ser conferido na figura 5.11. O diagrama de atividades refinando o novo caso de uso é mostrado na figura 5.12. Nota-se que agora foi colocada também uma dependência entre o caso de uso de aplicação Diagnosticar Casos de Teste e o caso de uso de infra-estrutura Gerenciar Rastrea-

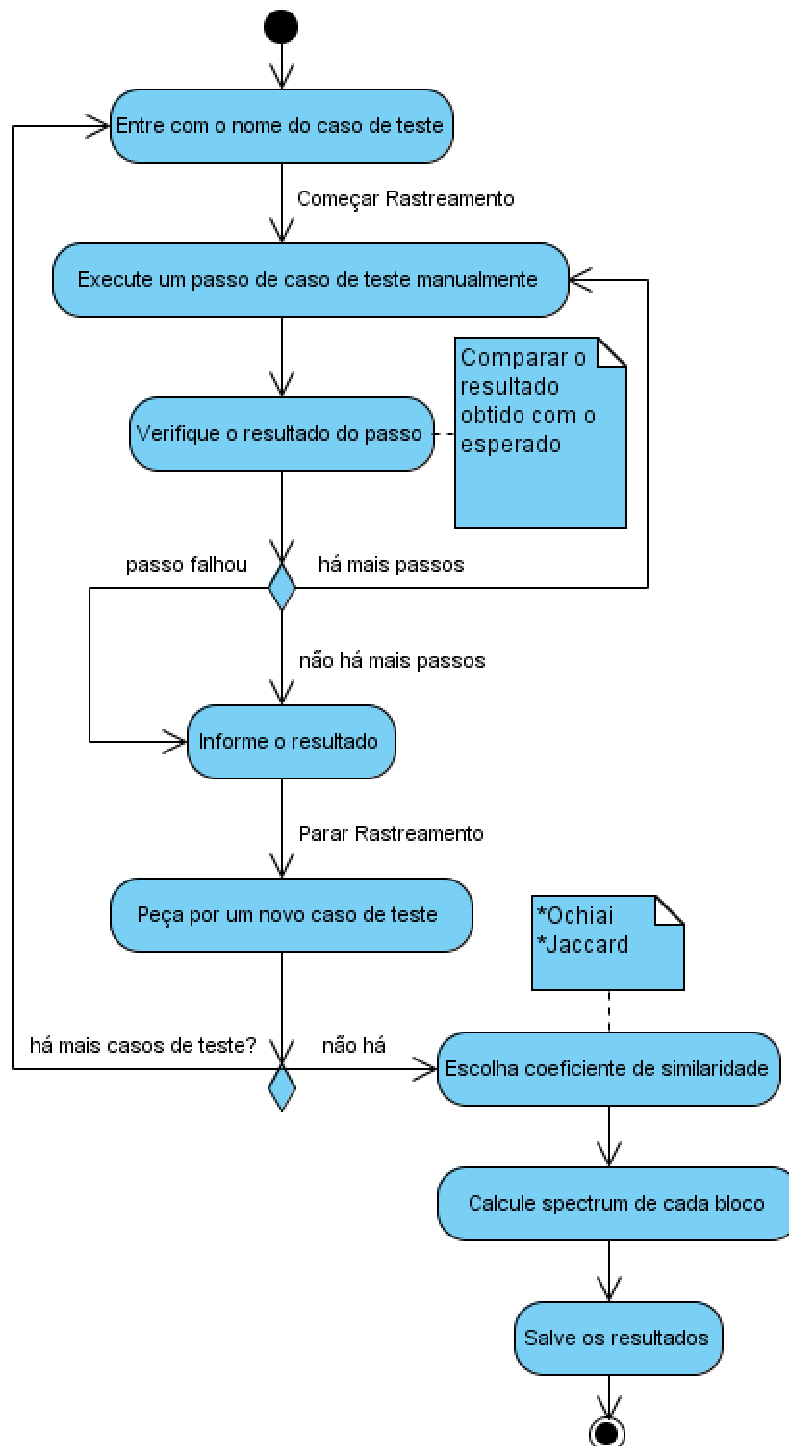


Figura 5.10: Diagrama de atividades do caso de uso Diagnosticar Casos de Teste

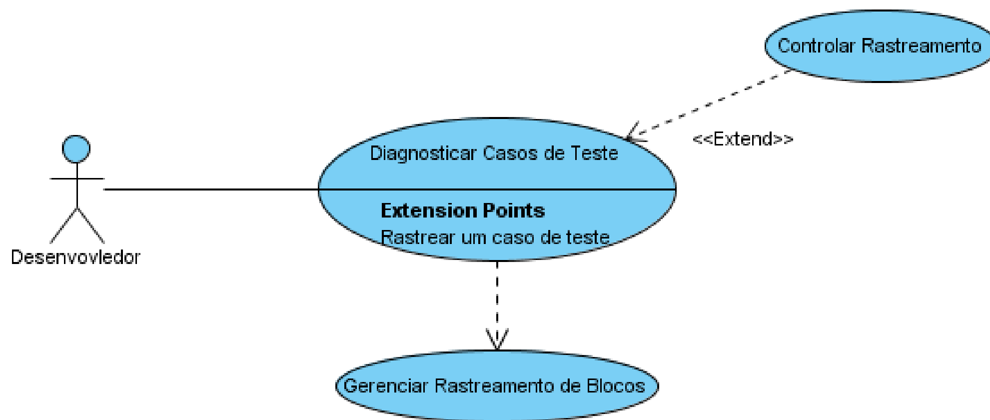


Figura 5.11: Diagrama final de casos de uso do *plug-in*

mento de Blocos, proveniente da análise e projeto do aspecto. Essa dependência demonstra que o caso de uso Diagnosticar Casos de Teste não pode executar se o caso de uso Gerenciar Rastreamento de Blocos não estiver executando também.

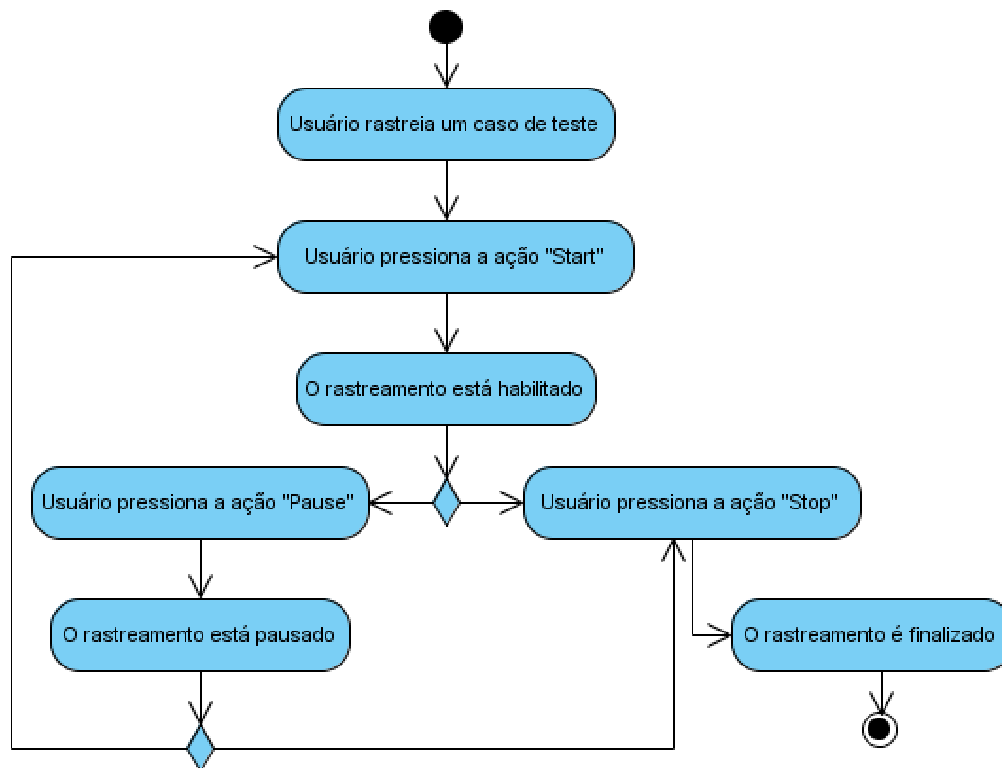


Figura 5.12: Diagrama de atividades do caso de uso Controlar Rastreamento

Para melhor compreensão da interação entre os objetos que participam de um caso de uso, foram elaborados dois diagramas de seqüência para refinar ainda mais cada caso de uso do sistema. Um **diagrama de seqüência** dá uma ênfase maior ao tempo (SILVA, 2007). Além disso, ele é um artefato que ajuda a capturar novos conceitos, atributos, métodos e relacionamentos

entre as classes do programa. Assim, ao finalizar o diagrama de seqüência pode-se refinar o diagrama conceitual, transformando-o em um diagrama de classes de projeto. Devido ao tamanho dos diagramas de seqüência, eles não são mostrados nessa seção. Os mesmos estão disponíveis no Anexo B.

Após refinar os casos de uso do sistema, atualiza-se o diagrama conceitual com os novos conceitos, atributos, relacionamentos e métodos descobertos. Assim, obtém-se o **diagrama de classes**. Segundo (SILVA, 2007), o diagrama de classes deve ser utilizado para representar os elementos de um programa orientado a objetos em tempo de desenvolvimento e não em tempo de execução. Ele é um diagrama estático e tem a capacidade de representar as classes com seus atributos e métodos e também os relacionamentos entre elas. O diagrama de classes de projeto encontra-se no Anexo B juntamente com os diagramas de seqüência do *plug-in*.

Baseando-se nos artefatos elaborados, inicia-se a programação. A programação será guiada principalmente pelos diagramas de sequência, pois eles demonstram a execução temporal de métodos das classes modeladas. Nesta fase, cria-se o *plug-in* de controle do rastreamento. Para criar esse *plug-in*, utilizou-se a IDE **Eclipse** (ECLIPSE, 2009a) para desenvolvimento, e o livro **Eclipse: Building Commercial-Quality Plug-ins** (CLAYBERG; RUBEL, 2004) como principal referência.

O primeiro passo da implementação foi a criação de um projeto de *plug-in* no Eclipse. A esse projeto foi dada a identificação de **spectrum**. Esse projeto representa o *plug-in* de controle do rastreamento e emissão de diagnóstico dos blocos. Ao criar o projeto utilizando um guia da IDE Eclipse (*wizard*), foram criados automaticamente alguns arquivos para o novo *plug-in*. Dois arquivos muito importantes são: **MANIFEST.MF** e **plugin.xml**. Esses arquivos indicam como o *plug-in* se relaciona com o restante do sistema. Algumas informações importantes contidas nesses arquivos:

- **Versão, nome e identificador do *plug-in***: Essas informações identificam o *plug-in* unicamente em um sistema com outros *plug-ins*. Ao *plug-in* de controle foi dado o nome de **Spectrum Fault Localization**, versão **1.0.0** e o seu identificador único é: **spectrum**.

- **Dependências**: Representam quais *plug-ins* são visíveis ao *plug-in* que está sendo criado. É importante ressaltar que não pode existir dependência em ciclo entre dois *plug-ins*. O *plug-in* deste trabalho necessita do pacote de *runtime* do AspectJ para capturar informações dos blocos monitorados. Necessita-se também de bibliotecas do Eclipse de execução e de interface.

- **Extensões e Pontos de Extensão**: Um projeto de *plug-in* pode declarar extensões, isto

é, estender um determinado comportamento de um outro plug-in. Esse comportamento executará em pontos de extensão pré definidos pelo *plug-in* que está sendo extendido. Nenhuma extensão foi declarada e também nenhum ponto de extensão foi realizado pelo *plug-in* de controle.

- **Execução**: Nesta seção especifica-se quais pacotes do *plug-in* que está sendo implementado estarão disponíveis a outros *plug-ins* em tempo de execução. O pacote **spectrum** foi disponibilizado a outros *plug-ins*. Esse pacote contém a classe TraceManager que será acessada pelas classes dos programas monitorados após a composição com os aspectos de rastreamentos.

Após configurar os arquivos de manifesto do *plug-in*, programaram-se as classes e métodos modelados nos diagramas de classes, atividades e seqüência das fases anteriores. Como dito anteriormente, o diagrama de seqüência é muito importante durante a programação, pois serve como um guia para o desenvolvimento dos algoritmos, e a sua ênfase no tempo ajuda a definir a ordem de chamada de métodos. Na programação foram utilizados alguns padrões de projeto como ***Factory*** e ***Singleton***. Um padrão de projeto é uma solução reusável para um problema recorrente do projeto de sistemas. Para maiores informações sobre padrões de projeto recomenda-se o livro **Design Patterns: Elements of Reusable Object-Oriented Software** (GAMMA et al., 1994). O código completo do *plug-in* de controle e emissão de diagnóstico pode ser visualizado no Anexo C.

Nesta seção, destacou-se a análise, projeto e implementação do *plug-in* para o controle e emissão de diagnósticos. A próxima seção explicará como foi feita a composição entre os aspectos e os programas. Após a composição, a ferramenta já poderá ser utilizada para se localizar e diagnosticar faltas.

5.4 Composição

Com o *plug-in* de controle implementado, precisa-se atualizar o aspecto para não monitorar blocos do *plug-in* relacionados com ações de interface do usuário. Para isso, retira-se o monitoramento de classes de ações de interface do usuário (*actions*) e de janelas (*dialogs*) ativadas por essas ações de interface. Essa modificação vale para os aspectos dos dois programas monitorados. Os aspectos finais podem ser visualizados no Anexo A.

Com os aspectos atualizados, é necessário compor os mesmos com os programas alvo para começar a registrar a execução de blocos. Cada programa alvo é composto por vários projetos

de *plug-in*. Logo, deve-se executar a composição com o aspecto para cada projeto que se deseja monitorar. Nesta seção é dado um exemplo da composição de um projeto do TAFStudio com o aspecto de rastreamento.

O projeto deste exemplo é o **TAFStudio.application**. Cria-se o pacote ***tracing*** dentro desse projeto e insere-se o aspecto do TAFStudio dentro do pacote recém criado. Para executar a composição, precisa-se fazer o *weaving* das classes existentes com o aspecto de rastreamento. Para isso, converte-se o nosso projeto para um projeto que suporta AspectJ. O Eclipse automaticamente gera os arquivos compilados com o novo comportamento. Isso só é possível devido ao projeto **AspectJ Development Tools (AJDT)** que traz algumas facilidades na programação com aspectos utilizando Eclipse. A grande vantagem de executar a composição dessa forma é a possibilidade de visualizar quais blocos estão sendo capturados pelo aspecto. O procedimento de composição é análogo para projetos do TAFPlus2.

Nesta monografia, a composição entre aspecto e projeto de um programa foi executada para cada projeto de *plug-in* a ser monitorado. No TAFStudio estão sendo monitorados dois projetos. Já no TAFPlus2 temos o monitoramento de dez projetos, pois este programa tem um número maior de classes.

Existe também a possibilidade de realizar a composição utilizando uma ferramenta de construção baseada em Java, denominada **Ant** (APACHE, 2009). A utilização de Ant é muito mais simples do que qualquer outra ferramenta de construção (*make*, *gnumake*, *nmake*, etc.). Ao invés de escrever comandos para serem executados em um terminal, criam-se arquivos de configuração baseados em **Extensible Markup Language (XML)** (CONSORTIUM, 2009) onde várias tarefas são executadas. A execução dessas tarefas resultará no programa final.

Para compor aspectos utilizando Ant, é proposta uma tarefa específica, denominada **Ajc-Task (IAJC)** (ECLIPSE, 2009b). Essa tarefa utiliza o compilador da linguagem AspectJ para compor os aspectos com o programa original. Na definição da tarefa são informados os aspectos e os arquivos fonte ou binário do programa original. Assim, é realizada a composição entre o aspecto e o programa, resultando em um novo programa com o comportamento inserido. As vantagens de se utilizar Ant é a necessidade de apenas uma composição por programa e a facilidade de configuração da tarefa que faz a composição.

Nesta monografia não se utiliza Ant para compor os aspectos, pois o *plug-in* AJDT para Eclipse faz a composição automaticamente e permite visualizar quais partes do programa são impactadas por um aspecto. Se fosse proposto lançar uma nova versão de um dos programas com o rastreamento habilitado, seria necessária a utilização de Ant para composição.

Após a composição dos aspectos com os programas alvo (introdução da funcionalidade de rastreamento da execução de blocos) e a implementação do *plug-in* de controle, tem-se a ferramenta finalizada. Neste momento já pode-se localizar e diagnosticar faltas do TAFStudio ou TAFPlus2. O próximo capítulo descreve o ciclo de execução de testes manuais, e avalia os resultados obtidos após utilizar a ferramenta para diagnosticar algumas faltas introduzidas propositadamente.

6 Experimentação e Resultados

Com a ferramenta para localização e diagnóstico de faltas implementada, entra-se numa nova etapa para avaliar se ela emite resultados confiáveis. Neste capítulo, serão criadas algumas suítes de casos de teste na seção 6.1. Na seção 6.2 serão introduzidas faltas em alguns blocos dos programas monitorados. Então, na seção 6.3, serão executadas as suítes de casos de teste com o rastreamento habilitado e será calculado o *spectrum* para cada bloco. Na seção 6.4, avalia-se a ferramenta quanto à emissão de diagnósticos e a outros quatro fatores que podem influenciar na precisão do diagnóstico. A última seção mostra os resultados obtidos para os dois programas monitorados: TAFPlus2 e TAFStudio.

6.1 Criação de Testes

Os testes criados nesta seção baseiam-se na técnica caixa preta. Como explicado no capítulo 3, esses testes não consideram os detalhes internos de implementação. Cada caso de teste pode ter um número limitado de passos e cada passo pode ou não ter um resultado esperado. Ao executar um teste, se o resultado obtido de um passo estiver de acordo com o resultado esperado do mesmo, então o próximo passo será executado. No momento que o resultado obtido de um passo diferir do esperado, o caso de teste já é declarado como falho. Se todos os resultados obtidos estiverem de acordo com os esperados, então o teste passou.

Nesta monografia, existe uma planilha de casos de teste manuais para cada programa monitorado. O TAFPlus2 já tinha uma planilha com alguns casos de teste que serão aproveitados e modificados. Além desses casos de teste, novos serão criados, pois alguns testes da planilha anterior estão desatualizados. Para o TAFStudio tornou-se necessário criar uma nova planilha de testes. Esses testes cobrem diversas funcionalidades do sistema, desde casos normais de uso até entradas inesperadas pelo usuário.

A tabela 6.1 mostra um exemplo de um caso de teste manual para o programa de execução de testes automatizados TAFPlus2. Esse programa foi explicado em detalhes na seção 3.6.2.

O caso de teste da tabela 6.1 está testando a funcionalidade de ações automáticas que serão executadas após uma execução. Observa-se nesse caso de teste que existe uma pré-condição exigindo um perfil de execução configurado antes que o teste possa ser executado. Essa pré-condição é definida implicitamente no primeiro passo. Nota-se também que todos os passos tem resultados esperados. Os testes criados nesta monografia baseiam-se no modelo de casos de teste mostrado nesta tabela.

Descrição do teste	Passo	Descrição do passo	Resultados esperados
Editar ações automáticas.	1	Tendo um perfil de execução corretamente configurado, o usuário cria uma nova configuração de execução.	Na aba 'Test List', nenhum grupo de teste está disponível. Na aba 'Phones config', não existe nenhum slot de telefone adicionado. Na aba 'TAF Configuration', as seguintes tags estão configuradas com o valor '1': TAF Developer Log Level, TAF Debug Log Level and PTF Log Level. A ação 'Starts Execution' está desabilitada. Na barra "Execution configuration", o usuário pode visualizar o alerta vermelho que diz que existem 2 erros, e se o mouse for passado pela mensagem, os erros são especificados em uma tooltip. Os erros são que uma execução não tem nenhum grupo de teste habilitado e nenhum slot de telefone habilitado.
	2	O usuário vai para aba 'Automatic Actions'.	O destino dos campos 'Export XLS File' e 'Export Unified Logs' está vazio.
	3	Selecione um arquivo como destino do campo 'Export XLS File'.	O arquivo aparece no campo de destino 'Export XLS File'.
	4	Selecione um diretório como destino do campo 'Export Unified Logs'.	O diretório aparece no campo de destino 'Export Unified Logs'.

Tabela 6.1: Um exemplo de caso de teste manual para o TAFPlus2

Após criar os casos de teste utilizando a estratégia caixa preta, decidiu-se agrupar os testes que têm um comportamento semelhante. Cada conjunto de casos de teste é uma suíte de testes. O caso de teste mostrado nesta seção faz parte da suíte de testes **Managing Execution Config-**

urations juntamente com outros oito casos de teste. Esses testes têm como objetivo modificar configurações de uma execução. As suítes agrupam os casos de teste que testam funcionalidades parecidas. As suítes de testes completas para o TAFStudio e TAFPlus2 podem ser conferidas no Anexo D.

6.2 Introdução de Faltas

Com o intuito de verificar se a ferramenta emite diagnósticos precisos, foram introduzidas faltas em determinados blocos de código. Esses blocos são executados pelas suítes de testes. O objetivo é que cada suíte de testes execute um bloco com falta. Ao se tratar de casos de teste, geralmente se tem um pequeno número de faltas entre duas versões de um programa. Esse comportamento foi observado no ambiente de testes da Motorola. Logo, a idéia de inserir uma falta por suíte de testes é aceitável. Essa idéia deve refletir a realidade quando a ferramenta for executada para localizar e diagnosticar defeitos que não foram inseridos propositavelmente.

Um exemplo de uma falta em um bloco do TAFPlus2 é o não fornecimento do valor quando um novo atributo (*tag*) for adicionado em um arquivo de configuração do telefone. Essa falta será inserida no método **addTag()** da classe **TagConfigEditor**. Esse comportamento fará com que alguns casos de teste da suíte executada falhem. Após executar manualmente todos os casos de teste dessa suíte, tem-se um diagnóstico de quais blocos são os mais prováveis de estarem com falta. Esse diagnóstico é o resultado da execução da suíte de testes. Como se sabe onde a falta está, pode-se verificar se o bloco com falta está dentre os blocos com maior probabilidade de estarem com defeito. A figura 6.1 mostra o código do bloco sem a falta. A figura 6.2 já mostra o código com a falta destacada dentro de um retângulo.

Um caso particular ocorreu na introdução de faltas no TAFPlus2. Quando um novo perfil de execução é adicionado, não é verificado se já existe um perfil com o mesmo nome. Nesse caso, como a falta não foi introduzida propositalmente, executou-se a suíte de testes de perfis de execução e obteve-se como resultado alguns blocos com maior probabilidade de estarem com falta. Ao analisar os mesmos, concluiu-se que o diagnóstico foi preciso e que algumas verificações não estavam sendo feitas nesses blocos. Assim, encontrou-se um defeito do TAFPlus2.

No total foram introduzidas oito faltas no TAFStudio e oito faltas no TAFPlus2. Estas faltas foram distribuídas em blocos que têm em funções diferentes. Na próxima seção cada falta será associada a uma suíte de testes. Assim, testa-se uma boa parte das funcionalidades dos programas em análise. As planilhas com todas as faltas introduzidas podem ser visualizadas

```java
/**
 * Adds a new configuration tag defined by the user.
 */
private void addTag()
{
    AddTagValueDialog addDialog = new AddTagValueDialog(parentShell, "Edit a tag",
        "Edit a tag to be added within the configuration.");

    // If there is no already imported config file and a tag needs to be
    // added create a new old config.
    if (this.oldConfig == null)
    {
        OldConfig config = new OldConfig(this.configType, this.phoneId, "");
        this.slot.insertConfigData(config);
        this.slot.saveConfigData();
    }

    int out = addDialog.open();
    if (out == Window.OK)
    {
        String tag = addDialog.getTagName();
        String value = addDialog.getTagValue();
        if (oldConfig != null)
        {
            this.oldConfig.setTagValue(true, tag, value);
        }
        else
        {
            TAFPlus2.log.showErrorDialog(this, "Unable to add the tag "
                + "within the configuration.", null);
        }
    }
}
```

Figura 6.1: Bloco TagConfigEditor.addTag() sem falta

```java
/**
 * Adds a new configuration tag defined by the user.
 */
private void addTag()
{
    AddTagValueDialog addDialog = new AddTagValueDialog(parentShell, "Edit a tag",
        "Edit a tag to be added within the configuration.");

    // If there is no already imported config file and a tag needs to be
    // added create a new old config.
    if (this.oldConfig == null)
    {
        OldConfig config = new OldConfig(this.configType, this.phoneId, "");
        this.slot.insertConfigData(config);
        this.slot.saveConfigData();
    }

    int out = addDialog.open();
    if (out == Window.OK)
    {
        String tag = addDialog.getTagName();
        String value = addDialog.getTagValue();
        if (oldConfig != null)
        {
            this.oldConfig.setTagValue(true, tag, "");
        }
        else
        {
            TAFPlus2.log.showErrorDialog(this, "Unable to add the tag "
                + "within the configuration.", null);
        }
    }
}
```

Figura 6.2: Bloco TagConfigEditor.addTag() com falta

no Anexo E.

6.3 Execução de Testes

Com as suítes criadas e os defeitos inseridos, parte-se para a execução manual das suítes de casos de teste com a ferramenta habilitada, a fim de diagnosticar possíveis blocos com falta. Para cada falta verifica-se qual suíte de testes é a mais impactada pela mesma. Assim, associa-se a falta a uma suíte de testes e executa-se essa suíte com a falta habilitada. Ao final, calcula-se o *spectrum* dessa suíte utilizando os dois coeficientes (Ochiai e Jaccard). Obtêm-se como resultado (diagnóstico) os blocos com maior probabilidade de estarem com falta. Então, pode-se verificar se o bloco com falta foi reportado pela ferramenta como um dos candidatos a estarem com defeito. Se o bloco tiver sido reportado então a ferramenta está funcionando corretamente.

Esse ciclo de execução é repetido para cada falta dos dois programas monitorados. A figura 6.3 demonstra o ciclo de execução manual de suítes de casos de teste. Ao final, obtêm-se uma grande quantidade de resultados para avaliação. Esses resultados são os diagnósticos emitidos pela ferramenta. Em cima desses resultados, além de verificar se a ferramenta emite diagnósticos precisos, avalia-se também quais fatores influenciam nessa precisão. Essa avaliação é descrita em detalhes na próxima seção.

6.4 Avaliação

Nesta seção, primeiramente avalia-se um resultado da execução de uma suíte de testes com uma falta habilitada. Após essa avaliação, avaliam-se todos os resultados obtidos neste trabalho com relação a quatro fatores: **coeficiente de similaridade**, **suíte de testes**, **programa alvo** e **granularidade de blocos**. Será verificado se esses fatores podem afetar a precisão da técnica.

6.4.1 Avaliação da execução de uma suíte de testes

Nesta seção, avalia-se o resultado emitido pela ferramenta após executar manualmente uma suíte de casos de teste com uma falta habilitada. Primeiramente foi selecionada uma falta a ser introduzida no programa TAFStudio. A falta foi introduzida no método **pushGroup(StepGroup newCurrentGroup)** da classe **RecordingSession**. Esse método é o bloco com defeito. O defeito faz com que ao inserir um novo grupo de passos, ele não seja inserido na lista de passos visualizados pelo testador. Na figura 6.4, pode-se observar o trecho de código

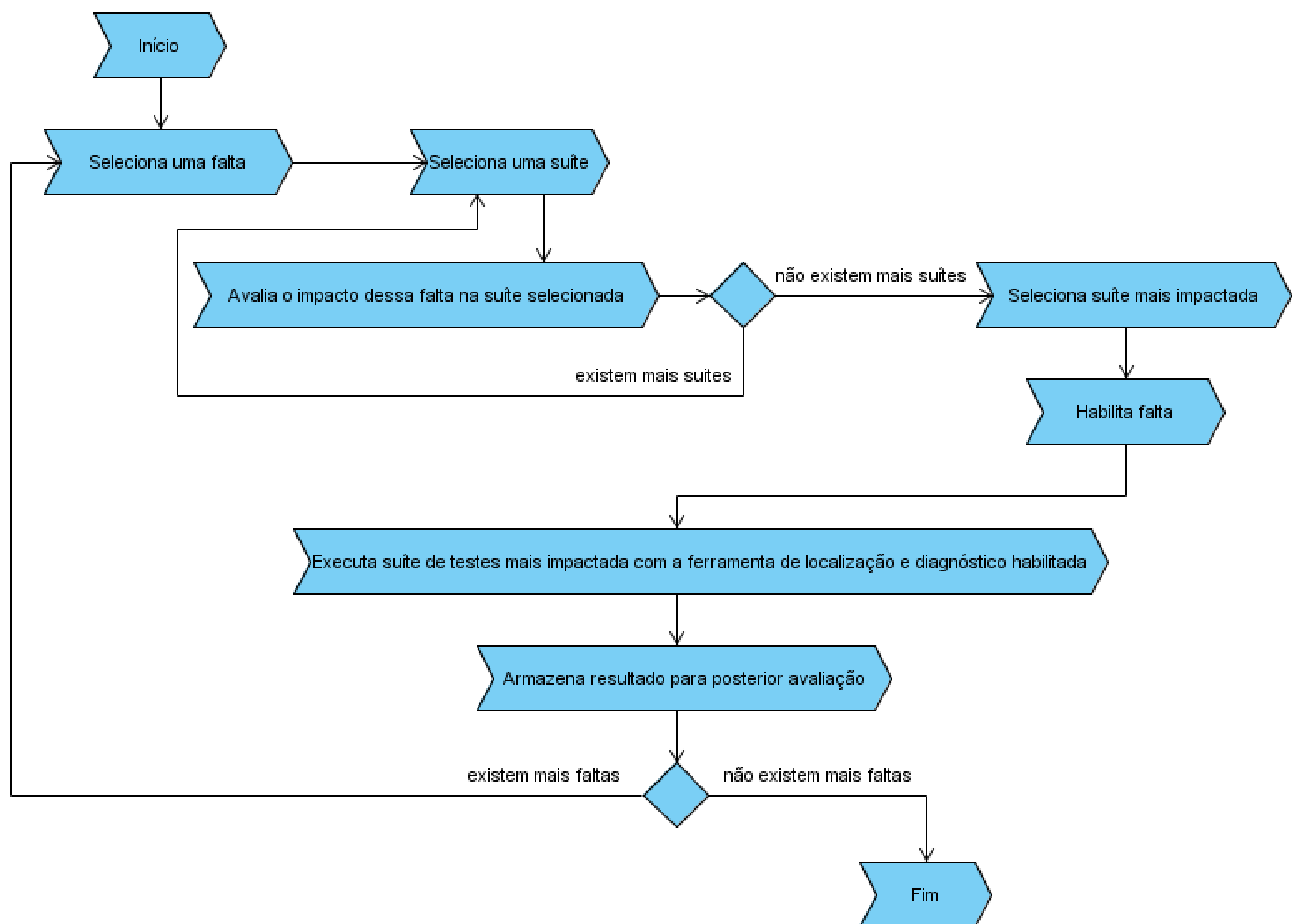


Figura 6.3: Ciclo de execução manual de suítes de testes

comentado, correspondente à falta inserida.

```
/**
* Inserts a new group and sets it as the current group for this recording session.
*
* @param newCurrentGroup The group to be set as the current one.
*/
public void pushGroup(StepGroup newCurrentGroup)
{
    newCurrentGroup.setParent(currentGroup);

    // FIXME Não inseriu o grupo na lista de passos.
    // insertStepOnList(newCurrentGroup);

    currentGroup = newCurrentGroup;
    this.sessionUpdated();
}
```

Figura 6.4: Bloco RecordingSession.pushGroup() com falta

Após analisar a falta com relação às suítes de testes criadas, concluiu-se que a suíte mais impactada pela falta é a suíte **Recording Test Cases**. Essa suíte testa funcionalidades referentes à gravação de passos de um caso de teste, e contêm onze casos de teste. Prosseguindo, habilitou-

se a falta e foram executados todos casos de teste da suíte selecionada com a ferramenta de localização e diagnóstico de faltas ativada. Ao final, três casos de teste falharam (53, 55 e 57) e os outros passaram. Então calculou-se o resultado (diagnóstico) para os dois coeficientes de similaridade: Ochiai e Jaccard. Uma parte do resultado obtido utilizando o coeficiente de Ochiai pode ser conferido na tabela 6.2. Dos 125 blocos reportados no diagnóstico, os 23 blocos com maior *spectrum* (maiores candidatos a estarem com falta) estão sendo mostrados na tabela.

Spectrum	Bloco com defeito
1.0	public void com motorola tafstudio ui action testcase CreateGroupAction run()
1.0	**public void com motorola tafstudio tcdevelopment RecordingSession pushGroup(com motorola tafstudio tcdevelopment steps StepGroup)**
1.0	public java lang String com motorola rcputils ui dialogs validators EmptyStringValidator isValid(java lang String)
1.0	public com motorola tafstudio tcdevelopment steps StepGroup(java lang String)
1.0	public com motorola tafstudio tcdevelopment steps StepGroup(com motorola tafstudio tcdevelopment steps StepGroup,java lang String)
1.0	public com motorola rcputils ui dialogs validators EmptyStringValidator()
0.707106781186547	public void com motorola tafstudio rendering StatusIcon accept(com motorola tafstudio visitors ScreenItemVisitor)
0.707106781186547	public void com motorola tafstudio rendering ScrollBar accept(com motorola tafstudio visitors ScreenItemVisitor)
0.707106781186547	public void com motorola tafstudio rendering ScreenRenderer visitStatusIcon(com motorola tafstudio rendering StatusIcon)
0.707106781186547	public void com motorola tafstudio rendering ScreenRenderer visitScrollBar(com motorola tafstudio rendering ScrollBar)
0.707106781186547	public com motorola tafstudio rendering StatusIcon(int,int,int)
0.707106781186547	public com motorola tafstudio rendering ScrollBar(int,java awt Rectangle)
0.654653670707977	public void com motorola tafstudio rendering TextItem accept(com motorola tafstudio visitors ScreenItemVisitor)
0.654653670707977	public void com motorola tafstudio rendering SynergyIcon accept(com motorola tafstudio visitors ScreenItemVisitor)
0.654653670707977	public void com motorola tafstudio rendering ScreenRenderer visitTextItem(com motorola tafstudio rendering TextItem)
0.654653670707977	public void com motorola tafstudio rendering ScreenRenderer visitSynergyIcon(com motorola tafstudio rendering SynergyIcon)
0.654653670707977	public void com motorola tafstudio rendering ScreenRenderer visitImageItem(com motorola tafstudio rendering ImageItem)
0.654653670707977	public void com motorola tafstudio rendering ScreenRenderer drawItem(com motorola tafstudio rendering ScreenItem,boolean)
0.654653670707977	public void com motorola tafstudio rendering PhoneDisplay addItem(com motorola tafstudio rendering ScreenItem)
0.654653670707977	public org eclipse swt graphics Color com motorola tafstudio rendering ScreenItem$Color toSWTColor()
0.654653670707977	public org eclipse swt graphics Color com motorola rcputils ui resources Colors clr()
0.654653670707977	public java lang String com motorola tafstudio tcdevelopment steps ScreenStep toString()
0.654653670707977	public com motorola tafstudio tcdevelopment steps ScreenStep(com motorola tafstudio rendering PhoneDisplay,int,long)
Total de blocos:	**125**

Tabela 6.2: Diagnóstico parcial da execução de uma suíte de testes

Observa-se no diagnóstico da tabela 6.2 que o bloco com defeito está entre os seis maiores

candidatos a estarem com falta (o bloco está destacado em negrito). Logo, dentre 125 blocos suspeitos, precisa-se inspecionar apenas 6 blocos para se localizar o defeito, uma porcentagem de 4,80% de blocos para inspeção manual. O procedimento descrito nesta seção foi executado para cada falta introduzida. Os resultados (diagnósticos) para cada execução de suíte de testes juntamente com as faltas introduzidas, podem ser visualizados no Anexo E.

6.4.2 Coeficiente de Similaridade

Esta seção avalia o diagnóstico dos dois coeficientes para o cálculo de *spectrum* utilizados nesse trabalho. Os coeficientes avaliados são: **Ochiai e Jaccard**.

Nas execuções manuais de suítes de testes para os dois programas monitorados, não se percebeu grande influência do coeficiente no resultado. Em todos os experimentos, o número de blocos que necessitam ser inspecionados foi o mesmo para ambos coeficientes. No entanto, notou-se que em algumas execuções outros blocos foram classificados com probabilidades diferentes pelos coeficientes. Se o defeito estivesse em algum desses blocos, poderia existir uma diferença no resultado. Logo, precisa-se saber qual coeficiente tem a maior confiabilidade. Segundo (ABREU; ZOETEWEIJ; GEMUND, 2006), o coeficiente de Ochiai apresenta melhores resultados, pois após uma bateria de testes com diversos coeficientes este coeficiente superou os outros em relação a precisão do diagnóstico em até 10%. Assim, optou-se por utilizar o coeficiente de Ochiai como base para avaliação dos resultados.

6.4.3 Suíte de Testes

A suíte de testes pode afetar a precisão do diagnóstico da ferramenta. Essa situação ocorre em ambos os coeficientes. Esse comportamento pode ocorrer quando um bloco com falta não leva a uma falha em determinados testes. Também pode acontecer quando temos mais de uma falta em uma suíte de testes.

No primeiro caso, esse problema poderia ser evitado se todo teste que passasse por um bloco com falta falhasse. Infelizmente, esse tipo de comportamento é impossível de se obter, pois em qualquer programa sempre existirá situações que não levam a um erro e a falta passará despercebida. Para compreender melhor essa situação, a seguir será mostrado um exemplo de uma falta do TAFPlus2 que passa despercebida em determinados testes. Nesse exemplo, esse comportamento resultou em um diagnóstico pouco confiável.

A falta foi inserida no construtor da classe **PhoneSlot**. Essa classe é utilizada para criação de conectores (*slots*) de telefone em branco ou conectores configurados. Normalmente (sem a

falta inserida), quando um conector em branco é criado, ele começa desabilitado. Após inserir a falta, um novo conector sempre será inicializado habilitado.

Para diagnosticar a falta, executa-se uma suíte de testes que executa funcionalidades de configuração de conectores de telefone. Após executar a suíte, apenas dois testes falharam, resultando em um diagnóstico ruim. O grande problema foi que o bloco com falta foi executado em outros testes e não levou a uma falha. Isso acontece pois quando se cria um conector já configurado, por padrão ele deve ser inicializado habilitado. Logo, nesse caso a falta nunca levará a uma falha. Se na suíte de testes não se tivesse testado conectores configurados, provavelmente se teria obtido um diagnóstico melhor. Mas devem-se testar todas as funcionalidades de criação de conectores e criar e editar conectores configurados são uma delas. Assim, o problema acima dificilmente poderá ser resolvido modificando a suíte de testes.

No segundo caso, quando existe mais de uma falta em uma suíte de testes, a ferramenta emite resultados pouco precisos. Isso acontece pois em alguns casos um bloco com falta não é executado em um determinado teste e esse teste falha. Esse comportamento faz com que o *spectrum* desse bloco diminua. Seguindo com a bateria de testes, existem alguns casos que o bloco será executado e o teste falhará, aumentando seu *spectrum*. Ao final, obtém-se um *spectrum* de valor mediano para esse bloco. Esse valor poderia ser maior se não fossem levados em conta os testes falhos que não passaram pelo bloco com falta. Esses testes falharam, pois passaram por outro bloco com falta. Assim, se os dois blocos com falta são executados em testes distintos, obtém-se um *spectrum* de valor mediano para cada um deles. Com isso, a ferramenta não tem tanta certeza em apontar que os dois blocos têm grande probabilidade de estarem com falta. E, ainda mais, se existir um bloco comum entre os testes que falharam, esse bloco será reportado com maior probabilidade de estar com falta.

Devido a esse segundo problema, recomenda-se agrupar os casos de teste que executam funcionalidades semelhantes. Essa é uma forma de prevenir que se tenha mais de uma falta por suíte de testes. Além disso, foi observado no ambiente de testes da Motorola, que entre duas versões de um programa poucas faltas são introduzidas. Logo, a suposição de existir apenas uma falta por suíte de testes é válida.

6.4.4 Programa Alvo

Outro objetivo desta monografia é verificar se a ferramenta emite resultados diferentes dependendo do programa monitorado. No TAFPlus2 e no TAFStudio foram introduzidas oito faltas para cada programa, resultando em dezesseis execuções diferentes. Em média, 20% dos blocos necessitam ser inspecionados após a execução da ferramenta no TAFPlus2. No TAFS-

tudio, em média necessita-se inspecionar 15% dos blocos para encontrar o bloco com falta. Conclui-se então que programas diferentes submetidos à ferramenta têm uma precisão de diagnóstico semelhante. A pequena diferença de precisão deve-se principalmente à diferença entre as suítes de testes para os dois programas. Vale ressaltar que o TAFPlus2 contém quatro vezes mais blocos que o TAFStudio.

6.4.5 Granularidade de Blocos

A primeira implementação da ferramenta tinha uma granularidade maior, utilizando classes como blocos. Foram executadas algumas suítes de testes para verificar se os resultados eram satisfatórios. Ao final, concluiu-se que a ferramenta não emitia bons resultados, pois ela não apontava com precisão em qual parte do bloco suspeito estava a falta. Como o bloco poderia ser uma classe de mais de 1000 linhas, seria necessário percorrer cada uma dessas linhas até encontrar a falta.

Assim, resolveu-se diminuir a granularidade e obter informações mais detalhadas de uma classe. Decidiu-se por considerar um construtor ou método como um bloco. Essa abordagem trouxe um aumento na precisão e maior rapidez na localização da falta, pois um método ou construtor não tem uma grande quantidade de linhas de código. É importante ressaltar que não é possível modificar a granularidade de blocos em tempo de execução. Essa granularidade deve ser modificada no código dos aspectos antes de executar a composição dos aspectos com os programas alvo.

Porém, em alguns casos ainda existem problemas com a granularidade dos blocos. Esses problemas acontecem em trechos de código onde existem diversos blocos de condição (*if*/*else*). A seguir, será mostrado um exemplo mostrando esta situação e propondo uma solução com menor granularidade de blocos a fim de se obter um diagnóstico melhor.

Considere o bloco **update()** da classe **AddGroupDialog** do TAFPlus2. Pode-se observar na figura 6.5 a presença de três blocos de condição nesse método. Esses blocos de condição podem levar a um comportamento indesejado. Vamos supor que seja introduzida uma falta nesse código. A falta será introduzida no trecho de código sublinhado. Agora o botão "Ok" continua habilitado mesmo que o usuário tenha digitado um nome vazio. O código do bloco com falta pode ser conferido na figura 6.6. A falta está dentro de um retângulo. O problema acontece quando dois testes nunca entram no bloco de condição com falta e assim têm como resultado sucesso. Mesmo que outro teste entre no bloco de condição com falta e leve a uma falha, para a ferramenta o *spectrum* do bloco **update()** não será tão alto, pois em dois testes o bloco foi executado e não levou a uma falha.

Se cada bloco de condição fosse considerado um bloco para a ferramenta, o resultado seria muito melhor. Para verificar que isso realmente acontece, modifica-se o código do método **update()**. Inserem-se duas chamadas a novos métodos nos últimos dois blocos de condição. Os métodos adicionados foram: **noProfile(Button okButton)** e **groupNameEmpty(Button okButton)**. O código modificado já com a falta introduzida (dentro do retângulo) pode ser conferido na figura 6.7.

Após executar novamente a suíte de testes, obtém-se como resultado um único bloco suspeito de estar com falta: **groupNameEmpty(Button okButton)** da classe **AddGroupDialog**. Precisa-se percorrer aproximadamente 1% dos blocos até encontrar a falta. O diagnóstico foi perfeito, pois a única vez que esse bloco foi executado levou a uma falha do caso de teste. Quando se executou a suíte antes de extrair os métodos, foi necessário inspecionar aproximadamente 58% dos blocos até encontrar o bloco com falta.

Esse exemplo demonstra que diminuindo a granularidade podem-se obter resultados melhores. Poder-se-ia considerar um bloco de código Java (*statement*) como um bloco para a ferramenta. O grande problema dessa abordagem é que existiria um número muito maior de blocos na memória e uma queda de desempenho da ferramenta.

```
/**
 * Updates the button's enablement.
 */
private void update()
{
    Button okButton = getButton(IDialogConstants.OK_ID);

    if (okButton == null)
    {
        return;
    }

    if ("".equals(groupName.trim()))
    {
        setErrorMessage("The name of the test group must be filled.");
        okButton.setEnabled(false);
        return;
    }

    if (profile == null)
    {
        setErrorMessage("The execution version profile must be filled.");
        okButton.setEnabled(false);
        return;
    }

    setErrorMessage(null);
    okButton.setEnabled(true);
}
```

Figura 6.5: Bloco AddGroupDialog.update() sem falta

```java
/**
 * Updates the button's enablement.
 */
private void update()
{
    Button okButton = getButton(IDialogConstants.OK_ID);

    if (okButton == null)
    {
        return;
    }

    if ("".equals(groupName.trim()))
    {
        setErrorMessage("The name of the test group must be filled.");
        okButton.setEnabled(true);
        return;
    }

    if (profile == null)
    {
        setErrorMessage("The execution version profile must be filled.");
        okButton.setEnabled(false);
        return;
    }

    setErrorMessage(null);
    okButton.setEnabled(true);
}
```

Figura 6.6: Bloco AddGroupDialog.update() com falta

6.5 Resultados

Nesta seção serão mostrados os resultados do ciclo de execução de testes para as 16 faltas introduzidas nos programas monitorados: TAFStudio e TAFPlus2. Cada suíte de testes foi executada com uma falta e a ferramenta habilitada. Ao final, obtém-se o diagnóstico que mostra quais blocos tem a maior probabilidade de estar com falta. A eficiência de um diagnóstico é avaliada em relação à porcentagem de blocos que necessitam ser inspecionados manualmente até se encontrar o bloco defeituoso. A porcentagem de blocos que necessitam ser inspecionados está sendo mostrada no eixo y do gráfico de resultados. O eixo x representa as faltas introduzidas.

Observando a figura 6.8, que representa os resultados para o TAFPlus2, nota-se que os piores resultados ocorreram nas faltas 2 e 3. Isso aconteceu pois muitos testes que executaram o bloco com falta não levaram a falhas. A não detecção de algumas falhas é uma das desvantagens da técnica de localização de faltas baseada no perfil de execução. Em média 20% dos blocos necessitam ser inspecionados manualmente até se encontrar o bloco com defeito no TAFPlus2.

Na figura 6.9, observa-se que os piores resultados aconteceram nas faltas 1 e 3. Como no caso anterior, nessas situações o bloco com falta foi executado mas não levou a falha. No TAFStudio necessita-se inspecionar 15% dos blocos em média.

```
/**
 * Updates the button's enablement.
 */
private void update()
{
    Button okButton = getButton(IDialogConstants.OK_ID);

    if (okButton == null)
    {
        return;
    }

    if ("".equals(groupName.trim()))
    {
        groupNameEmpty(okButton);
    }

    if (profile == null)
    {
        noProfile(okButton);
    }

    setErrorMessage(null);
    okButton.setEnabled(true);
}

private void noProfile(Button okButton)
{
    setErrorMessage("The execution version profile must be filled.");
    okButton.setEnabled(false);
}

private void groupNameEmpty(Button okButton)
{
    setErrorMessage("The name of the test group must be filled.");
    okButton.setEnabled(true);
}
```

Figura 6.7: Bloco AddGroupDialog.update() refatorado e com falta

Os resultados podem ser avaliados como satisfatórios, pois em média necessita-se inspecionar uma quantidade pequenas de blocos até se encontrar o bloco defeituoso. As faltas 2 e 3 do TAFPlus2 tiveram resultados considerados ruins, pois necessita-se inspecionar mais de 55% dos blocos até se encontrar o defeito. As outras faltas do TAFPlus2 tiveram resultados muito bons. Já os resultados do TAFStudio têm uma precisão semelhante entre eles. A falta 1 tem o pior resultado onde mais de 40% dos blocos necessitam ser inspecionados. A falta 3 teve um resultado razoável, pois quase 30% dos blocos necessitam ser inspecionados manualmente.

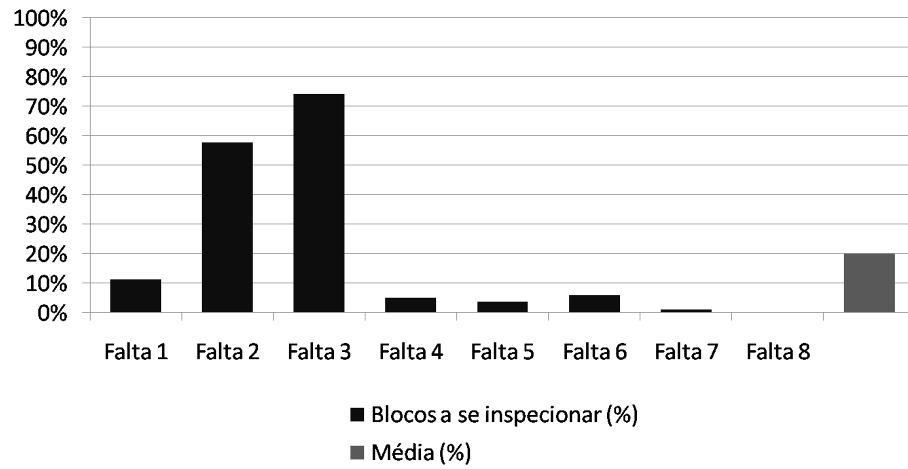


Figura 6.8: Resultados do TAFPlus2

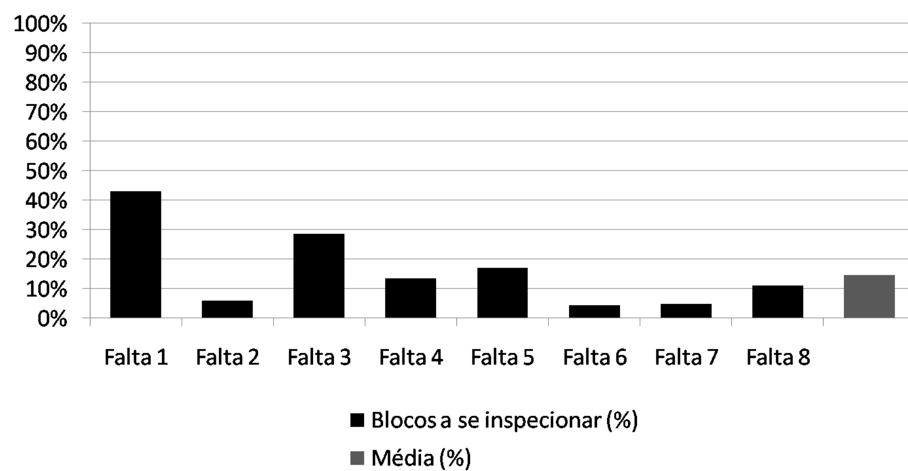


Figura 6.9: Resultados do TAFStudio

7 Conclusão

A ferramenta de localização e diagnóstico de faltas diminui consideravelmente o tempo gasto para se encontrar um bloco com defeito em um programa, pois o diagnóstico emitido pela ferramenta diminui a quantidade de blocos que precisam ser inspecionados manualmente até se encontrar o bloco com falta. Assim, no desenvolvimento de um novo programa, o tempo de entrega ao cliente será menor, possibilitando um menor ciclo de desenvolvimento. Na manutenção de um programa, a execução de suítes de testes com a ferramenta ajudará a localizar erros e diminuirá o esforço do testador até encontrar os blocos com faltas. Partes do programa com defeito serão diagnosticadas e corrigidas, resultando em um programa com maior confiabilidade.

Concluiu-se também que quanto menor a granularidade dos blocos monitorados, maior a eficiência da técnica de diagnóstico. Neste trabalho, foi utilizada uma granularidade média, onde um bloco é um método ou construtor. Assim, existiu um balanceamento entre precisão do diagnóstico e tempo de processamento, pois quanto menor a granularidade, maior o processamento necessário.

O programa monitorado e o coeficiente de similaridade são fatores que quase não influenciam o resultado do diagnóstico. Os dois coeficientes de similaridade geraram resultados muito parecidos. Em relação aos programas monitorados, a precisão média da técnica também é semelhante. A diferença foi de apenas 5 pontos percentuais a mais de blocos que precisam ser inspecionados no TAFPlus2.

Pode-se destacar também que a suíte de testes afeta a precisão de diagnóstico da técnica. Os testes devem executar funções semelhantes, para que se previna a possibilidade de existir muitas faltas em uma mesma suíte. Além disso, o grande problema da técnica de *spectrum* é que alguns erros podem passar despercebidos, sem causar uma falha. Esse comportamento é difícil de ser evitado, pois isso pode ser uma característica do programa.

Em geral, a ferramenta emite resultados satisfatórios e que auxiliam o testador a encontrar o bloco com falta em um menor tempo. Os resultados ruins acontecem quanto muitos testes executam blocos com falta que não levam a falhas. Mas esse tipo de problema acontece também

quando um testador utiliza o diagnóstico manual e, portanto, é inerente a técnica utilizada.

7.1 Trabalhos Futuros

Como trabalhos futuros, pode-se diminuir a granularidade dos blocos para quantificar o ganho de precisão obtido. Pode-se também investigar a possibilidade de automatizar os testes manuais dos programas alvo. Assim, o testador apenas elaboraria os testes em código Java, habilitaria a ferramenta e receberia os resultados quando uma suíte de testes terminasse de executar. Outro aspecto importante que poderia ser investigado é a representatividade dos casos de teste. Também seria interessante elaborar alguma maneira de detectar faltas que passam despercebidas em determinadas execuções. Essa última proposta é um pouco mais complicada, devido à complexidade de se detectar esse tipo de falta.

Referências Bibliográficas

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. V. An evaluation of similarity coefficients for software fault localization. In: 12TH PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING. *12th Pacific Rim International Symposium on Dependable Computing*. [S.l.], 2006.

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. V. On the accuracy of spectrum-based fault localization. *IEEE Computer*, 2007.

APACHE. *Ant*. 2009. Acesso em: Mar. 2009. Disponível em: <http://ant.apache.org/>.

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. In: *Dependable and Secure Computing*. [S.l.]: IEEE, 2004. p. 11–33.

CLAYBERG, E.; RUBEL, D. *Eclipse: Building Commercial-Quality Plug-ins*. [S.l.]: Addison-Wesley Professional, 2004. (Eclipse).

COLYER, A. et al. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. [S.l.]: Addison-Wesley Professional, 2004. (Eclipse).

CONSORTIUM, W. W. W. *Extensible Markup Language (XML)*. 2009. Acesso em: Mai. 2009. Disponível em: <http://www.w3.org/XML/>.

DALLMEIER, V.; LINDIG, C.; ZELLER, A. Lightweight defect localization for java. In: 19TH EUROPEAN CONFERENCE GLASGOW (ECOOP). *19th European Conference Glasgow (ECOOP)*. [S.l.], 2005. p. 528–550.

ECLIPSE. *Eclipse Corporation*. 2009. Acesso em: Fev. 2009. Disponível em: <http://www.eclipse.org>.

ECLIPSE. *IAJC AspectJ Task*. 2009. Acesso em: Mar. 2009. Disponível em: <http://www.eclipse.org/aspectj/doc/released/devguide/antTasks-iajc.html>.

ESIPCHUK, I.; VALIDOV, D. Ptf-based test automation for java applications on mobile phones. In: IEEE 10TH INTERNATIONAL SYMPOSIUM ON CONSUMER ELECTRONICS. *Proceedings of 8th IEEE LATW 2007*. [S.l.], 2006. p. 1–3.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley Professional, 1994. (Addison-Wesley Professional Computing Series).

HAILPERN, B.; SANTHANAM, P. Software debugging, testing and verification. In: *IBM Research*. IBM Journal, 2002. p. 4–12. Disponível em: <http://www.research.ibm.com/journal/sj/411/hailpern.html>.

JACOBSON, I.; NG, P.-W. *Aspect-Oriented Software Development with Use Cases*. [S.l.]: Addison-Wesley, 2005.

JAIN, A. K.; DUBES, R. C. *Algorithms for Clustering Data*. [S.l.]: Prentice-Hall, 1988.

JONES, J. A.; HARROLD, M. J.; STASKO, J. Visualization of test information to assist fault localization. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. *International Conference on Software Engineering*. [S.l.], 2002. p. 467–477.

KAWAKAMI, L. et al. A test automation framework for mobile phones. In: 8TH IEEE LATW 2007. *Proceedings of 8th IEEE LATW 2007*. [S.l.], 2007.

KICZALES, G. et al. Aspect-oriented programming. In: *Aspect-Oriented Programming*. [S.l.]: European Conference on Object-Oriented Programming, 1997.

MEYER, A. da S. et al. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, v. 27, n. 1, p. 83–91, 2004.

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

PETROSKY, B. M. Geração automática de casos de teste automatizados no contexto de uma suíte de testes em telefones celulares. *Universidade Federal de Santa Catarina*, 2006.

RECHIA, D. et al. An object-oriented framework for improving software reuse on automated testing of mobile phones. In: IFIP 19TH TESTCOM - 7TH FATES. *IFIP 19th TESTCOM - 7th FATES*. [S.l.], 2007.

REPS, T. et al. The use of program profiling for software maintenance with applications to the year 2000 problem. In: 6TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH THE 5TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING. *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. [S.l.], 1997. p. 432–449.

SILVA, R. P. e. *UML 2 em Modelagem Orientada a Objetos*. Florianópolis: Visual Books, 2007.

XEROX. *The AspectJ Programming Guide*. 1998. Acesso em: Set. 2008. Disponível em: <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>.

ANEXO A – Código dos aspectos

A.1 Código do aspecto para o TAFStudio

```
public aspect Tracing
{
    /**
     * The trace manager.
     */
    private TraceManager traceManager = TraceManager.getInstance();

    /**
     * The scope of this aspect.
     */
    pointcut classeAlvo() : within(com.motorola..*)
        && !within(com.motorola.tafstudio.ui.action.tracing.*)
        && !within(*..CalculateSpectrumDialog)
        && !within(*..StopTracingDialog);

    /**
     * A point cut grouping some join points.
     */
    pointcut execucaoDeMetodo(Object t) : classeAlvo()
        && execution(* *(..))
        && !execution(* get*(..))
        && !execution(* set*(..))
        && target(t);

    /**
```

```
     * A point cut grouping some join points.
     */
    pointcut execucaoDeConstrutor(Object t) : classeAlvo()
        && execution(new(..))
        && target(t);

    /**
     * Before the execution flow of a method or constructor verify if trace
     * is enabled, and if is trace this entry (Signature).
     */
    before(Object t) : execucaoDeMetodo(t) || execucaoDeConstrutor(t) {
        if (traceManager.isTraceEnabled())
        {
            traceManager.traceEntry(thisJoinPoint.getSignature());
        }
    }
}
```

A.2 Código do aspecto para o TAFPlus2

```
 public aspect Tracing
{
    /**
     * The trace manager.
     */
    private TraceManager traceManager = TraceManager.getInstance();

    /**
     * The scope of this aspect.
     */
    pointcut classeAlvo() : within(com.motorola..*)
        && !within(*..events.*)
        && !within(com.motorola.tafplus2.ui.action.tracing.*)
        && !within(*..CalculateSpectrumDialog)
        && !within(*..StopTracingDialog);
```

```
    /**
     * A point cut grouping some join points.
     */
    pointcut execucaoDeMetodo(Object t) : classeAlvo()
    && execution(* *(..))
    && !execution(* get*(..))
    && !execution(* set*(..))
    && !execution(* unregisterListener*(..))
    && !execution(* registerListener*(..))
    && !execution(* registerHighwayListener*(..))
    && !execution(* unregisterHighwayListener*(..))
    && !execution(* dispatchEvent*(..))
    && !execution(* checkPermission(..))
    && target(t);

    /**
     * A point cut grouping some join points.
     */
    pointcut execucaoDeConstrutor(Object t) : classeAlvo() && execution(new(..))
    && target(t);

    /**
     * Before the execution flow of a method or constructor verify if trace is
     * enabled, and if is trace this entry (Signature).
     */
    before(Object t) : execucaoDeMetodo(t) || execucaoDeConstrutor(t) {
        if (traceManager.isTraceEnabled())
        {
            traceManager.traceEntry(thisJoinPoint.getSignature());
        }
    }
}
```

ANEXO B – Diagramas de seqüência e de classes do plug-in

B.1 Diagrama de Classes

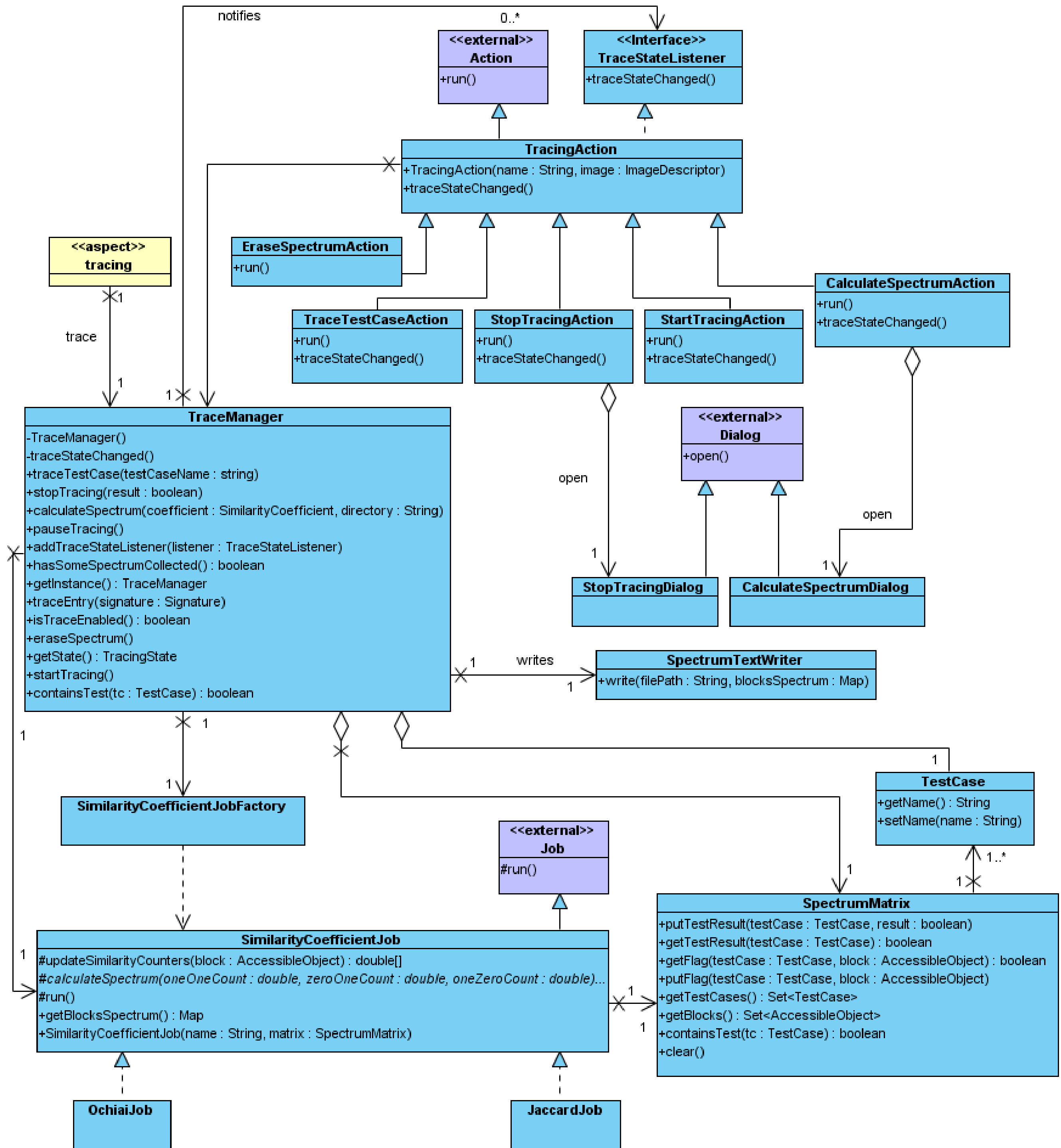


Figura B.1: Diagrama de classes

B.2 Diagrama de Seqüência: Diagnosticar Casos de Teste

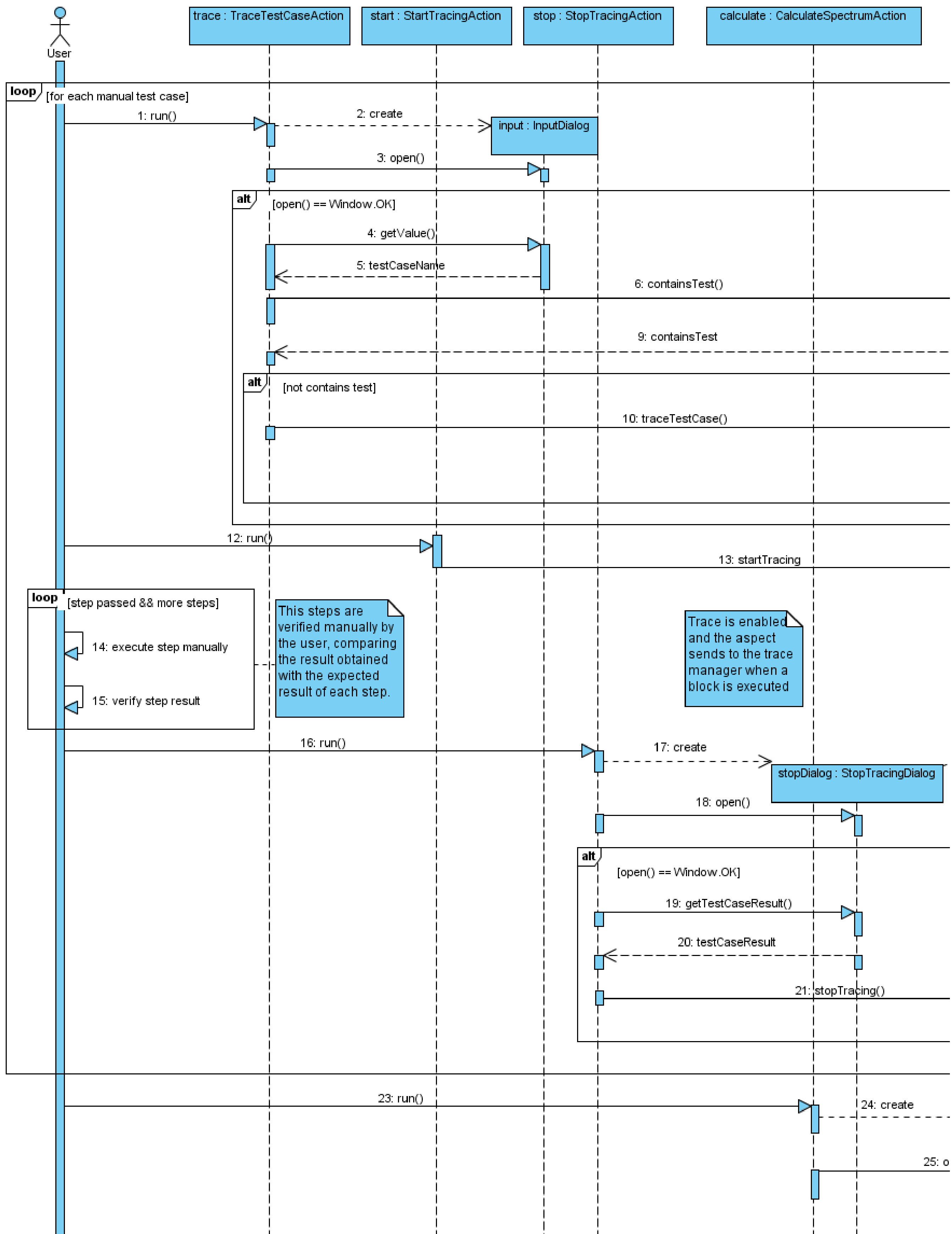


Figura B.2: Diagrama de sequência Diagnosticar Casos de Teste (1)

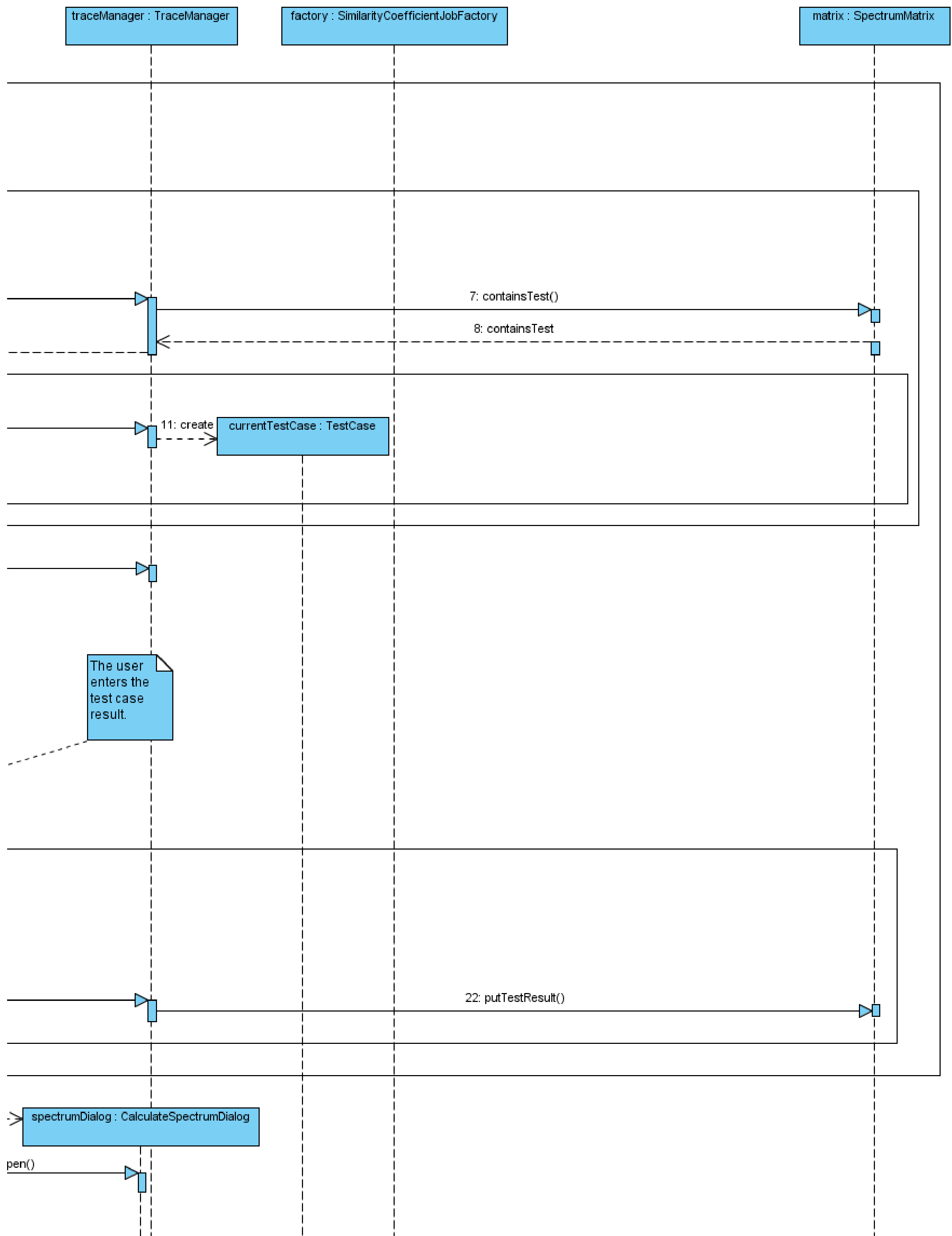


Figura B.3: Diagrama de sequência Diagnosticar Casos de Teste (2)

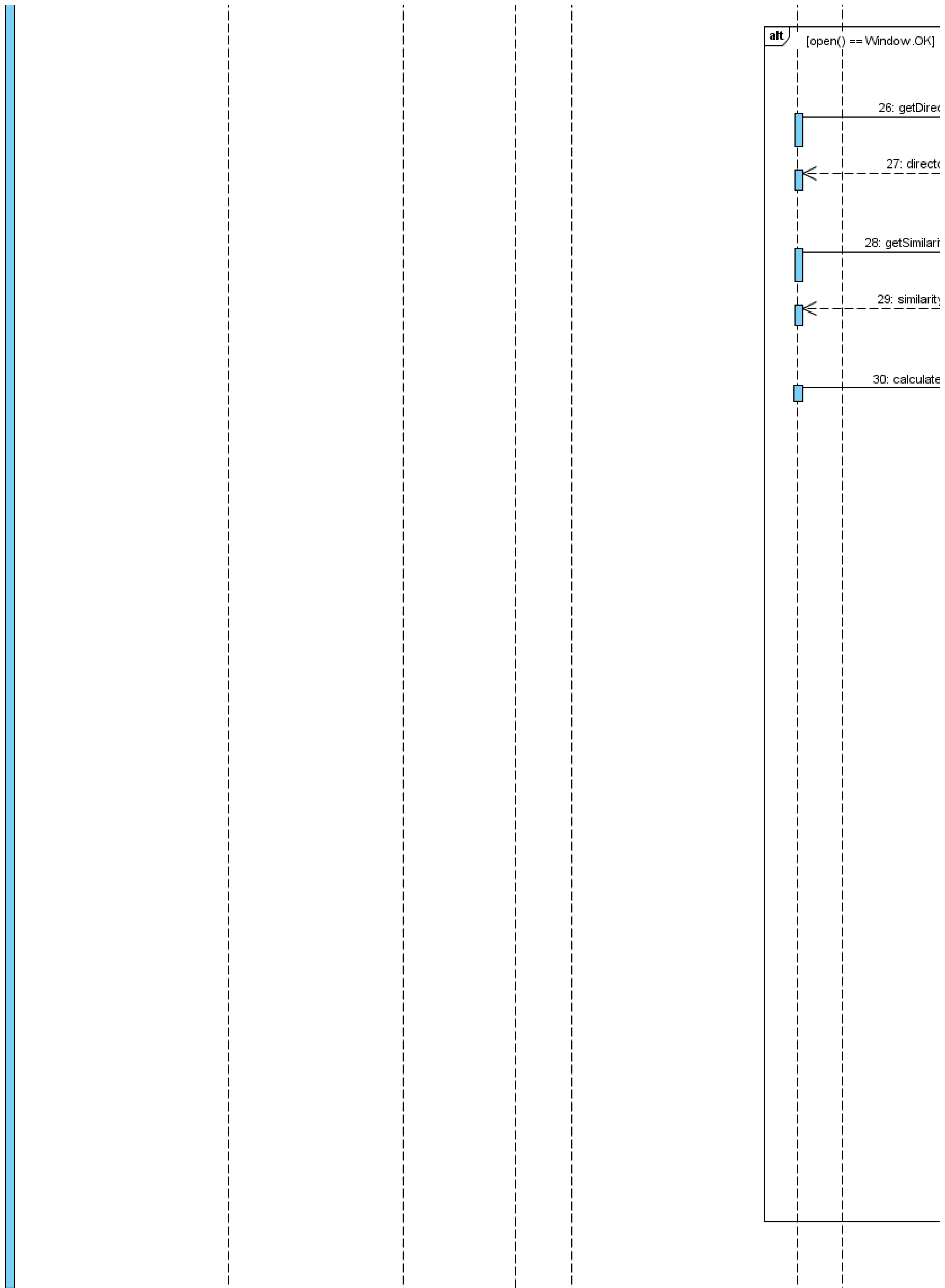


Figura B.4: Diagrama de sequência Diagnosticar Casos de Teste (3)

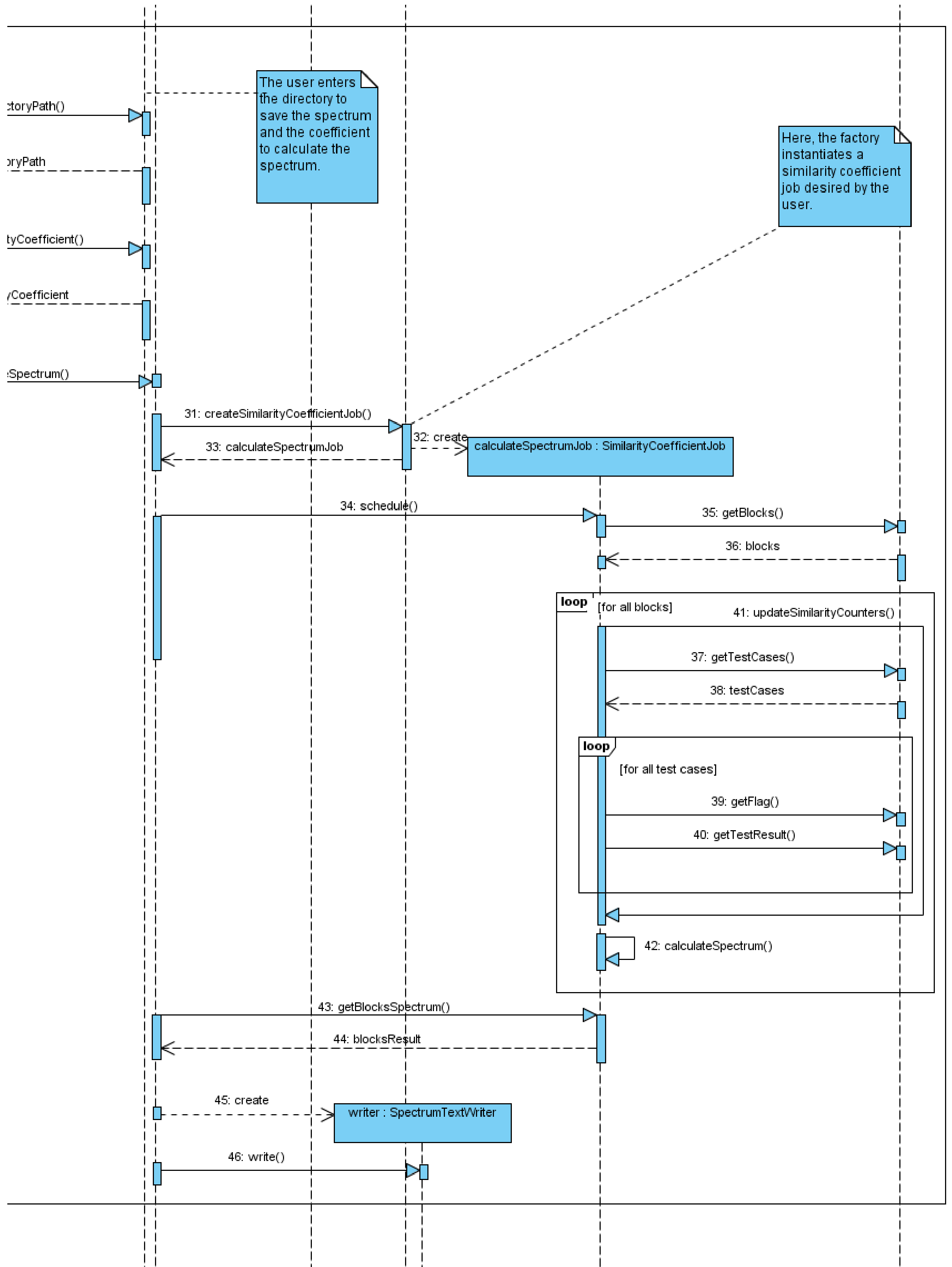


Figura B.5: Diagrama de sequência Diagnosticar Casos de Teste (4)

B.3 Diagrama de Seqüência: Controlar Rastreamento

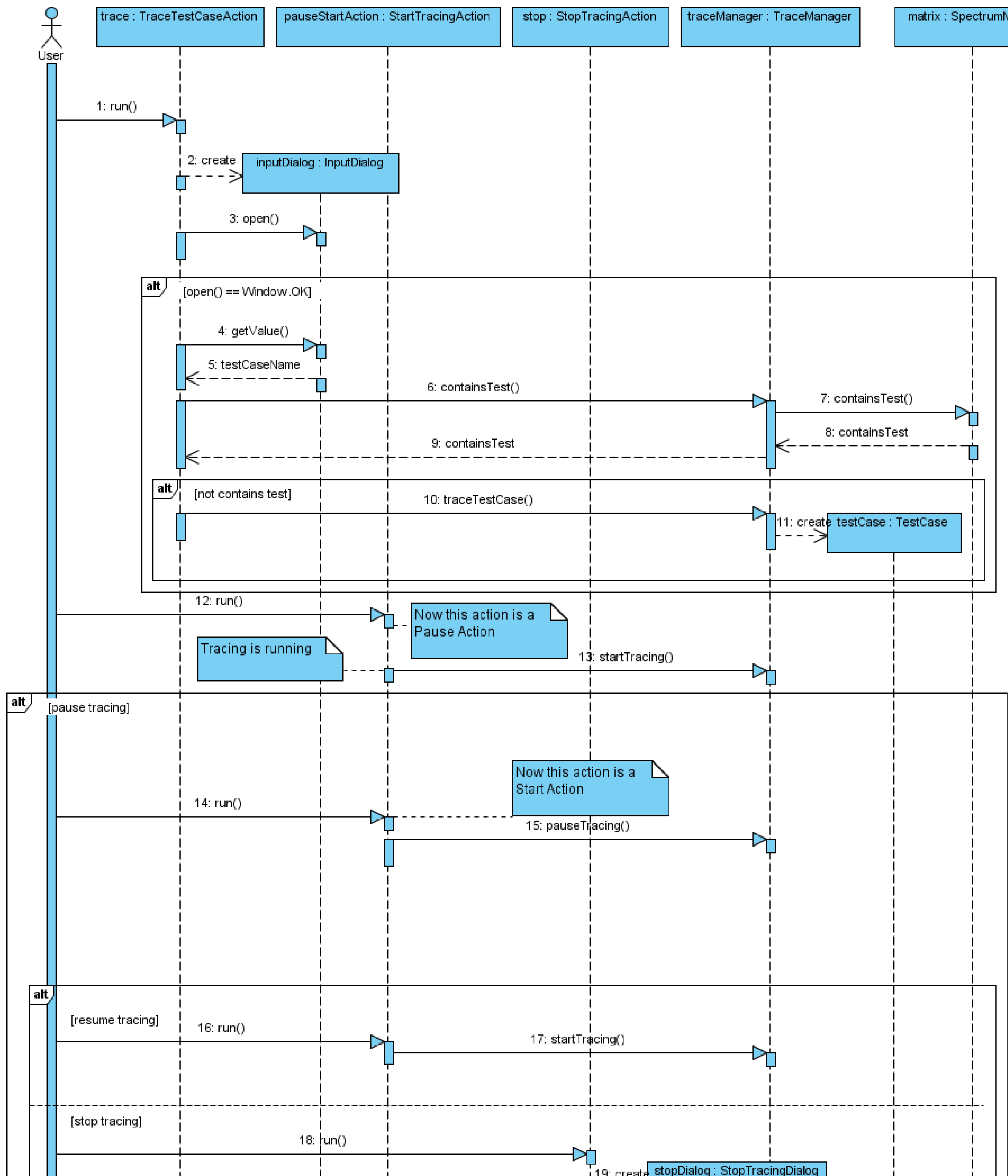


Figura B.6: Diagrama de sequência Controlar Rastreamento (1)

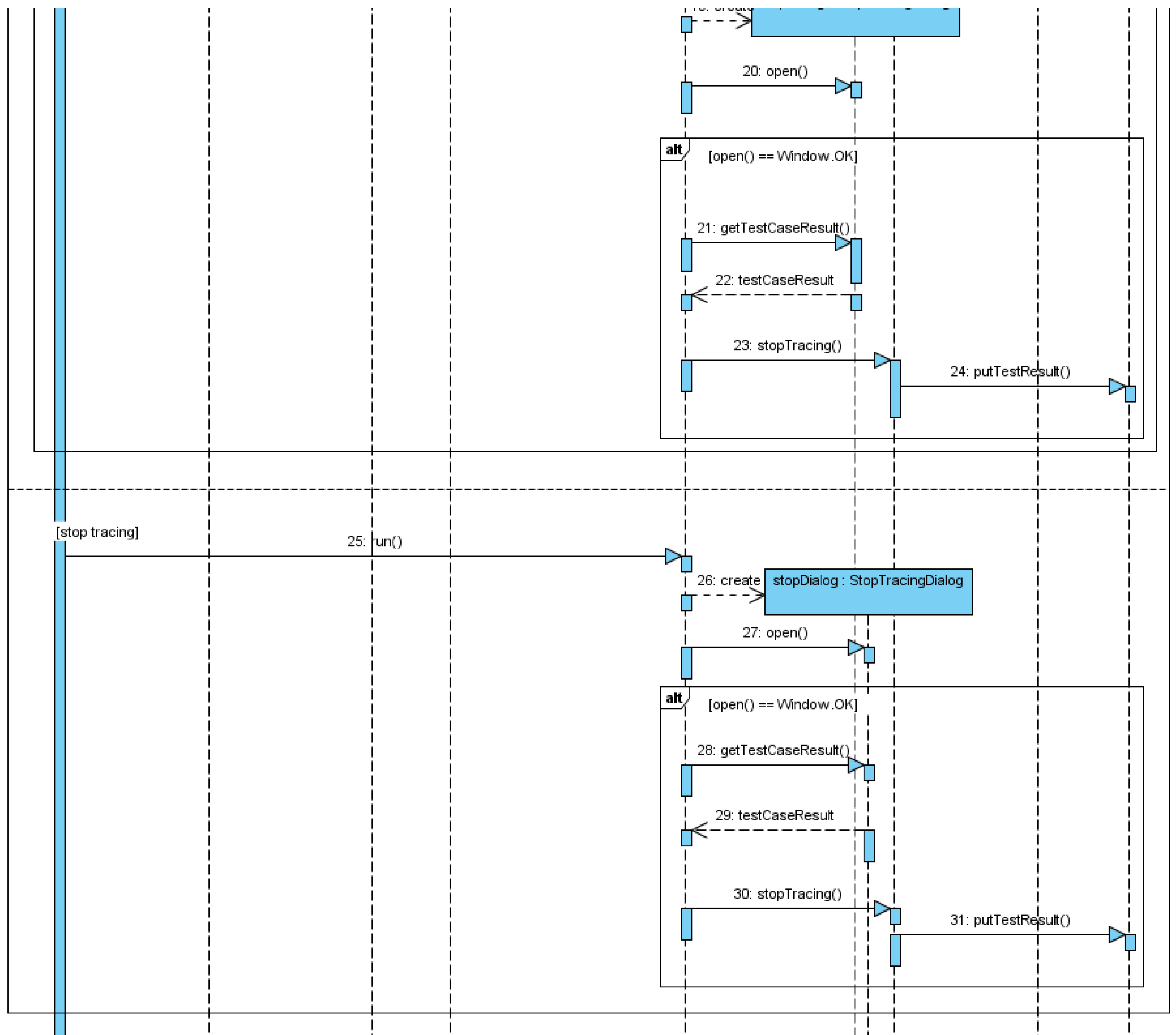


Figura B.7: Diagrama de sequência Controlar Rastreamento (2)

ANEXO C – Código do plug-in

CalculateSpectrumAction

```
package com.motorola.tafstudio.ui.action.tracing;

import org.eclipse.jface.window.Window;

import spectrum.TracingState;

import com.motorola.rcputils.ui.dialogs.DialogUtil;
import com.motorola.tafstudio.ui.dialogs.CalculateSpectrumDialog;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * This action calculates the spectrum of an executed test case suite.
 *
 * <pre>
 * CLASS:
 * Calculates the spectrum of a test suite execution.
 *
 * RESPONSIBILITIES:
 * 1) Calculates the spectrum.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 *</pre>
 */
public class CalculateSpectrumAction extends TracingAction
{

    /**
     * Action ID.
     */
```

```
    private static final String ID = "CALCULATE_SPECTRUM";

    /**
     * Constructor.
     */
    public CalculateSpectrumAction()
    {
        super("Calculate Spectrum", Images.CALCULATE_SPECTRUM.descriptor());
        setToolTipText("Calculate the spectrum collected.");
        this.setEnabled(false);
        setId(ID);
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#run()
     */
    @Override
    public void run()
    {
        TracingState status = traceManager.getState();
        if (status == TracingState.IDLE && traceManager.hasSomeSpectrumCollected())
        {
            CalculateSpectrumDialog dialog = new CalculateSpectrumDialog();

            if (dialog.open() == Window.OK)
            {
                traceManager.calculateSpectrum(dialog.getSimilarityCoefficient(),
dialog.getDirectoryPath());
            }
        }
        else
        {
            DialogUtil.showInformationMessage("Unable to calculate the spectrum",
                "at least one test case must be executed.");
        }
    }

    /*
     * (non-Javadoc)
     * @see com.motorola.tafstudio.ui.action.tracing.TraceStatusListener#
     * traceStateChanged()
     */
```

```
    @Override
    public void traceStateChanged()
    {
        TracingState status = traceManager.getState();
        setEnabled(status == TracingState.IDLE &&
traceManager.hasSomeSpectrumCollected());
    }
}
```

EraseSpectrumAction

```
package com.motorola.tafstudio.ui.action.tracing;

import com.motorola.tafstudio.ui.resources.Images;

/**
 * This action erases all the collected spectrum.
 *
 * <pre>
 * CLASS:
 * Erases the spectrum of all test suite execution.
 *
 * RESPONSIBILITIES:
 * 1) Erases the spectrum.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 *</pre>
 */
public class EraseSpectrumAction  extends TracingAction
{

    /**
     * Action ID.
     */
    private static final String ID = "TRACE_TEST";

    /**
     * Constructor.
     */
    public EraseSpectrumAction()
    {
```

```
        super("Erase Spectrum", Images.ERASE_SPECTRUM.descriptor());
        setToolTipText("Erase the spectrum collected.");
        this.setEnabled(false);
        setId(ID);
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#run()
     */
    @Override
    public void run()
    {
        traceManager.eraseSpectrum();
    }

    /*
     * (non-Javadoc)
     * @see com.motorola.tafstudio.ui.action.tracing.TraceStatusListener
     * #traceStateChanged()
     */
    @Override
    public void traceStateChanged()
    {
        setEnabled(traceManager.hasSomeSpectrumCollected());
    }

}
```

StartTracingAction

```
package com.motorola.tafstudio.ui.action.tracing;

import spectrum.TracingState;

import com.motorola.rcputils.ui.dialogs.DialogUtil;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * This action starts/pauses the tracing.
 *
 * <pre>
 * CLASS:
```

```
 * This class Start/Pause the tracing. When the tracing is running, this action
 * will behave as a Pause Action. But when the tracing is stopped or paused this
 * action will behave as a Start/Resume Action.
 *
 * RESPONSIBILITIES:
 * 1) Start/Pause the spectrum.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 *</pre>
 */
public class StartTracingAction extends TracingAction
{

    /**
     * Action ID.
     */
    private static final String ID = "START_TRACING";

    /**
     * Constructor.
     */
    public StartTracingAction()
    {
        super("Start Tracing", Images.START_TRACING.descriptor());
        this.setEnabled(false);
        setToolTipText("Start tracing of current execution.");
        setId(ID);
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#run()
     */
    @Override
    public void run()
    {
        TracingState status = traceManager.getState();
        if (status == TracingState.READY || status == TracingState.PAUSED)
        {
            traceManager.startTracing();
        }
```

```
        else if (status == TracingState.RUNNING)
        {
            traceManager.pauseTracing();
        }
        else
        {
            DialogUtil.showInformationMessage("Unable to run",
                "Only one execution can run at a time.");
        }
    }

    /*
     * (non-Javadoc)
     * @see com.motorola.tafstudio.ui.action.tracing.TraceStatusListener
     * #traceStateChanged()
     */
    @Override
    public void traceStateChanged()
    {
        TracingState status = traceManager.getState();
        if (status == TracingState.PAUSED || status == TracingState.IDLE)
        {
            setImageDescriptor(Images.START_TRACING.descriptor());
            setText("Start Tracing");
            setToolTipText("Start tracing of current execution.");
        }
        else if (status == TracingState.RUNNING)
        {
            setImageDescriptor(Images.PAUSE_TRACING.descriptor());
            setText("Pause Tracing");
            setToolTipText("Pause tracing of current execution.");
        }

        setEnabled(status == TracingState.PAUSED || status == TracingState.READY
            || status == TracingState.RUNNING);
    }
}
```

StopTracingAction

```
package com.motorola.tafstudio.ui.action.tracing;

import org.eclipse.jface.window.Window;
```

```
import spectrum.TracingState;

import com.motorola.rcputils.ui.dialogs.DialogUtil;
import com.motorola.tafstudio.ui.dialogs.StopTracingDialog;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * This action stops the tracing.
 *
 * <pre>
 * CLASS:
 * This class stops the tracing of a test case.
 *
 * RESPONSIBILITIES:
 * 1) Stops the tracing of a test case.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 *</pre>
 */
public class StopTracingAction extends TracingAction
{

    /**
     * Action ID.
     */
    private static final String ID = "STOP_TRACING";

    /**
     * Constructor.
     */
    public StopTracingAction()
    {
        super("Stop Tracing", Images.STOP_TRACING.descriptor());
        setToolTipText("Stop the tracing.");
        setId(ID);
        setEnabled(false);
    }

    /*
     * (non-Javadoc)
```

```
     * @see org.eclipse.jface.action.Action#run()
     */
    @Override
    public void run()
    {
        TracingState status = traceManager.getState();
        if (status == TracingState.RUNNING ||
traceManager.getState() == TracingState.PAUSED)
        {
            StopTracingDialog dialog = new StopTracingDialog();
            if (dialog.open() == Window.OK)
            {
                traceManager.stopTracing(dialog.getTestCaseResult());
            }
        }
        else
        {
            DialogUtil.showInformationMessage("Unable to run",
                "The trace must be enabled to stop it.");
        }
    }

    /*
     * (non-Javadoc)
     * @see com.motorola.tafstudio.ui.action.tracing.TraceStatusListener
     * #traceStateChanged()
     */
    @Override
    public void traceStateChanged()
    {
        setEnabled(traceManager.getState() == TracingState.RUNNING
            || traceManager.getState() == TracingState.PAUSED);
    }

}
```

TraceTestCaseAction

```
package com.motorola.tafstudio.ui.action.tracing;

import org.eclipse.jface.dialogs.InputDialog;
import org.eclipse.jface.window.Window;
```

```
import spectrum.TracingState;

import com.motorola.rcputils.ui.dialogs.DialogUtil;
import com.motorola.rcputils.ui.dialogs.validators.EmptyStringValidator;
import com.motorola.tafstudio.ui.resources.Images;

/**
 * This action starts the trace of a test case. The user will
 * submit a test case to be executed. When this action was
 * executed, the action Start/Pause tracing is enabled.
 *
 * <pre>
 * CLASS:
 * This class starts the trace of a test case. The user will
 * submit a test case to be executed. When this action was
 * executed, the action Start/Pause tracing is enabled.
 *
 * RESPONSIBILITIES:
 * 1) Trace a test case.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 *</pre>
 */
public class TraceTestCaseAction extends TracingAction
{

    /**
     * Action ID.
     */
    private static final String ID = "TRACE_TEST";

    /**
     * Constructor.
     */
    public TraceTestCaseAction()
    {
        super("Trace Test Case", Images.TRACE_TEST.descriptor());
        setToolTipText("Trace a manual test case.");
        setId(ID);
    }
```

```
    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.action.Action#run()
     */
    @Override
    public void run()
    {
        TracingState status = traceManager.getState();
        if (status == TracingState.IDLE)
        {
            InputDialog dialog = new InputDialog(DialogUtil.getShell(null),
"Enter test case",
                "Enter the test case name to trace:", "",
new EmptyStringValidator());
            if (dialog.open() == Window.OK)
            {
                String testName = dialog.getValue();
                if (!traceManager.containsTest(testName))
                {
                    traceManager.traceTestCase(testName);
                }
                else
                {
                   DialogUtil.showErrorMessage("This test case cannot be traced.",
                     "This test cannot be traced. Exists a test with the same name.");
                }
            }
        }
        else
        {
            DialogUtil.showInformationMessage("Unable to run",
"One test case is running/paused.");
        }
    }

    /*
     * (non-Javadoc)
     * @see com.motorola.tafstudio.ui.action.tracing.TraceStatusListener#
     * traceStateChanged()
     */
    @Override
    public void traceStateChanged()
    {
```

```
        TracingState status = traceManager.getState();
        this.setEnabled(status == TracingState.IDLE);
    }

}
```

TracingAction

```
package com.motorola.tafstudio.ui.action.tracing;

import org.eclipse.jface.action.Action;
import org.eclipse.jface.resource.ImageDescriptor;

import spectrum.TraceManager;
import spectrum.TraceStateListener;

/**
 * This class represents an abstract tracing action.
 *
 * <pre>
 * CLASS:
 * This class represents an abstract tracing action and must
 * be extended by each tracing action.
 *
 * RESPONSIBILITIES:
 * 1) Abstract implementation of a tracing action.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 *</pre>
 */
public class TracingAction extends Action implements TraceStateListener
{
    /**
     * The Trace Manager.
     */
    protected TraceManager traceManager;

    /**
     * Constructor.
     *
     * @param actionName The action name.
```

```
     * @param image The image descriptor.
     */
    public TracingAction(String actionName, ImageDescriptor image)
    {
        super(actionName, image);
        traceManager = TraceManager.getInstance();
        TraceManager.addTraceStateListener(this);
    }

    /* (non-Javadoc)
     * @see spectrum.TraceStatusListener#traceStateChanged()
     */
    @Override
    public void traceStateChanged()
    {
        // Must be implemented by the subclasses.
    }

}
```

CalculateSpectrumDialog

```
package com.motorola.tafstudio.ui.dialogs;

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.jface.layout.GridLayoutFactory;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.SelectionEvent;
import org.eclipse.swt.events.SelectionListener;
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.*;

import spectrum.SimilarityCoefficient;

import com.motorola.rcputils.ui.dialogs.DialogUtil;

/**
 * This class open a dialog to calculate the spectrum of
 * a test case execution. The user selects the similarity
 * coefficient to calculate the spectrum and the path to
 * save the diagnosis file.
 *
```

```
 * <pre>
 * CLASS:
 * This class open a dialog to calculate the spectrum of
 * a test case execution. The user selects the similarity
 * coefficient to calculate the spectrum and the path to
 * save the diagnosis file.
 *
 * RESPONSIBILITIES:
 * 1) Calculate the spectrum.
 * 2) Save the spectrum.
 *
 * COLLABORATORS:
 * 1) SimilarityCoefficient.
 * 2) TraceManager
 *</pre>
 */
public class CalculateSpectrumDialog extends Dialog implements SelectionListener
{

    /**
     * The radio that represents the Ochiai Coefficient.
     */
    private Button ochiaiRadio;

    /**
     * The radio that represents the Jaccard Coefficient.
     */
    private Button jaccardRadio;

    /**
     * The Similarity Coefficient.
     */
    private SimilarityCoefficient similarityCoefficient;

    /**
     * The directory path to save the spectrum results.
     */
    private String directoryPath;

    /**
     * The button to choose a directory path.
     */
    private Button chooseDirectoryButton;
```

```
/**
 * The field that stores the selected directory path.
 */
private Text chooseDirectoryField;

/**
 * Constructor.
 */
public CalculateSpectrumDialog()
{
    super(DialogUtil.getShell(null));
}

/*
 * (non-Javadoc)
 * @see org.eclipse.jface.window.Window#configureShell(
 * org.eclipse.swt.widgets.Shell)
 */
@Override
protected void configureShell(Shell newShell)
{
    super.configureShell(newShell);
    newShell.setText("Calculate spectrm");
}

/*
 * (non-Javadoc)
 * @see org.eclipse.jface.dialogs.Dialog#createDialogArea(
 * org.eclipse.swt.widgets.Composite)
 */
@Override
protected Control createDialogArea(Composite parent)
{
    Composite content = (Composite) super.createDialogArea(parent);
    content.setLayout(GridLayoutFactory.swtDefaults().numColumns(1).create());

    // The select result group.
    Group radioGroup = new Group(content, SWT.NONE);
    GridData rggd = new GridData();
    rggd.minimumHeight = 160;
    rggd.minimumWidth = 160;
    rggd.grabExcessHorizontalSpace = true;
```

```
    radioGroup.setText("Similarity Coefficient:");
    radioGroup.setLayoutData(rggd);
    radioGroup.setLayout(new GridLayout());

    ochiaiRadio = new Button(radioGroup, SWT.RADIO);
    ochiaiRadio.setText("Ochiai");
    ochiaiRadio.setSelection(true);
    jaccardRadio = new Button(radioGroup, SWT.RADIO);
    jaccardRadio.setText("Jaccard");

    // The select result group.
    Group directoryGroup = new Group(content, SWT.NONE);
    GridData drg = new GridData();
    drg.minimumHeight = 160;
    drg.minimumWidth = 160;
    drg.grabExcessHorizontalSpace = true;
    directoryGroup.setText("Directory to save:");
    directoryGroup.setLayoutData(drg);
    directoryGroup.setLayout(new GridLayout(2, false));

    chooseDirectoryField = new Text(directoryGroup, SWT.BORDER);
    chooseDirectoryField.setEditable(false);
    chooseDirectoryField.setLayoutData(new GridData(100, 15));

    chooseDirectoryButton = new Button(directoryGroup, SWT.NONE);
    GridData cdgd = new GridData();
    chooseDirectoryButton.setLayoutData(cdgd);
    chooseDirectoryButton.setText("...");
    chooseDirectoryButton.addSelectionListener(this);

    return content;
}

/*
 * (non-Javadoc)
 * @see org.eclipse.jface.dialogs.Dialog#okPressed()
 */
@Override
protected void okPressed()
{
    if (ochiaiRadio.getSelection())
    {
        similarityCoefficient = SimilarityCoefficient.OCHIAI;
```

```
        }
        else if (jaccardRadio.getSelection())
        {
            similarityCoefficient = SimilarityCoefficient.JACCARD;
        }

        if (directoryPath == null)
        {
            DialogUtil.showWarningMessage("Directory Empty",
                "No one directory was selected to save the spectrum."
                  + "Please, choose a directory.");
        }
        else
        {
            super.okPressed();
        }
    }

    /*
     * (non-Javadoc)
     * @seeorg.eclipse.swt.events.SelectionListener#widgetDefaultSelected(
     * org.eclipse.swt.events.SelectionEvent)
     */
    @Override
    public void widgetDefaultSelected(SelectionEvent e)
    {
        // empty
    }

    /*
     * (non-Javadoc)
     * @see
     * org.eclipse.swt.events.SelectionListener#widgetSelected(
     * org.eclipse.swt.events.SelectionEvent
     * )
     */
    @Override
    public void widgetSelected(SelectionEvent e)
    {
        if (e.getSource().equals(chooseDirectoryButton))
        {
            DirectoryDialog directoryDialog = new DirectoryDialog(
DialogUtil.getShell(null));
```

```
            String selectedDirectory = directoryDialog.open();
            if (selectedDirectory == null)
            {
                DialogUtil.showErrorMessage("The directory selected is invalid!",
                    "Invalid directory. Please, select another directory.");
            }
            else
            {
                directoryPath = selectedDirectory;
                chooseDirectoryField.setText(directoryPath);
            }
        }
    }

    /**
     * Gets the similarity coefficient to calculate the spectrum.
     *
     * @return SimilarityCoefficient The similarity coefficient.
     */
    public SimilarityCoefficient getSimilarityCoefficient()
    {
        return similarityCoefficient;
    }

    /**
     * Gets the directory path to save the spectrum results.
     *
     * @return String The directory path.
     */
    public String getDirectoryPath()
    {
        return directoryPath;
    }

}
```

StopTracingDialog

```
package com.motorola.tafstudio.ui.dialogs;

import org.eclipse.jface.dialogs.Dialog;
import org.eclipse.jface.layout.GridLayoutFactory;
import org.eclipse.swt.SWT;
```

```
import org.eclipse.swt.layout.GridData;
import org.eclipse.swt.layout.GridLayout;
import org.eclipse.swt.widgets.*;

import com.motorola.rcputils.ui.dialogs.DialogUtil;

/**
 * This class opens a dialog to stop the tracing and submit the
 * result of a test case execution.
 *
 * <pre>
 * CLASS:
 * This class opens a dialog to stop the tracing and submit the
 * result of a test case execution.
 *
 * RESPONSIBILITIES:
 * 1) Stop the tracing.
 * 2) Collect the test case result.
 *
 * COLLABORATORS:
 * 1) TraceManager.
 *
 */
public class StopTracingDialog extends Dialog
{

    /**
     * The radio that represents the PASSED result.
     */
    private Button passRadio;

    /**
     * The radio that represents the FAILED result.
     */
    private Button failRadio;

    /**
     * The test case result.
     */
    private boolean testCaseResult;

    /**
     * Constructor.
```

```
     */
    public StopTracingDialog()
    {
        super(DialogUtil.getShell(null));
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.window.Window#configureShell(
     * org.eclipse.swt.widgets.Shell)
     */
    @Override
    protected void configureShell(Shell newShell)
    {
        super.configureShell(newShell);
        newShell.setText("Stop tracing");
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.dialogs.Dialog#createDialogArea(
     * org.eclipse.swt.widgets.Composite)
     */
    @Override
    protected Control createDialogArea(Composite parent)
    {
        Composite content = (Composite) super.createDialogArea(parent);
        content.setLayout(GridLayoutFactory.swtDefaults().numColumns(2).create());

        // The select result group.
        Group radioGroup = new Group(content, SWT.NONE);
        GridData rggd = new GridData();
        rggd.minimumHeight = 160;
        rggd.minimumWidth = 160;
        rggd.grabExcessHorizontalSpace = true;
        radioGroup.setText("Test Result:");
        radioGroup.setLayoutData(rggd);
        radioGroup.setLayout(new GridLayout());

        failRadio = new Button(radioGroup, SWT.RADIO);
        failRadio.setText("Fail");
        failRadio.setSelection(true);
        passRadio = new Button(radioGroup, SWT.RADIO);
```

```
        passRadio.setText("Pass");

        return content;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.jface.dialogs.Dialog#okPressed()
     */
    @Override
    protected void okPressed()
    {
        testCaseResult = passRadio.getSelection();
        super.okPressed();
    }

    /**
     * Get the test case result.
     *
     * @return {@code true} if the test was passed, {@code false} otherwise.
     */
    public boolean getTestCaseResult()
    {
        return testCaseResult;
    }

}
```

SimilarityCoefficientJobFactory

```
package factory;

import jobs.JaccardJob;
import jobs.OchiaiJob;
import jobs.SimilarityCoefficientJob;
import spectrum.SimilarityCoefficient;
import spectrum.SpectrumMatrix;

/**
 * Represents a factory to create SimilarityCoefficientJob‘s.
 *
 * <pre>
 * CLASS:
```

```
 * This class creates SimilarityCoefficientJob‘s accordingly
 * to the coefficient supplied by the user.
 *
 * RESPONSIBILITIES:
 * 1) Creates SimilarityCoefficientJob instances.
 *
 * COLLABORATORS:
 * 1) SimilarityCoefficient.
 * 2) SimilarityCoefficientJob.
 *
 *</pre>
 */
public class SimilarityCoefficientJobFactory {

    /**
     * Create a job to calculate similarity accordingly to the coefficient
     * supplied.
     *
     * @param coefficient
     *            The supplied coefficient.
     * @param spectrumMatrix
     *            The spectrum matrix with the collected spectrum
     * @return A Job to calculate similarity.
     */
    public static SimilarityCoefficientJob createSimilarityCoefficientJob(
            SimilarityCoefficient coefficient,
            SpectrumMatrix spectrumMatrix) {

        if (coefficient == SimilarityCoefficient.OCHIAI) {
            return new OchiaiJob(SimilarityCoefficient.OCHIAI.toString(),
                    spectrumMatrix);
        } else if (coefficient == SimilarityCoefficient.JACCARD) {
            return new JaccardJob(SimilarityCoefficient.JACCARD.toString(),
                    spectrumMatrix);
        }

        throw new IllegalArgumentException("The coefficient supplied"
                + coefficient + "isn’t supported yet.");
    }

}
```

JaccardJob

```
package jobs;

import spectrum.SpectrumMatrix;

public class JaccardJob extends SimilarityCoefficientJob {

    /**
     * Constructor.
     *
     * @param name
     * @param spectrumMatrix
     */
    public JaccardJob(String name, SpectrumMatrix spectrumMatrix) {
        super(name, spectrumMatrix);
    }

    /*
     * (non-Javadoc)
     *
     * @see spectrum.TraceManager.SimilarityCoefficientJob#calculateNumerator()
     */
    public double calculateSpectrum(double oneOneCount, double zeroOneCount,
            double oneZeroCount) {
        return oneOneCount / (oneOneCount + zeroOneCount + oneZeroCount);
    }
}
```

OchiaiJob

```
package jobs;

import spectrum.SpectrumMatrix;

public class OchiaiJob extends SimilarityCoefficientJob {


    /**
     * Constructor.
     *
     * @param name
     * @param spectrumMatrix
     */
    public OchiaiJob(String name, SpectrumMatrix spectrumMatrix) {
```

```
        super(name, spectrumMatrix);
    }

    /*
     * (non-Javadoc)
     *
     * @see spectrum.TraceManager.SimilarityCoefficientJob#calculateNumerator()
     */
    public double calculateSpectrum(double oneOneCount, double zeroOneCount,
            double oneZeroCount) {
        return oneOneCount
                / (Math
                        .sqrt(((oneOneCount + zeroOneCount)
                         * (oneOneCount + oneZeroCount))));
    }

}
```

SimilarityCoefficientJob

```
package jobs;

import java.lang.reflect.AccessibleObject;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.Job;

import spectrum.SpectrumMatrix;
import spectrum.TestCase;

/**
 * Represents a job that calculates the spectrum of
 * an execution using a similarity coefficient.
 *
 * <pre>
 * CLASS:
 * This class represents a job that calculates the spectrum of
```

```
 * an execution using a similarity coefficient.
 *
 * RESPONSIBILITIES:
 * 1) Calculates the spectrum of an execution.
 *
 * COLLABORATORS:
 * 1) SpectrumMatrix.
 * 2) TestCase.
 *
 *</pre>
 */
public abstract class SimilarityCoefficientJob extends Job {

    /**
     * The blocks result.
     */
    private Map<AccessibleObject, Double> blocksResult;

    /**
     * The spectrum matrix.
     */
    private SpectrumMatrix spectrumMatrix;

    /**
     * Constructor.
     *
     * @param name
     * @param spectrumMatrix
     */
    public SimilarityCoefficientJob(String name, SpectrumMatrix spectrumMatrix) {
        super(name);
        this.spectrumMatrix = spectrumMatrix;
    }

    /*
     * (non-Javadoc)
     *
     * @seeorg.eclipse.core.runtime.jobs.Job#run(org.eclipse.core.runtime.
     * IProgressMonitor)
     */
    public IStatus run(IProgressMonitor monitor) {
        blocksResult = new HashMap<AccessibleObject, Double>();
        Set<AccessibleObject> blocks = spectrumMatrix.getBlocks();
```

```
    monitor.beginTask("Calculating the Spectrum", blocks.size());

    // Iterate over all blocks.
    Iterator<AccessibleObject> it = blocks.iterator();
    while (it.hasNext()) {
        AccessibleObject block = it.next();
        double[] similarityCounters = updateSimilarityCounters(block);
        double spectrum = calculateSpectrum(similarityCounters[0],
                similarityCounters[1], similarityCounters[2]);
        blocksResult.put(block, spectrum);
    }

    monitor.done();
    return Status.OK_STATUS;
}

/**
 * Update the similarity counters of a block.
 *
 * @param block
 *            The target block.
 * @return The similarity counters in the format: {a11, a10, a01}.
 */
protected double[] updateSimilarityCounters(AccessibleObject block) {
    double oneOneCount = 0, zeroOneCount = 0, oneZeroCount = 0;
    Set<TestCase> testCases = spectrumMatrix.getTestCases();

    Iterator<TestCase> it = testCases.iterator();
    while (it.hasNext()) {
        TestCase testCase = it.next();
        boolean flag = spectrumMatrix.getFlag(testCase, block);
        boolean passed = spectrumMatrix.getTestResult(testCase);

        // passed by this block and test failed.
        if (flag && !passed) {
            oneOneCount++;
        }
        // not passed by this block and the test failed.
        else if (!flag && !passed) {
            zeroOneCount++;
        }
        // passed by this block and test passed.
        else if (flag && passed) {
```

```
                oneZeroCount++;
            }
        }
        return new double[] { oneOneCount, zeroOneCount, oneZeroCount };
    }

    /**
     * Get the spectrum of the blocks.
     *
     * @return A Map of a block to its spectrum.
     */
    public Map<AccessibleObject, Double> getBlocksSpectrum() {
        return blocksResult;
    }

    /**
     * Calculate the spectrum using a similarity coefficient.
     *
     * @param oneOneCount
     *            Indicates the number of times that a block were executed in a
     *            test case and the test fails.
     * @param zeroOneCount
     *            Indicates the number of times that a block weren't executed in
     *            a test case and the test fails.
     * @param oneZeroCount
     *            Indicates the number of times that a block were executed in a
     *            test case and the test pass.
     * @return The spectrum of a block.
     */
    protected abstract double calculateSpectrum(double oneOneCount,
            double zeroOneCount, double oneZeroCount);

}
```

SimilarityCoefficient

```
package spectrum;

/**
 *  Represents the types of similarity coefficients to calculate
 *  the spectrum.
 *
 */
```

```java
public enum SimilarityCoefficient {

    JACCARD("Jaccard"), OCHIAI("Ochiai");

    /**
     * The coefficient name.
     */
    private String name;

    /**
     * Constructor..
     *
     * @param name The coefficient name.
     */
    SimilarityCoefficient(String name) {
        this.name = name;
    }

    /*
     * (non-Javadoc)
     * @see java.lang.Enum#toString()
     */
    public String toString() {
        return name;
    }

}
```

Spectrum

```java
package spectrum;

import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.osgi.framework.BundleContext;

/**
 * The activator class controls the plug-in life cycle
 */
public class Spectrum extends AbstractUIPlugin {

    // The plug-in ID
    public static final String PLUGIN_ID = "spectrum";
```

```
    // The shared instance
    private static Spectrum plugin;

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        plugin = this;
    }

    /*
     * (non-Javadoc)
     * @see org.eclipse.ui.plugin.AbstractUIPlugin#stop(org.osgi.framework.BundleContext)
     */
    public void stop(BundleContext context) throws Exception {
        plugin = null;
        super.stop(context);
    }

    /**
     * Returns the shared instance
     *
     * @return the shared instance
     */
    public static Spectrum getDefault() {
        return plugin;
    }

}
```

SpectrumMatrix

```
package spectrum;

import java.lang.reflect.AccessibleObject;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

/**
```

```
 * Represents the spectrum matrix that stores the executed blocks
 * and the test case results of an execution.
 *
 * <pre>
 * CLASS:
 * This class represents the spectrum matrix that stores the executed
 * blocks and the test case results of an execution.
 *
 * RESPONSIBILITIES:
 * 1) Store the trace of an execution in a matrix.
 *
 * COLLABORATORS:
 * NONE
 *
 *</pre>
 */
public class SpectrumMatrix {

    /**
     * Maps a test case to blocks executed by this test.
     */
    private Map<TestCase, Map<AccessibleObject, Boolean>> mainMatrix = new HashMap<TestCase,
                            Map<AccessibleObject, Boolean>>();

    /**
     * Maps a test case to the result of this test.
     */
    private Map<TestCase, Boolean> errorMatrix = new HashMap<TestCase, Boolean>();

    /**
     * Put a flag indicating that in this test case, the block was executed.
     *
     * @param currentTestCase
     *            The current test case being executed.
     * @param block
     *            The block that was executed.
     */
    public void putFlag(TestCase currentTestCase, AccessibleObject block,
            Boolean t) {
        Map<AccessibleObject, Boolean> row = mainMatrix.get(currentTestCase);
        if (row == null) {
            row = new HashMap<AccessibleObject, Boolean>();
            mainMatrix.put(currentTestCase, row);
```

```
    }
    row.put(block, t);
}

/**
 * Get the flag of a block in a specific test case.
 *
 * @param testCase
 *            The specific test case.
 * @param block
 *            The block to retrieve the flag.
 *
 * @return {@code true} if in this test case the block was executed, {@code
 *         false} otherwise.
 */
public Boolean getFlag(TestCase testCase, AccessibleObject block) {

    Map<AccessibleObject, Boolean> row = mainMatrix.get(testCase);
    Boolean flag = row.get(block);
    // If this block doesn't has a flag...
    if (flag == null) {
        // ... is because it wasn't executed in this test case, so return false.
        return false;
    }
    return flag;
}

/**
 * Put the test result of a test case.
 *
 * @param testCase
 *            The target test case.
 * @param result
 *            The result.
 */
public void putTestResult(TestCase testCase, boolean result) {
    errorMatrix.put(testCase, result);
}

/**
 * Get the result of a test case.
 *
 * @param testCase
```

```
 *            The test case to retrieve the result.
 * @return {@code true} if the test was passed, {@code false} otherwise.
 */
public boolean getTestResult(TestCase testCase) {
    return errorMatrix.get(testCase);
}

/**
 * Get all the test cases of this matrix.
 *
 * @return A Set with the test cases of the spectrum matrix.
 */
public Set<TestCase> getTestCases() {
    return mainMatrix.keySet();
}

/**
 * Get all the blocks executed and traced in this matrix.
 *
 * @return A Set with the blocks of the spectrum matrix.
 */
public Set<AccessibleObject> getBlocks() {
    Map<AccessibleObject, Boolean> executedBlocks = new HashMap<AccessibleObject, Boolean>();
    Iterator<TestCase> it = mainMatrix.keySet().iterator();

    while (it.hasNext()) {
        // Put all the blocks of each test case in a big map.
        executedBlocks.putAll(mainMatrix.get(it.next()));
    }

    // Return the set of executed blocks.
    return executedBlocks.keySet();
}

/**
 * Verify if the error matrix contains this test case.
 *
 * @param testName
 *            The test case name.
 * @return {@code true} if the test is inside the matrix, {@code false}
 *         otherwise.
 */
public boolean containsTest(TestCase tc) {
```

```
        Iterator<TestCase> it = errorMatrix.keySet().iterator();
        while (it.hasNext()) {
            TestCase itTc = it.next();
            if (itTc.equals(tc)) {
                return true;
            }
        }
        return false;
    }

    /**
     * Clear this matrix.
     */
    public void clear() {
        mainMatrix.clear();
        errorMatrix.clear();
    }

}
```

TraceManager

```
package spectrum;

import java.lang.reflect.Constructor;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;

import jobs.SimilarityCoefficientJob;

import org.aspectj.lang.Signature;
import org.aspectj.lang.reflect.ConstructorSignature;
import org.aspectj.lang.reflect.MethodSignature;
import org.eclipse.core.runtime.jobs.IJobChangeEvent;
import org.eclipse.core.runtime.jobs.IJobChangeListener;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.core.runtime.jobs.JobChangeAdapter;

import writer.SpectrumTextWriter;
import factory.SimilarityCoefficientJobFactory;

/**
```

```
 * Manages the trace operations of TAFStudio and TAFPlus2 plug-ins.
 *
 * <pre>
 * CLASS:
 * This class controls the trace operations of TAFStudio and TAFPlus2 plug-ins.
 * This class trace when a block is executed in an execution flow. After the trace
 * operation, the class calculate the spectrum of all blocks using a similarity
 * coefficient.
 *
 * RESPONSIBILITIES:
 * 1) Manages the the tracing of executions.
 * 2) Calculate spectrum of executions.
 *
 * COLLABORATORS:
 * NONE
 *
 * USAGE:
 * 1) Trace a entry when an advice is triggered in an aspect file.
 * Signature signature = thisJoinPoint.getSignature();
 * TraceManager.getInstance().traceEntry(signature);
 *</pre>
 */
public class TraceManager {

    /**
     * The spectrum file name.
     */
    protected static final String SPECTRUM_FILE_NAME = "spectrum";

    /**
     * The singleton Trace Manager.
     */
    private static TraceManager traceInstance;

    /**
     * The status of the tracing.
     */
    private TracingState state = TracingState.IDLE;

    /**
     * Represents the current test case (i.e TC_001).
     */
    private TestCase currentTestCase;
```

```
/**
 * This matrix stores the blocks spectrum.
 */
private SpectrumMatrix spectrumMatrix;

/**
 * The Job to calculate the spectrum of an execution.
 */
private SimilarityCoefficientJob calculateSpectrumJob;

/**
 * The directory to save the spectrum results.
 */
private String directoryPath;

/**
 * List of listeners of test running states.
 */
private static List<TraceStateListener> stateListeners = new ArrayList<TraceStateListener>();

/**
 * Job listener to save the spectrum when the calculate spectrum job has
 * finished.
 */
private IJobChangeListener spectrumJobListener = new JobChangeAdapter() {
    /*
     * (non-Javadoc)
     *
     * @see
     * org.eclipse.core.runtime.jobs.JobChangeAdapter#done(org.eclipse.core
     * .runtime.jobs. IJobChangeEvent)
     */
    @Override
    public void done(IJobChangeEvent event) {
        if (event.getJob() instanceof SimilarityCoefficientJob) {
            // When the job is done, save the results in a file.
            SpectrumTextWriter stw = new SpectrumTextWriter();
            stw.write(directoryPath + "\\" + SPECTRUM_FILE_NAME + ".txt",
                    ((SimilarityCoefficientJob) event.getJob())
                            .getBlocksSpectrum());
        }
    }
```

```
};

/**
 * Constructor.
 */
private TraceManager() {
    spectrumMatrix = new SpectrumMatrix();
}

/**
 * Get the singleton instance of this class.
 *
 * @return TraceManager singleton
 */
public static TraceManager getInstance() {
    if (traceInstance == null) {
        traceInstance = new TraceManager();
    }
    return traceInstance;
}

/**
 * Trace an entry (for example a method) of the current test case.
 *
 * @param signature
 *            Signature that represents a block.
 */
public void traceEntry(Signature signature) {
    if (signature instanceof MethodSignature) {
        MethodSignature methodSignature = (MethodSignature) signature;
        Method method = methodSignature.getMethod();
        spectrumMatrix.putFlag(currentTestCase, method, true);
    } else if (signature instanceof ConstructorSignature) {
        ConstructorSignature constructorSignature = (ConstructorSignature) signature;
        Constructor<?> constructor = constructorSignature.getConstructor();
        spectrumMatrix.putFlag(currentTestCase, constructor, true);
    }
}

/**
 * Verify if the trace is enabled.
 *
 * @return <true> if trace is enabled, <false> otherwise.
```

```
     */
    public boolean isTraceEnabled() {
        return state == TracingState.RUNNING;
    }

    /**
     * Clear spectrum.
     */
    public void eraseSpectrum() {
        spectrumMatrix.clear();
        traceStateChanged();
    }

    /**
     * Calculate the spectrum of a test suite and notify the listeners.
     *
     * @param similarityCoefficient
     *            The coefficient to calculate the spectrum.
     */
    public void calculateSpectrum(SimilarityCoefficient similarityCoefficient,
            String directoryPath) {
        // Save the directory path selected by the user.
        this.directoryPath = directoryPath;

        // Creates the job to calculate the spectrum.
        calculateSpectrumJob = SimilarityCoefficientJobFactory
                .createSimilarityCoefficientJob(similarityCoefficient,
                        spectrumMatrix);
        calculateSpectrumJob.addJobChangeListener(spectrumJobListener);
        calculateSpectrumJob.setPriority(Job.LONG);
        calculateSpectrumJob.schedule();

        // Notify listeners.
        traceStateChanged();
    }

    /**
     * Return the state of the tracing.
     *
     * @return The status.
     */
    public TracingState getState() {
        return state;
```

```
}

/**
 * Trace a test case.
 *
 * @param testCase
 *            The test case that will execute.
 */
public void traceTestCase(String name) {
    currentTestCase = new TestCase(name);
    state = TracingState.READY;
    traceStateChanged();
}

/**
 * Start or resume tracing.
 */
public void startTracing() {
    state = TracingState.RUNNING;
    traceStateChanged();
}

/**
 * Pause tracing.
 */
public void pauseTracing() {
    state = TracingState.PAUSED;
    traceStateChanged();
}

/**
 * Adds a listener to the trace state events of this class.
 *
 * @param listener
 *            The listener to this class.
 */
public static void addTraceStateListener(TraceStateListener listener) {
    stateListeners.add(listener);
}

/**
 * Alerts all listeners of this class that the state of the
 * {@link TraceManager} has changed.
```

```
     */
    private static void traceStateChanged() {
        for (TraceStateListener listener : stateListeners) {
            listener.traceStateChanged();
        }
    }

    /**
     * Stop tracing and submit a result to the test case.
     *
     * @param result
     *            The test case result.
     */
    public void stopTracing(boolean result) {
        state = TracingState.IDLE;
        spectrumMatrix.putTestResult(currentTestCase, result);
        traceStateChanged();
    }

    /**
     * Inform if the tracing has collected some spectrum.
     *
     * @return {@code true} if spectrum isn't empty, {@code false} otherwise.
     */
    public boolean hasSomeSpectrumCollected() {
        return spectrumMatrix.getTestCases().size() >= 1
                && state == TracingState.IDLE;
    }

    /**
     * Verify if the desired test case was traced.
     *
     * @param testName
     *            The test case name.
     * @return {@code true} if the test was traced, {@code false} otherwise.
     */
    public boolean containsTest(String testName) {
        return spectrumMatrix.containsTest(new TestCase(testName));
    }

}
```

TraceStateListener

```
package spectrum;

/**
 * This class is a trace listener.
 *<pre>
 * CLASS:
 * This class is a trace listener. All the classes that implements
 * this interface will receive events generated by the class
 * TraceManager.
 *
 * COLLABORATORS:
 * NONE
 *
 * RESPONSIBILITIES:
 * 1) Listen the events of the class TraceManager.
 */
public interface TraceStateListener
{

    /**
     * Notifies all listeners that the {@link TraceManager} state has changed.
     */
    public void traceStateChanged();

}
```

TestCase

```
package spectrum;

/**
 * This class represents a manual test case.
 *<pre>
 *
 * CLASS:
 * This class represents a manual test case.
 *
 * RESPONSIBILITIES:
 * 1) Be a manual test case.
 *
 */
public class TestCase {
```

```
    /**
     * The test case name.
     */
    private String name;

    /**
     * Constructor.
     * @param name The test case name.
     */
    public TestCase(String name) {
        this.name = name;
    }

    /**
     * Gets the test case name.
     * @return name
     */
    public String getName() {
        return name;
    }

    /*
     * (non-Javadoc)
     * @see java.lang.Object#equals(java.lang.Object)
     */
    public boolean equals(Object o) {
        return this.toString().equals(o.toString());
    }

    /*
     * (non-Javadoc)
     * @see java.lang.Object#toString()
     */
    public String toString() {
        return name;
    }

}
```

TracingState

```
package spectrum;
```

```
/**
 * The states of a tracing execution.
 *
 */
public enum TracingState {

    IDLE, READY, RUNNING, PAUSED;

}
```

SpectrumTextWriter

```
package writer;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;
import java.lang.reflect.AccessibleObject;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Map;

/**
 * This class writes the spectrum of all blocks in a file.
 *
 *<pre>
 * CLASS:
 * This class writes the spectrum of all blocks in a file.
 *
 * COLLABORATORS:
 * NONE
 *
 * RESPONSIBILITIES:
 * 1) Writes the spectrum.
 */
public class SpectrumTextWriter {

    /**
     * Format the blocks spectrum map to a list and order the result.
     *
```

```
     * @param blocksSpectrumMap
     */
    private List<String> formatBlocksSpectrum(
            Map<AccessibleObject, Double> blocksSpectrumMap) {
        List<String> blocksSpectrumList = new ArrayList<String>();
        for (int i = 0; i < blocksSpectrumMap.size(); i++) {
            AccessibleObject block = (AccessibleObject) blocksSpectrumMap
                    .keySet().toArray()[i];
            Double spectrum = blocksSpectrumMap.get(block);
            if (!spectrum.isNaN() && spectrum != 0) {
                blocksSpectrumList.add(spectrum + " | " + block);
            }
        }
        Collections.sort(blocksSpectrumList);
        Collections.reverse(blocksSpectrumList);

        return blocksSpectrumList;
    }

    /**
     * Write the spectrum results to a file.
     *
     * @param filePath
     *            The file path to write the results.
     */
    public void write(String filePath,
            Map<AccessibleObject, Double> blocksSpectrum) {
        List<String> blocksSpectrumMap = formatBlocksSpectrum(blocksSpectrum);
        Writer writer = null;

        File file = new File(filePath);
        try {
            file.createNewFile();
            writer = new BufferedWriter(new FileWriter(file));

            writer.write("Blocks spectrum: \n \n");
            writer.write("Blocks size: " + blocksSpectrumMap.size() + "\n \n");
            for (String block : blocksSpectrumMap) {
                writer.write(block + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
```

```
            try {
                if (writer != null) {
                    writer.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```

ANEXO D – Suítes de testes

D.1 Suítes de casos de teste manuais do TAFPlus2

As suítes de casos de teste manuais para o TAFPlus2 estão sendo mostradas nas tabelas D.1 até D.34.

SUITE NAME: **MANAGING EXECUTIONS CONFIGURATIONS (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_001	Create an integration execution	1	Having a properly configured execution version profile, the user creates a new integration execution configuration.	On the tab 'Test List', no test group are available. On the tab 'Phones config', there is no added phone slot. On the tab 'TAF Configuration', the following tags are configured with the value '1': TAF Developer Log Level, TAF Debug Log Level and PTF Log Level. The action 'Starts Execution' is disabled. On the "Execution configuration"bar, the user can read the red warning that says there are 2 errors, and if the mouse is passed over the message a tooltip is shown specifying the errors (in this case, that the execution has no enabled test group and no enabled phone slot).
TP2_TEST_002	Create an stress execution	1	Having a properly configured execution version profile, the user creates a new stress execution configuration.	On the tab 'Test List', no test group are available. On the tab 'Phones config', there is no added phone slot. On the tab 'TAF Configuration', the following tags are configured with the value '1': TAF Developer Log Level, TAF Debug Log Level and PTF Log Level. The action 'Starts Execution' is disabled. On the "Execution configuration"bar, the user can read the red warning that says there are 2 errors, and if the mouse is passed over the message a tooltip is shown specifying the errors (in this case, that the execution has no enabled test group and no enabled phone slot).

Tabela D.1: Suíte 001 do TAFPlus2: Testes 001 e 002

SUITE NAME: **MANAGING EXECUTIONS CONFIGURATIONS (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_003	Renaming one stress execution.	1	Having a properly configured execution version profile and an integration/stress execution configuration, the user selects to rename this execution.	A dialog appears asking for the new execution name. The old name appears on the field.
		2	Enter "E"and submit.	The execution was renamed and the new name appears on the navigation view.
TP2_TEST_004	Rename one execution to a name that already exists.	1	Having a properly configured execution version profile and two integration/stress execution configurations, the user selects to rename the first execution.	A dialog appears asking for the new execution name. The old name appears on the field.
		2	Enter "E"and submit.	The execution was renamed and the new name appears on the navigation view.
		3	Select to rename the other Execution.	A dialog appears asking for the new execution name. The old name appears on the field.
		4	Enter "E"and submit.	A message informs that this operation can't be done because exists one execution with this name.
		5	Close the dialog and cancel the operation.	The execution wasn't renamed.

Tabela D.2: Suíte 001 do TAFPlus2: Testes 003 e 004

SUITE NAME: **MANAGING EXECUTIONS CONFIGURATIONS (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_005	Edit a TAF configuration	1	Having a properly configured execution version profile, the user creates a new execution configuration.	The following tags are configured with the value '1': TAF Developer Log Level, TAF Debug Log Level and PTF Log Level.
		2	The user tries to change each log level with the values: -1, 0 , 1, 5, 6, 100	On the expand bar 'Other Tags', the three tags are shown with their respective defined value on the expand bar 'TAF Configuration' above. When the values -1, 6 and 1000 are inserted a warning shows that is not possible to do so.
		3	The user adds another different tag.	The new added tag is shown with its name and value defined previously on the dialog 'Edit a tag'.
		4	The user removes this previously added tag.	The old tag is not shown anymore.
		5	The user selects one of the default tags (TAF DEBUG LEVEL, PTF LOG LEVEL or TAF DEVELOPER LEVEL) and press the button 'Remove'.	Nothing occurs, because the default tags cannot be removed, and the selected tag continues being shown.
		6	The user adds new tags from a specific file (by pressing the button 'Import File').	Each tag and its value written on the file are shown in the table on the expand bar 'Other Tags', together with the default tags.

Tabela D.3: Suíte 001 do TAFPlus2: Teste 005

SUITE NAME: **MANAGING EXECUTIONS CONFIGURATIONS (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_006	Edit automatic actions.	1	Having a properly configured execution version profile, the user creates a new execution configuration.	On the tab 'Test List', no test group are available. On the tab 'Phones config', there is no added phone slot. On the tab 'TAF Configuration', the following tags are configured with the value '1': TAF Developer Log Level, TAF Debug Log Level and PTF Log Level. The action 'Starts Execution' is disabled. On the "Execution configuration"bar, the user can read the red warning that says there are 2 errors, and if the mouse is passed over the message a tooltip is shown specifying the errors (in this case, that the execution has no enabled test group and no enabled phone slot).
		2	The user goes to the tab Automatic Actions.	The destination of the files Export XLS File and Export Unified Logs are empty.
		3	Choose a file to be the Export XLS File destination	The file appears on the destination field of the Export XLS File field.
		4	Choose a directory to be the Export Unifeied Logs destination	The directory appears on the destination field of the Export Unified Logs.

Tabela D.4: Suíte 001 do TAFPlus2: Teste 006

SUITE NAME: **MANAGING EXECUTIONS CONFIGURATIONS (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_007	Rename an execution opened.	1	Having a properly configured execution version profile and an integration/stress execution configuration opened, the user selects to rename this execution.	A dialog appears asking for the new execution name. The old name appears on the field.
		2	Enter ”E”and submit.	The execution was renamed and the new name appears on the navigation view and on the execution editor.
TP2_TEST_008	Remove an execution.	1	Having a properly configured execution version profile and an integration/stress execution configuration the user selects to remove this execution.	A dialog appears alerting the user that if the execution is deleted, all files in the execution folder will be remvoed too.
		2	Accept the dialog.	The execution was removed.
TP2_TEST_009	Remove three executions.	1	Having a properly configured execution version profile and three integration/stress execution configuration the user selects to remove all executions.	A dialog appears alerting the user that if the executions are deleted, all files in the executions folders will be remvoed too.
		2	Accept the dialog.	All executions were removed.

Tabela D.5: Suíte 001 do TAFPlus2: Testes 007, 008 e 009

SUITE NAME: **MANAGING GROUPS (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_010	Create three groups and some test cases on an integration execution configuration.	1	Having a properly configured execution version profile and an opened integration execution configuration, the user creates a new group.	A dialog appears and the fields priority, group name and execution version profile are mandatory.
		2	Fill the mandatory fields and creates the group.	The group was created and appears on the test list tab. Verify that the fields Attempts, TC Suite and TC Enviromnent Name appears on the group fields.
		3	Select to Add Manually three test cases to this group.	The tests appears on the test cases list and are enabled. The error message says that exists only one error now: The execution has no enabled phone slot.
		4	Create more two groups.	The groups were created and appears on the test list tab. Verify that the fields Attempts, TC Suite and TC Enviromnent Name appears on the group fields. The error message stay the same: The execution has no enabled phone slot.

Tabela D.6: Suíte 002 do TAFPlus2: Teste 010

SUITE NAME: **MANAGING GROUPS (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_011	Create a group with one test case and disable it.	1	Having a properly configured execution version profile and an opened integration execution configuration, the user creates a new group.	A dialog appears and the fields priority, group name and execution version profile are mandatory.
		2	Fill the mandatory fields and creates the group.	The group was created and appears on the test list tab. Verify that the fields Attempts, TC Suite and TC Enviromnent Name appears on the group fields.
		3	Select to Add Manually one test case to this group.	The test appears on the test cases list and are enabled. The error message says that exists only one error now: The execution has no enabled phone slot.
		4	Disable the group.	The group was disabled and the error message says that exists 2 errors again: No enabled phone slot and no enabled test group.

Tabela D.7: Suíte 002 do TAFPlus2: Teste 011

SUITE NAME: **MANAGING GROUPS (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_012	Creating groups with the same name.	1	Having a properly configured execution version profile and an opened integration execution configuration, the user creates a new group.	A dialog appears and the fields priority, group name and execution version profile are mandatory.
		2	Fill the mandatory fields and creates the group.	The group was created and appears on the test list tab. Verify that the fields Attempts, TC Suite and TC Enviromnent Name appears on the group fields.
		3	Create another group.	A dialog appears and the fields priority, group name and execution version profile are mandatory.
		4	Fill the mandatory fields and creates the group. But, in the name field enter the same name of the previously created group.	An exception was throw and says that exists a group with this name.

Tabela D.8: Suíte 002 do TAFPlus2: Teste 012

SUITE NAME: **MANAGING GROUPS (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_013	Create three groups and some test cases on an stress execution configuration.	1	Having a properly configured execution version profile and a opened stress execution configuration, the user creates a new group.	A dialog appears and the fields priority, group name and execution version profile are mandatory.
		2	Fill the mandatory fields and creates the group.	The group was created and appears on the test list tab. Verify that the fields Loop Iterations and Random Model appears on the group fields.
		3	Select to Add Manually three test cases to this group.	The tests appears on the test cases list and are enabled. The error message says that exists only one error now: The execution has no enabled phone slot.
		4	Create more two groups on this stress configuration.	The groups were created and appears on the test list tab. Verify that the fields Loop Iterations and Random Model appears on the group fields.
TP2_TEST_014	Create a group without name.	1	Having a properly configured execution version profile and a opened stress execution configuration, the user creates a new group.	A dialog appears and the fields priority, group name and execution version profile are mandatory.
		2	Fill the mandatory fields and creates the group. But, in the name field only enter a space.	A message appears on the dialog saying that you must enter a name to the field name.
		3	Close the dialog.	The dialog was closed and the group wasn't created.

Tabela D.9: Suíte 002 do TAFPlus2: Testes 013 e 014

SUITE NAME: **MANAGING GROUPS (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_015	Rename a group.	1	Having a properly configured execution version profile and an stress execution configuration with one group without test cases, open this group.	The group was opened with no one test case.
		2	Enter a new name to this group.	The group name was updated on the name field and on the tab window title.
TP2_TEST_016	Add test cases to a group and create a blank phone slot.	1	Having a properly configured execution version profile and an opened integration execution configuration with one group without test cases. Open this group.	The group was opened with no one test case.
		2	Add manually some tests cases.	The tests appears on the test cases list and are enabled. The error message says that exists only one error now: The execution has no enabled phone slot.
		3	Go to the tab Phones config.	No one phone slot exists.
		4	Add a blank phone slot.	The phone slot was created and isn't enabled.
		5	Enable this phone slot.	A warning appears saying that is impossible to enable this slot due to problems with the configuration.

Tabela D.10: Suíte 002 do TAFPlus2: Testes 015 e 016

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_030	Add test cases to a group and create a blank phone slot.	1	Having a properly configured execution version profile and an integration execution configuration with one group without test cases. Open this group.	The group was opened with no one test case.
		2	Add manually some tests cases.	The tests appears on the test cases list and are enabled. The error message says that exists only one error now: The execution has no enabled phone slot.
		3	Go to the tab Phones config.	No one phone slot exists.
		4	Add a blank phone slot.	The phone slot was created and isn't enabled.
		5	Enable this phone slot.	A warning message says that it's impossible to enable this phone slot due the configuration problems.
TP2_TEST_031	Add two blank phones slots and enable one of them.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. Go to the tab Phones Config.	No phone slot exists.
		2	Add one blank phone slot.	The phone slot was created and isn't enabled.
		3	Add another phone slot.	The phone slot was created and isn't enabled.
		4	Enable the second phone slot.	A warning message says that it's impossible to enable this phone slot due the configuration problems.

Tabela D.11: Suíte 003 do TAFPlus2: Testes 030 e 031

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_032	Renaming phone slots.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. Go to the tab Phones Config that has two enabled phone slots.	Two phone slots are listed and appears on the tab. The slots are enabled.
		2	Select the first phone slot.	The phone slot details window is opened.
		3	Rename this phone slot to ”PS1”	The phone slot was renamed and the name was updated on the phone slot list.
		4	Select the second phone slot.	The phone slot details window is opened.
		5	Rename this phone slot to ”PS2”	The phone slot was renamed and the name was updated on the phone slot list.
TP2_TEST_033	Removing phone slots.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. Go to the tab Phones Config that has two enabled phone slots.	Two phone slots are listed and appears on the tab. The slots are enabled.
		2	Select to remove the first phone slot.	The phone slot was removed and doesn’t appears on the phone slot list.
		3	Select to remove the second phone slot.	The phone slot was removed and doesn’t appears on the phone slot list. Now, the error message indicating that no one phone slot is enabled.

Tabela D.12: Suíte 003 do TAFPlus2: Testes 032 e 033

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_034	Add a phone slot from XML.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. No one phone slot exists. Go to the tab Phones Config.	No phone slot exists.
		2	Add a phone slot from XML.	An unexpected error appears saying that prior you need to configure the location of the XML templates.
		3	Close the error message.	The phone slot was not created.
TP2_TEST_035	Add a configured phone slot from tagset.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. No one phone slot exists. Go to the tab Phones Config.	No phone slot exists.
		2	Add a phone slot from tagset	The TAF Configuration Generator window appears.
		3	Add a phone configuration.	A new dialog appears to choose the phone configuration.
		4	Choose one of them and submit.	The phone configuration was listed on the phone configurations list and a error appears saying that you need to add a test data.
		5	Configure a test data.	A new dialog appears to choose a test data.
		6	Choose one of them and submit.	A window appears to configure the Test Data Configuration. The description field is show on this window.
		7	Submit the test data.	The test data appears on the Test Data Configuration field.
		8	Submit this tagset.	The slot was created and is enabled. No one error appears.

Tabela D.13: Suíte 003 do TAFPlus2: Testes 034 e 035

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_036	Edit a configured phone slot from tagset.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. One phone slot from tagset exists. Go to the tab Phones Config.	One phone slot is listed and enabled.
		2	Select this phone slot.	The phone slot details appears showing the configurations of this tagset. The Phone and Test Data configuration appears on the details.
		3	Edit this tagset.	The TAF Configuration Generator window appears.
		5	Configure a test data.	A window appears to configure the Test Data Configuration. The description field is show on this window.
		6	Change the description and submit the changes.	The changes were submitted.
		7	Submit this tagset.	The test data was changed and the new description is show on the Phone Slot Details.

Tabela D.14: Suíte 003 do TAFPlus2: Teste 036

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_037	Edit a blank phone slot.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. One blank phone slots exists and is disabled. Go to the tab Phones Config.	One phone slot is listed and disabled. An error is displayed on the error message saying that no one phone slot is enabled.
		2	Select the blank phone slot.	The phone slot details appears showing the configurations of this phone slot: PTF Config (expanded), TAF Main Config and Phone 0 Config. All fields of this configurations are empty.
		3	Add a configuration file to the PTF Config.	The file was loaded and the tags of this file appears on the PTF Config tags. Some tags can be enabled (green circle) and others can be disabled (red circle).
		4	Add a new tag to the PTF Config.	A dialog to enter the name and the value of this tag appears.
		5	Submit a name different from the other tags.	The tag was added and appears on the tag list enabled.
		6	Enable the phone slot on the phone slot list.	No error appears on the error message.

Tabela D.15: Suíte 003 do TAFPlus2: Teste 037

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_038	Create two tags with the same name on the PTF Config of a phone slot.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. One phone slots exists with a PTF Config file at least and is disabled. Go to the tab Phones Config.	One phone slot is listed and disabled. An error is displayed on the error message saying that no one phone slot is enabled.
		2	Select the phone slot.	The phone slot details appears showing the configurations of this phone slot: PTF Config (expanded), TAF Main Config and Phone 0 Config. Some tags are show in the configuration files.
		3	Add a new tag to the PTF Config.	A dialog to enter the name and the value of this tag appears.
		4	Submit the name ”MyTag”and some value.	The tag was added and appears on the tag list enabled.
		5	Add another tag with the same name of the previous but with a different value.	No error was throw. The value of the previous tag changes and now reflects to the value of the last added tag.
		6	Enable the phone slot on the phone slot list.	No error appears on the error message.

Tabela D.16: Suíte 003 do TAFPlus2: Teste 038

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_039	Change the number of phones without changing the configuration files.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. One phone slots exists with a PTF Config file at least and is disabled. This configuration is for one phone. Go to the tab Phones Config.	One phone slot is listed and disabled. An error is displayed on the error message saying that no one phone slot is enabled.
		2	On the Phone Slots list change the number of phones of the phone slot.	The number of phones is updated on the phone slots list.
		3	Select the phone slot.	The phone slot details appears showing the configurations of this phone slot: PTF Config (expanded), TAF Main Config and Phone 0 Config. Some tags are show in the configuration files.
		4	Open the PTF Config.	Verify that the tag TTAF_NUMBER_OF_PHONES reflects the new value and that new Phone_x_Config was/were added on the configuration details.
		5	Enable the phone slot on the phone slot list.	No error appears on the error message.

Tabela D.17: Suíte 003 do TAFPlus2: Teste 039

SUITE NAME: **MANAGING PHONE SLOTS (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_040	Create a new tagset phone slot with SIM Card configuration.	1	Having a properly configured execution version profile and an integration execution configuration with one group with test cases. No one phone slot exists. Go to the tab Phones Config.	No phone slot exists.
		2	Add a phone slot from tagset	The TAF Configuration Generator window appears.
		3	Add a phone configuration.	A new dialog appears to choose the phone configuration.
		4	Choose one of them and submit.	The phone configuration was listed on the phone configurations list and a error appears saying that you need to add a test data.
		5	Configure a test data.	A new dialog appears to choose a test data.
		6	Choose one of them and submit.	A window appears to configure the Test Data Configuration. The description field is show on this window.
		7	Submit the test data configuration changes.	The test data appears on the Test Data Configuration field.
		8	Configure a SIM Card data.	A dialog asking the user name and password appears.
		9	Cancel the SIM Card data configuration and close the dialog.	The TAF Configuration Generator window appears.
		10	Submit this tagset.	The slot was created and is enabled. No one error appears on the error message.

Tabela D.18: Suíte 003 do TAFPlus2: Teste 040

SUITE NAME: **MANAGING EXECUTION VERSION PROFILES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_050	Create a basic execution version profile.	1	Without Execution Version Profiles, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list empty.
		2	Select the New option and fill the Profile Name and PTF Version, then submit the changes.	The EVP was created and now it's possible to create an execution configuration (stress/integration).
		3	Click on the action Create a new execution version profile.	Now, the EVP‘s list has the previously created EVP as the unique EVP available.
		4	Cancel the creation.	The window was closed.
TP2_TEST_051	Create an EVP with TAF.	1	Without Execution Version Profiles, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list empty.
		2	Select the New option and fill the Profile Name Check to use TAF and submit a directory where TAF will be found, and a directory to PTF Version, then submit the changes.	The EVP was created and now it's possible to create an execution configuration (stress/integration).
		3	Click on the action Create a new execution version profile.	Now, the EVP‘s list has the previously created EVP as the unique EVP available.
		4	Cancel the creation.	The window was closed.

Tabela D.19: Suíte 004 do TAFPlus2: Testes 050 e 051

SUITE NAME: **MANAGING EXECUTION VERSION PROFILES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_052	Create an EVP with TAF and with at least one patch.	1	Without Execution Version Profiles, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list empty.
		2	Select the New option and fill the Profile Name Check to use TAF and submit a directory where TAF will be found, and a directory to PTF Version. Import at least one patch too and then submit the changes.	The patch was imported and is checked on this EVP. The EVP was created and now it’s possible to create an execution configuration (stress/integration).
		3	Click on the action Create a new execution version profile.	Now, the EVP‘s list has the previously created EVP as the unique EVP available.
		4	Cancel the creation.	The window was closed.
TP2_TEST_053	Create two EVP‘s with the same name.	1	Without Execution Version Profiles, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list empty.
		2	Select the New option and fill the Profile Name and PTF Version, then submit the changes.	The EVP was created and now it’s possible to create an execution configuration (stress/integration).
		3	Click on the action Create a new execution version profile.	Now, the EVP‘s list has the previously created EVP as the unique EVP available.
		4	Select the New option and create a EVP with the same name as the previous.	An error will be throw because it’s impossible to create two EVP‘s with the same name.

Tabela D.20: Suíte 004 do TAFPlus2: Testes 052 e 053

SUITE NAME: **MANAGING EXECUTION VERSION PROFILES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_054	Remove a patch that is being used by others EVP‘s.	1	With two EVP‘s with at least one patch that is being used by one of this EVP‘s, create a new execution profile.	The Execution Version Profile Configuration window appears with the EVP‘s list has two EVP‘s.
		2	Select the EVP that isn't using the patch and edit it.	The Edit Profile window appears.
		3	Remove the patch on the Patches list.	An warning will be throw saying that this patch cannot be removed because it is being used by another EVP.
		4	Close the Execution Version Profile Configuration window and cancel the edition of this EVP.	The window was closed.
TP2_TEST_055	Remove an EVP.	1	With two EVP‘s, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list has two EVP‘s.
		2	Remove the first EVP.	The EVP was removed and doesn't appears on the EVP‘s list.
		3	Close the Execution Version Profile Configuration window and cancel the edition of this EVP.	The window was closed.
TP2_TEST_056	Remove all EVP‘s	1	With two EVP‘s, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list has two EVP‘s.
		2	Remove all EVP‘s	All EVP‘s were removed and now it's impossible to create execution configurations (stress/integration).
		3	Close the Execution Version Profile Configuration window and cancel the edition of this EVP.	The window was closed.

Tabela D.21: Suíte 004 do TAFPlus2: Testes 054, 055 e 056

SUITE NAME: **MANAGING EXECUTION VERSION PROFILES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_057	Edit an EVP to use TAF and change the name of it.	1	With an EVP that uses only PTF, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list has one EVP.
		2	Select the EVP and edit it’s configurations to use TAF. Change the name of it too and submit the changes.	The EVP was renamed and now uses TAF.
		3	Again, click to create a new EVP.	The Execution Version Profile Configuration window appears with the EVP‘s list has two EVP‘s.
		4	Edit the unique EVP.	TAF version is show on the TAF field.
TP2_TEST_058	Copy an EVP	1	With an EVP that uses only PTF, create a new execution version profile.	The Execution Version Profile Configuration window appears with the EVP‘s list has one EVP.
		2	Select the EVP and copy it.	A dialog to enter the name of the EVP appears with the sugestion: Copy of..
		3	Enter a different name and submit.	The EVP was renamed and the new name is reflected on the EVP‘s list.

Tabela D.22: Suíte 004 do TAFPlus2: Testes 057 e 058

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_070	Re-launching an integration execution	1	Configure an integration execution with one test group. The test group must have one test case that will fail and one that will pass and the test group itself must be enabled. The test group must be assigned to at least 3 execution attempts.	
		2	Launch the previous configured integration execution.	The test cases must run in the same order of the test group's test cases list. After running all test cases for the first attempt, only the not-passed test case should run for the second execution attempt. And so on, until the not-passed test cases ran as many execution attempts as the test group is assigned. Verify also that there is the possibility of online reporting to the Test Central.
		3	Wait until the execution ends and re-launch the previous configured integration execution.	While the execution is running, change the test group's test cases order and/or enable a test case and/or disable a test case. Verify that the current execution follow these changes.
		4	Wait until the execution ends.	Verify that it is possible to report to the Test Central the results.

Tabela D.23: Suíte 005 do TAFPlus2: Teste 070

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_071	Running a test case that will pass.	1	Configure an integration execution with one test group. The test group must have one test case that will pass. The test group must be enabled. The test group must be assigned to at least 1 execution attempt.	
		2	Launch the previous configured integration execution.	Only one test case is running. Verify that there is the possibility of online reporting to the Test Central.
		3	Wait until the execution ends.	The session of this execution appears on the screen with the execution results and the possibility to export logs, export xls, report to test central and view the stack trace (if an error was trow). Verify that the result is PASSED and does not exists a stack trace of this execution.

Tabela D.24: Suíte 005 do TAFPlus2: Teste 071

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_072	Running a test case that will fail and view stack trace.	1	Configure an integration execution with one test group. The test group must have one test case that will fail. The test group must be enabled. The test group must be assigned to at least 1 execution attempt.	
		2	Launch the previous configured integration execution.	Only one test case is running. Verify that there is the possibility of online reporting to the Test Central.
		3	Wait until the execution ends.	The session of this execution appears on the screen with the execution results and the possibility to export logs, export xls, report to test central and view the stack trace (if an error was trow). Verify that the result is FAILED and that it's possible to view the stack trace.
		4	Select the test and view the stack trace.	The stack trace is expanded and show on the screen.

Tabela D.25: Suíte 005 do TAFPlus2: Teste 072

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_073	Run an execution with a TAF EVP, a phone slot configured for PTF and some TAF test cases	1	Having a previously configured execution with a TAF EVP, a phone slot configured only for PTF tests and at least one test group with some enabled TAF test cases, start the execution.	A dialog is shown stating that the user has selected an EVP with TAF and at least one of the phone slots is only configured to run PTF tests and asks the user if he/she wants to proceed.
		2	The user chooses to proceed.	The execution starts but can't run the TAF test cases because there is missing configuration.
		3	Wait for the execution to finish and check the session results page and PTF logs of the TAF tests.	In the session results page all the auto execution times of the TAF test cases must be zero. In their PTF logs there must have a 'MissingConfigurationException' when the test cases are being started.

Tabela D.26: Suíte 005 do TAFPlus2: Teste 073

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_074	Verify that if an execution that needs one-configured phone can run with a two-configured phone slot with only one phone connected	1	Configure an execution with one test group and without phone slots. The test group must have at least three test cases that need only one phone (at least one of these test cases must pass) and at least one test that needs two phones to run.	The ”Start Execution”action is disabled and the reason is that the execution has no enabled phone slot.
		2	Go to the ”Phones Config”tab and add a new phone slot. Configure properly the created phone slot for two phones, and after enable it.	The ”Start Execution”action is enabled (the execution can run) and no warning is shown in the ”Test List”tab.
		3	In the ”Test List”tab, disable any test case that need more than one phone to run.	In the ”Phones Config”tab, the number of expand bars for the phone config files must be the same at the number of configurable phones that is shown on the spinner. The ’Start Execution’ action is still enabled (the execution can run) and no warning is shown in the ”Test List”tab.
		4	Make sure only the phone under test is connected to the computer and start the execution.	The execution runs properly and yield the expected results.

Tabela D.27: Suíte 005 do TAFPlus2: Teste 074

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_075	Verify the session results of a INTEGRATION execution to xls exporting	1	Create and configure an integration execution with one test group. It must have tests that will yield the following results: at least one PASSED and other non-passed (such as CANCELLED, NOT_RAN, PASSED_BUT_NEED_ACTION, UNKNOWN, FAILED, NOT_CONFIGURED, BLOCKED).	
		2	Change the test group number of attempts to 2 and run the execution, keeping track of the executed tests sequence.	
		3	Open the integration execution session tab and try to export the session results by clicking the 'Export to XLS' button.	A system file dialog has opened.

Tabela D.28: Suíte 005 do TAFPlus2: Teste 075

Continuação do Teste 75				
SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
		4	The user selects the new xls file name and presses the 'Save' button.	On the chosen system file location, the user can find the new xls created. He can open the file and see all the test results on the integration execution session correctly exported, sorted by start time. He can verify the correct information about the Date, Number of tests, Auto Elapsed Time, Total Time, Test Central Tester and Test Central Suite, all with the same values from the execution session resutls on TAFPlus2. Each test case result is shown in details on the test results table. This table contains the Test Case name, the Automatic Time, the Automatic Result, the Manual Time, the Manual Result, the Elapsed Time and the Failure Details informations. Each test is shown in their correspondent color on the spreadsheet. The 'Failed' and 'Not Configured' tests have the exception stack trace as the value of their Failure Details column. The 'Passed', 'Passed But Need Action', 'Unknown', 'Blocked', 'User Cancelled', 'TAFPlus internal error' and 'Not Ran' tests have 'NA' as the value of their Failure Details column.

Tabela D.29: Suíte 005 do TAFPlus2: Teste 075 (continuação)

Continuação do Teste 75				
SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
		5	The user closes the exported file, and turn again to TAFPlus2, on the previously generated integration execution session tab. He/she changes the consolidated results of some non-passed tests and define the manual effort to them. After that, he/she clicks the 'Export to XLS' button again.	The system file dialog has opened.
		6	The user selects the previously generated xls file and presses the 'Save' button.	Another dialog appears asking for the user if he really wants to overwrite the previously created file.
		7	The user chooses to overwrite the file.	The user opens the overwritten file and see all the test results on the stress execution session correctly exported. He can verify the correct updated information about the Auto Elapsed Time and Total Time. Each test case result is shown in details on the test results table. The updated test results are being shown with their correct status and color, and with the Manual Time, Manual Result and Elapsed Time following the updated values from the execution session resutls on TAFPlus2.

Tabela D.30: Suíte 005 do TAFPlus2: Teste 075 (continuação)

Continuação do Teste 75				
SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
		8	The user closes the exported file, and turn again to TAFPlus2, on the previously generated integration execution session tab. He changes the consolidated results of some non-passed tests and define the manual effort to them. After that, it clicks the 'Export to XLS' button again.	The system file dialog has opened.
		9	The user selects the previously generated xls file and presses the 'Save' button.	Another dialog appears asking for the user if he really wants to overwrite the previously created file.
		10	The user chooses to not overwrite the file.	The user opens the previously generated file and see that it has no changes comparing with the last time that the user verified it.

Tabela D.31: Suíte 005 do TAFPlus2: Teste 075 (continuação)

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_076	Runing an integration execution with three groups (two enabled and one disabled).	1	Configure an integration execution with three test groups. One test group has one test case and must be disabled. The other two test groups have one test case each one. One of this test cases will fail, and the other will pass. Set execution attempts to 3.	
		2	Launch the previous configured integration execution.	The test cases must run in the same order of the test group's test cases list. After running all test cases for the first attempt, only the not-passed test case should run for the second execution attempt. And so on, until the not-passed test cases ran as many execution attempts as the test group is assigned. Verify also that there is the possibility of online reporting to the Test Central.
		3	Wait until the execution ends.	The session of this execution appears on the screen with the execution results and the possibility to export logs, export xls, report to test central and view the stack trace (if an error was trow). The result of one test case is passed and don't exists a stack trace to it. The result of the other test case is failed and exists a stack trace.

Tabela D.32: Suíte 005 do TAFPlus2: Teste 076

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_077	Run an integration execution with one group and cancel the execution.	1	Configure an integration execution with one group. This group has one test case and is enabled.	
		2	Launch the previous configured integration execution.	The test case start ruinng. Verify also that there is the possibility of online reporting to the Test Central.
		3	Cancel the execution.	The execution was cancelled and the session of this execution appears on the screen with the execution results and the possibility to export logs, export xls, report to test central and view the stack trace (if an error was trow). The result of one test case is cancelled and does not exists a stack trace.
TP2_TEST_078	Change the test case runtime order while a execution.	1	Configure an integration execution with one group. This group has four test cases and is enabled.	
		2	Launch the previous configured integration execution.	The test case start ruinng. Verify also that there is the possibility of online reporting to the Test Central.
		3	Move the last test case to be the second test case to execute.	Verify that the new order is reflected on the execution and the next test to be executed is the last test case.
		4	Wait until the execution ends.	The session of this execution appears on the screen with the execution results and the possibility to export logs, export xls, report to test central and view the stack trace (if an error was trow). The results of test cases appears on this screen.

Tabela D.33: Suíte 005 do TAFPlus2: Testes 077 e 078

SUITE NAME: **TEST CASES EXECUTION CONTROL (SUITE 005)**				
Test ID	Test Description	Step	Step description	Expected Results
TP2_TEST_079	Run integration execution with groups priorities.	1	Configure an integration execution with three groups. One group has one test case and must be disabled. The other two groups have one test case each one. One of this test cases will fail, and the other will pass. The first group has priority 4 and the second has priority 0.	
		2	Launch the previous configured integration execution.	The test cases must run in the same order of the test group's test cases list. The test case of the group with less priority runs first of the other group. Verify also that there is the possibility of online reporting to the Test Central.
		3	Wait until the execution ends.	The session of this execution appears on the screen with the execution results and the possibility to export logs, export xls, report to test central and view the stack trace (if an error was trow). The results of test cases appears on this screen.
TP2_TEST_080	Run execution without a phone connected.	1	Configure an integration execution with one group. This group has one test case that is enabled. No phone is connected.	
		2	Launch the previous configured integration execution.	An error is throw and the session results appears on the screen. One test case was marked as Not Run.
		3	Select the test and view the stack trace.	No stack trace is show.

Tabela D.34: Suíte 005 do TAFPlus2: Testes 079 e 080

D.2 Suítes de casos de teste manuais do TAFStudio

As suítes de casos de teste manuais para o TAFStudio estão sendo mostradas nas tabelas D.35 até D.55.

SUITE NAME: **CREATING AND REMOVING TCs AND TSs (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_001	Creating test cases.	1	Create a test case by clicking on the ”Creates test case”icon and set the name as ”MyTest”.	The test case is created, appears on the test cases view and the test case editor is opened with no one step.
		2	Create another test case and set the name as ”MyTest2”.	The test case is created and appears on the test cases view. The test case editor is opened with no one step and the editor of the last created test case has focus.
		3	Close the first created test case.	The first test case editor is closed and the focus still in the second test case editor.
TS_TEST_002	Creating test suite with test cases.	1	Create a test case by clicking on the ”Creates test case”icon and set the name as ”MyTest”.	The test case is created , appears on the test case view and the test case editor is opened with no one step.
		2	Create a test suite by clicking on the ”Creates test suite”icon and set the name as ”MySuite”.	The test suite is created and appears on the test cases view.
		3	Move the created test case to inside the test suite previously created.	The test case was moved inside the test suite and doesn’t appears on the test cases view anymore.
		4	Select the test suite and then add another test case by clicking on the ”Creates test case”icon and set the name as ”MyTest2”.	The test case is created, doesn’t appears on the test cases view but appears inside the test suite ”MySuite”and the test case editor is opened with no one step.

Tabela D.35: Suíte 001 do TAFStudio: Testes 001 e 002

SUITE NAME: **CREATING AND REMOVING TCs AND TSs (SUITE 001)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_003	Creating and removing test suites.	1	Create a test suite and then creates three test cases to this test suite.	The suite is created and appears on the test cases view. The three test cases appears inside the test suite and were opened on the test case editor.
		2	Delete the test suite.	A dialog appears and says that if you delete the test suite, all test cases will be deleted too.
		3	Accept the dialog.	The test suite and all test cases were deleted and the suite doesn't appears on the test cases view.
TS_TEST_004	Trying to create a test case without name.	1	Create a test case by clicking on the "Creates test case"icon and in the name only press space.	A message appears saying that this name is invalid on this platform.
		2	Close the dialog.	The dialog was closed and the test case was not created.
TS_TEST_005	Removing three test cases.	1	With three test cases, select all of them.	
		2	Delete the three test cases.	The three test cases were deleted and don't appears on the test cases view.
TS_TEST_006	Removing a test case inside a test suite and another outside the suite.	1	With one test case outside the test suite and one test suite with only one test case inside it, select the test inside the suite and the test outside the suite.	
		2	Delete the two test cases.	The two test cases were deleted and now the test suite doesn't has no one test case. The test case outside the test suite doesn't appears on the test cases view anymore.

Tabela D.36: Suíte 001 do TAFStudio: Testes 003, 004, 005 e 006

SUITE NAME: **MOVING TCs AND TSs (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_020	Moving test suites inside a test case.	1	With two test suites and one test case on the test cases view.	
		2	Select the two test suites and move them inside the test case.	Nothing occurs and the test suites and test cases still on the same position before the move operation.
TS_TEST_021	Moving and removing test cases between test suites.	1	With a test suite with three test cases.	
		2	Create another test suite.	The test suite is created and appears on the test cases view.
		3	Move two tests of the first suite to the previously created test suite.	The tests were moved and now the first suite has one test and the last has two tests.
		4	Remove the last suite.	The suite was removed and now only remains the first suite with one test case.
TS_TEST_022	Moving test cases inside a test suite.	1	With a test suite with three test cases.	
		2	Move the first of them to the middle of the others.	The test case appears on the middle of the others inside the test suite.
		3	Move the last test case to the first position.	The test case appears on the first position inside the test suite.
TS_TEST_023	Moving test cases on the test cases view.	1	With three test cases on the test cases view.	
		2	Move the first of them to the last position of the list.	The test case appears on the last position of the test cases view.
		3	Move the last test case to the first position.	The test case appears on the first position of the test cases view and now the test case order is the same as before the moving operations.

Tabela D.37: Suíte 002 do TAFStudio: Testes 020, 021, 022 e 023

SUITE NAME: **MOVING TCs AND TSs (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_024	Moving a test suite inside another test suite.	1	With two test suites on the test cases view.	
		2	Select the first suite and move it inside the other test suite.	Nothing occurs and the test suites still on the same position before the move operation.
TS_TEST_025	Moving a test case inside a test suite.	1	With one test suite without test cases and one test case on the test cases view.	
		2	Move the test case inside the test suite.	The test case is moved and now appears only inside the test suite.
TS_TEST_026	Moving two test cases in the test cases view to a suite.	1	With two test cases on the test cases view and one test suite without test case.	
		2	Select the two test cases and move it inside the test suite.	The test cases were moved and now the last suite has two test cases and no one test cases are on the test cases view.

Tabela D.38: Suíte 002 do TAFStudio: Testes 024, 025 e 026

SUITE NAME: **MOVING TCs AND TSs (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_027	Moving a test case to a suite and test cases between test suites.	1	With two suites, one empty and the other with one test case. And, one test case on the test cases view.	
		2	Move the test case on the test cases view to the empty suite.	The test case is moved and now appears only inside the test suite.
		3	Select the test case on the first suite and move it to the second suite.	The test case was moved and appears on the second suite but doesn't appears on the first suite anymore. The second suite now has two test cases.
		4	Select the test case on the second suite and move it to the first suite.	The test case was moved and appears on the first suite but doesn't appears on the second suite anymore. The first suite now has only one test and the second suite has one test too.
TS_TEST_028	Moving all test cases of a suite and one test case in the test cases view to another suite with test cases.	1	With two suites with more than one test case each one and one test case on the test cases view.	
		2	Select all test cases of the first suite and the test on the test cases view and move it to the second suite.	All test cases were moved and now the first test suite is empty and no one test case appears on the test cases view.

Tabela D.39: Suíte 002 do TAFStudio: Testes 027 e 028

SUITE NAME: **MOVING TCs AND TSs (SUITE 002)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_029	Moving one test case of the test cases view and one inside a test suite to another test suite that is empty.	1	With one test case on the test cases view, one suite with one test case and one empty suite.	
		2	Select all test cases and move it to the empty suite.	All test cases were moved and now the first test suite is empty, the test cases view only has two test suites and the last test suite has two test cases.

Tabela D.40: Suíte 002 do TAFStudio: Teste 029

SUITE NAME: **OPEN AND RENAME TCs AND TSs (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_030	Renaming test cases.	1	With more than two test cases created and don't opened, select one of them to rename.	A dialog appears to enter the name of the test case. In this dialog, the old name appears.
		2	Submit the name "MT1"	The test was renamed and appears on the test cases view.
		3	Select other test to rename.	A dialog appears to enter the name of the test case. In this dialog, the old name appears.
		4	Submit the name "MT2"	The test was renamed and appears on the test cases view after the first test case.
TS_TEST_031	Rename a test case more than once.	1	With more than two test cases created and don't opened, select one of them to rename.	A dialog appears to enter the name of the test case. In this dialog, the old name appears.
		2	Submit the name "MT1"	The test was renamed and appears on the test cases view.
		3	Select the same test case and select to rename.	A dialog appears to enter the name of the test case. In this dialog, the old name (now "MT1") appears.
		3	Submit the name "MT11"	The test was renamed and appears on the test cases view.
TS_TEST_032	Open two test cases.	1	With more than two test cases select one of them and open it.	The test case editor is opened.
		2	Close the editor.	The editor is closed.
		3	Open another test case.	The test case editor is opened.
		4	Open another test case.	The test case editor is opened and now two test cases are open.
		5	Close the two opened test cases.	No one test case is open.

Tabela D.41: Suíte 003 do TAFStudio: Testes 030, 031 e 032

SUITE NAME: **OPEN AND RENAME TCs AND TSs (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_033	Open renamed test cases.	1	With one test case on the test cases view, open this test case.	The test case editor is opened.
		2	Close the test case editor.	No one test case is open.
		3	Select to rename this test case.	A dialog appears to enter the name of the test case. In this dialog, the old name appears.
		4	Submit the name ”MT1”	The test was renamed and appears on the test cases view.
		5	Open the renamed test case.	The test case editor is opened and now the name reflects to the new name.
TS_TEST_034	Rename a test suite.	1	With one test suite on the test cases view.	
		2	Select the suite and then to rename it.	A dialog appears to enter the name of the suite and the old name appears on the name field.
		3	Submit the name ”MS”	The test suite was renamed and the new name appears on the test cases view.
TS_TEST_035	Renaming two test suites at the same time.	1	With two test suites on the test cases view.	
		2	Select the two suites and rename them with two distinct names.	A dialog appears to enter the name of the first suite and the old name appears on the name field.
		3	Submit the name ”RS1”	A dialog appears to enter the name of the second suite and the old name appears on the name field.
		4	Submit the name ”RS2”	Now, the two suites were renamed and appears on the test cases view.

Tabela D.42: Suíte 003 do TAFStudio: Testes 033, 034 e 035

SUITE NAME: **OPEN AND RENAME TCs AND TSs (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_036	Rename an opened test case.	1	With one test case on the test cases view, open it.	The test case editor was opened.
		2	Select to rename this test case.	A dialog appears to enter the name of the test case. In this dialog, the old name appears.
		3	Submit the name ”RS1”	The test was renamed and appears on the test cases view but the name on the test case editor doesn’t changes.
		4	Close the test case editor.	The test case editor was closed.
TS_TEST_037	Open two test suites at the same time.	1	With two test suites on the test cases view.	
		2	Select the two suites and open it	The two suites were opened and two suite editors appears.
TS_TEST_038	Open a suite and a test case inside it at the same time.	1	With one test suite with one test case on the test cases view.	
		2	Select the suite and the test case and open it.	The suite was opened and the test case too. A suite editor appears and a test case editor too. The suite editor appears first of the test case editor.

Tabela D.43: Suíte 003 do TAFStudio: Testes 036, 037 e 038

SUITE NAME: **OPEN AND RENAME TCs AND TSs (SUITE 003)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_039	Renaming two test suites at the same time with the same name.	1	With two test suites on the test cases view.	
		2	Select the two suites and rename them with the same name.	A dialog appears to enter the name of the first suite.
		3	Submit the name ”RS1”	A dialog appears to enter the name of the second suite.
		4	Submit the name ”RS1”	It‘s impossible to rename this suite with this name because the first suite has this name.
		5	Close the rename dialog.	The dialog was closed and on the test cases view only the first suite was renamed.
TS_TEST_040	Renaming a test suite and them create a test suite with the old name of the renamed suite.	1	With one test suite with the name ”Old-Suite”on the test cases view.	
		2	Rename this test suite.	A dialog appears to enter the name of the suite and the old name appears on the name field.
		3	Submit the name ”RS1”	The suite was renamed and appears on the test cases view.
		4	Create a new suite.	A dialog appears to enter the name of the suite.
		5	Submit the name ”OldSuite”.	The suite was created, appears on the test case view and is opened.

Tabela D.44: Suíte 003 do TAFStudio: Testes 039 e 040

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_050	Record some steps.	1	With a test case opened. The actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled. No one phone is connected to the computer.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Connect a phone.	A dialog Phone Connection appears on the screen and a new phone will appears on the views Phones.
		4	Look the test case editor.	One step to the idle screen appears on this editor.
		5	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.45: Suíte 004 do TAFStudio: Teste 050

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_051	Pause and resume test case recording.	1	With a test case opened and a phone connected. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute some steps on the phone.	This steps are show in the test case editor.
		4	Pause the recording.	The Pause/Resume Recording is toggled.
		5	Execute some steps on the phone.	This steps aren't show in the test case editor.
		6	Resume the recording.	The Pause/Resume Recording isn't toggled.
		7	Execute some steps on the phone.	This steps are show in the test case editor.
		8	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.46: Suíte 004 do TAFStudio: Teste 051

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_052	Pause test case recording.	1	With a test case opened and a phone connected. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute some steps on the phone.	This steps are show in the test case editor.
		4	Pause the recording.	The Pause/Resume Recording is toggled.

Tabela D.47: Suíte 004 do TAFStudio: Teste 052

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_053	Record some steps into a group.	1	With a test case opened and a phone connected. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor.
		4	Press the action Create new step group.	A dialog appears asking for the group description.
		5	Enter a group description and submit.	The group was created and appears on the test case editor without steps.
		6	Execute three steps on the phone.	This steps are show in the test case editor, inside the new group previously created.
		7	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.48: Suíte 004 do TAFStudio: Teste 053

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_054	Record some steps without phone connected.	1	With a test case opened. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled. No one phone is connected to the computer.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps aren't show in the test case editor.
		4	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.49: Suíte 004 do TAFStudio: Teste 054

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_055	Record some steps into two groups.	1	With a test case opened. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Connect a phone and then click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor.
		4	Press the action Create new step group.	A dialog appears asking for the group description.
		5	Enter a group description and submit.	The group was created and appears on the test case editor without steps.
		6	Execute three steps on the phone.	This steps are show in the test case editor, inside the new group previously created.
		7	Press the action Create new step group.	A dialog appears asking for the group description.
		8	Enter a group description (different from the first one) and submit.	The group was created and appears on the test case editor without steps.
		9	Execute three steps on the phone.	This steps are show in the test case editor, inside the new group previously created.
		10	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.50: Suíte 004 do TAFStudio: Teste 055

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_056	Disconnect a phone while recording.	1	With a test case opened and a phone connected. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor.
		4	Disconnect the phone from USB.	A dialog appears indicating that the phone is disconnected and the phone doesn't appears on the phones view anymore.
		5	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.51: Suíte 004 do TAFStudio: Teste 056

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_057	Record steps inside and outside groups.	1	With a test case opened and a phone connected. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor.
		4	Press the action Create new step group.	A dialog appears asking for the group description.
		5	Enter a group description and submit.	The group was created and appears on the test case editor without steps.
		6	Execute three steps on the phone.	This steps are show in the test case editor, inside the new group previously created.
		7	Press the action Stop current step group.	The step group was stopped.
		8	Execute three steps on the phone.	This steps are show in the test case editor.
		9	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.52: Suíte 004 do TAFStudio: Teste 057

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_058	Record steps to more than one test case at same time.	1	With two test cases opened and a phone connected. And the actions Start/Stop Recording, Create new step group and Stop current step group are enabled. And the action Pause/Resume Recording is disabled.	
		2	Click on the action Start/Stop Recording on the first test case editor.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor of the first test case.
		4	Change to the test case editor of the second test case.	The editor of the second test case appears on the screen.
		5	Click on the action Start/Stop Recording on the first test case editor.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		6	Execute three steps on the phone.	This steps are show in the test case editor of the second test case and on the first test case too.
		7	Stop recording on both test case editors.	The action Pause/Resume Recording is disabled on both editors and the recording stops.

Tabela D.53: Suíte 004 do TAFStudio: Teste 058

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_059	Stop a group recording without having a group.	1	With a test case opened, a phone connected. , the actions Start/Stop Recording, Create new step group and Stop current step group enabled and the action Pause/Resume Recording disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor.
		4	Press the action Stop current step group.	A dialog appears on the screen saying that doesn't has a group to be stopped.
		5	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.

Tabela D.54: Suíte 004 do TAFStudio: Teste 059

SUITE NAME: **RECORDING TEST CASES (SUITE 004)**				
Test ID	Test Description	Step	Step description	Expected Results
TS_TEST_060	Record steps with the recording stopped.	1	With a test case opened, a phone connected. , the actions Start/Stop Recording, Create new step group and Stop current step group enabled and the action Pause/Resume Recording disabled.	
		2	Click on the action Start/Stop Recording.	Now, the action Pause/Resume Recording is enabled and the recording starts.
		3	Execute three steps on the phone.	This steps are show in the test case editor.
		4	Stop recording.	The action Pause/Resume Recording is disabled and the recording stops.
		5	Execute three steps on the phone.	This steps aren't show in the test case editor.

Tabela D.55: Suíte 004 do TAFStudio: Teste 060

ANEXO E – Faltas introduzidas, Execuções de Testes e Diagnósticos

E.1 Faltas introduzidas no TAFStudio e execuções manuais de suítes de testes

A Tabela E.1 demonstra em quais blocos foram introduzidas faltas. Ela faz também uma breve avaliação sobre o diagnóstico emitido pela ferramenta após a a execução manual dos casos de teste da suíte impactada pela falta. As tabelas E.2 e E.3 trazem informações sobre as execuções manuais de suítes de testes, mostrando os testes que foram executados para cada falta, os resultados dos testes e a porcentagem de blocos que necessitam ser inspecionados até se encontrar o bloco com falta. Com o número da suíte e do defeito da Tabela E.1, pode-se visualizar as informações sobre a execução da suíte de testes nas tabelas E.2 e E.3.

Bloco defeituoso	Suíte	Avaliação do Resultado (Diagnóstico)
NewTestCaseAction.run()	Suíte 001 Defeito 1	O resultado foi ruim, pois um teste executou o bloco com falta mas não levou a uma falha. Isso faz com que o *spectrum* do bloco com falta diminua.
TestCaseCatalog.createTestSuite(String name)	Suíte 001 Defeito 2	O resultado foi excelente, pois apenas os testes que executaram o bloco com falta falharam.
TestCaseCatalog.moveTestCases(Object location, TestCase[] movingTcs)	Suíte 002 Defeito 1	O resultado foi bom, mesmo existindo alguns testes que executaram o bloco com falta e não levaram a falha. O *spectrum* do bloco com falta se manteve alto.
OpenEditorAction.run()	Suíte 003 Defeito 1	O resultado foi excelente, pois precisa-se procurar em apenas um bloco até encontrar o bloco faltoso. Nesse caso o bloco faltoso já foi reportado e o testador não gastará tempo algum.
TestCaseCatalog.renameTestSuite(TestSuite ts, String newName)	Suíte 003 Defeito 2	O resultado foi razoável, pois existiram casos de teste que passaram pelo bloco com falta mas não levaram a falha, e, em apenas um caso de teste, o bloco com falta levou a uma falha. O impacto desse comportamento foi que o bloco com falta não está entre os 2 blocos com maior *spectrum*. Isso aconteceu pois esses 2 blocos foram executados apenas no teste que falhou, implicando em um *spectrum* alto para eles. Já o bloco com falta, foi executado em outros testes que não falharam, diminuindo seu *spectrum*.
TestCaseCatalog.renameTestCase(TestCase tc, String newName)	Suíte 003 Defeito 3	O resultado foi excelente. Todos os testes que executaram o bloco defeituoso falharam.
RecordingSession.pushGroup(StepGroup newCurrentGroup)	Suíte 004 Defeito 1	O resultado foi excelente. Todos os testes que executaram o bloco defeituoso falharam.
RecordingSession.stop()	Suíte 004 Defeito 2	O resultado foi bom. Mas, muitos testes passaram pelo bloco com falta mas não falharam. Mesmo assim a ferramenta conseguiu emitir um diagnóstico muito bom.

Tabela E.1: Faltas inseridas no TAFStudio e avaliações de resultados

Suíte	Testes	Resultado	Blocos a se inspecionar
Suite_001	TS_TEST_001	F	43,10%
Defeito 1	TS_TEST_002	F	
	TS_TEST_003	F	
	TS_TEST_004	P	
	TS_TEST_005	P	
	TS_TEST_006	P	
Suite_001	TS_TEST_001	P	5,94%
Defeito 2	TS_TEST_002	F	
	TS_TEST_003	F	
	TS_TEST_004	P	
	TS_TEST_005	P	
	TS_TEST_006	N/A	
Suite_002	TS_TEST_020	P	28,57%
Defeito 1	TS_TEST_021	P	
	TS_TEST_022	P	
	TS_TEST_023	F	
	TS_TEST_024	P	
	TS_TEST_025	F	
Suite_003	TS_TEST_030	P	1,35%
Defeito 1	TS_TEST_031	P	
	TS_TEST_032	P	
	TS_TEST_033	P	
	TS_TEST_034	P	
	TS_TEST_035	P	
	TS_TEST_036	P	
	TS_TEST_037	F	
	TS_TEST_038	F	
	TS_TEST_039	P	
Suite_003	TS_TEST_030	P	17,14%
Defeito 2	TS_TEST_031	P	
	TS_TEST_032	P	
	TS_TEST_033	P	
	TS_TEST_034	P	
	TS_TEST_035	P	
	TS_TEST_036	P	
	TS_TEST_037	P	
	TS_TEST_038	P	
	TS_TEST_039	P	
	TS_TEST_040	F	
Essa tabela segue na próxima página, na Tabela E.3.			

Tabela E.2: Execução de testes manuais do TAFStudio (1)

Continuação da Tabela E.2.			
Suíte	Testes	Resultado	Blocos a se inspecionar
Suite_003	TS_TEST_030	F	4,35%
Defeito 3	TS_TEST_031	F	
	TS_TEST_032	P	
	TS_TEST_033	F	
	TS_TEST_034	P	
	TS_TEST_035	P	
	TS_TEST_036	F	
	TS_TEST_037	P	
	TS_TEST_038	P	
	TS_TEST_039	P	
	TS_TEST_040	P	
Suite_004	TS_TEST_050	P	4,80%
Defeito 1	TS_TEST_051	P	
	TS_TEST_052	P	
	TS_TEST_053	F	
	TS_TEST_054	P	
	TS_TEST_055	F	
	TS_TEST_056	P	
	TS_TEST_057	F	
	TS_TEST_058	P	
	TS_TEST_059	P	
	TS_TEST_060	P	
Suite_004	TS_TEST_050	P	11,11%
Defeito 2	TS_TEST_051	P	
	TS_TEST_052	P	
	TS_TEST_053	P	
	TS_TEST_054	P	
	TS_TEST_055	P	
	TS_TEST_056	P	
	TS_TEST_057	P	
	TS_TEST_058	P	
	TS_TEST_059	P	
	TS_TEST_060	F	
Média de blocos a se inspecionar: 14,54%			

Tabela E.3: Execução de testes manuais do TAFStudio (2)

E.2 Faltas introduzidas no TAFPlus2 e execuções manuais de suítes de testes

A Tabela E.4 demonstra em quais blocos foram introduzidas faltas. Ela faz também uma breve avaliação sobre o diagnóstico emitido pela ferramenta após a a execução manual dos casos de teste da suíte impactada pela falta. As tabelas E.5 e E.6 trazem informações sobre as execuções manuais de suítes de testes, mostrando os testes que foram executados para cada falta, os resultados dos testes e a porcentagem de blocos que necessitam ser inspecionados até se encontrar o bloco com falta. Com o número da suíte e do defeito da Tabela E.4, pode-se visualizar as informações sobre a execução da suíte de testes nas tabelas E.5 e E.6.

Bloco defeituoso	Suíte	Avaliação do Resultado (Diagnóstico)
RenameExecutionAction.run()	Suíte 001 Defeito 1	O resultado foi bom, pois todos testes que executaram o bloco com falta falharam.
AddGroupDialog.update()	Suíte 002 Defeito 1	O resultado foi ruim, pois muitos testes executaram o bloco com falta mas não levaram a falha. Nesse caso uma diminuição da granularidade pode melhorar o diagnóstico.
PhoneSlot(int noofPhones)	Suíte 003 Defeito 1	O resultado foi ruim, pois muitos testes executaram o bloco com falta mas não levaram a falha. Nesse caso uma diminuição da granularidade não melhorará o diagnóstico, pois o comportamento observado na execução da suíte de testes com essa falta habilitada é uma característica do TAFPlus2.
TAFConfigEditor.addTag()	Suíte 003 Defeito 2	O resultado foi bom, pois todos testes que executaram o bloco com falta falharam.
ExecutionProfilePatchButtonComposite. removePath()	Suíte 004 Defeito 1	O resultado foi excelente. Todos os testes que executaram o bloco defeituoso falharam. O teste 53 não foi executado pois está bloqueado.
Não está validando dados de um EVP corretamente	Suíte 004 Defeito 2	O resultado foi bom, pois o teste que passou pelo bloco com falta levou a uma falha. Esse defeito já estava presente no TAFPlus2 e foi descoberto na elaboração desse trabalho. O teste 53 apenas será executado nessa suíte de testes pois ele sempre levará a uma falha. Para as outras execuções da Suíte 004, o teste 53 está bloqueado. O bloco com defeito descoberto através desse diagnóstico é: EditExecutionVersionProfileWizardPage.keyPressed(org .eclipse.swt.events.KeyEvent)
TestGroup.move(TestCase tc, int currentPosition, int targetPosition)	Suíte 005 Defeito 1	O resultado foi excelente, pois todos os testes que executaram o bloco defeituoso falharam.
TestList.compare(TestGroup tg1, TestGroup tg2)	Suíte 005 Defeito 2	O resultado foi excelente, pois o bloco foi executado em apenas dois testes: 76 e 79. O teste 79 falhou, aumentando o spectrum do bloco. Como os outros blocos foram executados em muitos testes que passaram, o spectrum dos mesmos diminuiu muito, deixando o bloco com falta entre os 3 maiores candidatos de estarem com falta.

Tabela E.4: Faltas inseridas no TAFPlus2 e avaliações de resultados

Suíte	Testes	Resultado	Blocos a se inspecionar
Suite_001	TP2_TEST_001	P	11,29%
Defeito 1	TP2_TEST_002	P	
	TP2_TEST_003	F	
	TP2_TEST_004	F	
	TP2_TEST_005	P	
	TP2_TEST_006	P	
	TP2_TEST_007	F	
	TP2_TEST_008	P	
	TP2_TEST_009	P	
Suite_002	TP2_TEST_010	P	57,56%
Defeito 1	TP2_TEST_011	P	
	TP2_TEST_012	P	
	TP2_TEST_013	P	
	TP2_TEST_014	F	
	TP2_TEST_015	P	
	TP2_TEST_016	P	
Suite_003	TP2_TEST_030	F	74,03%
Defeito 1	TP2_TEST_031	F	
	TP2_TEST_032	P	
	TP2_TEST_033	P	
	TP2_TEST_034	P	
	TP2_TEST_035	P	
	TP2_TEST_036	P	
	TP2_TEST_037	P	
	TP2_TEST_038	P	
	TP2_TEST_039	P	
	TP2_TEST_040	P	
Suite_003	TP2_TEST_030	P	5,19%
Defeito 2	TP2_TEST_031	P	
	TP2_TEST_032	P	
	TP2_TEST_033	P	
	TP2_TEST_034	P	
	TP2_TEST_035	P	
	TP2_TEST_036	P	
	TP2_TEST_037	F	
	TP2_TEST_038	F	
	TP2_TEST_039	P	
	TP2_TEST_040	P	
Essa tabela segue na próxima página, na Tabela E.6.			

Tabela E.5: Execução de testes manuais do TAFPlus2 (1)

Continuação da Tabela E.5.			
Suíte	Testes	Resultado	Blocos a se inspecionar
Suite_004 Defeito 1	TP2_TEST_050	P	3,75%
	TP2_TEST_051	P	
	TP2_TEST_052	P	
	TP2_TEST_053	N/A	
	TP2_TEST_054	F	
	TP2_TEST_055	P	
	TP2_TEST_056	P	
	TP2_TEST_057	P	
	TP2_TEST_058	P	
Suite_004 Defeito 2	TP2_TEST_050	P	5,83%
	TP2_TEST_051	P	
	TP2_TEST_052	P	
	TP2_TEST_053	F	
	TP2_TEST_054	P	
	TP2_TEST_055	P	
	TP2_TEST_056	P	
	TP2_TEST_057	P	
	TP2_TEST_058	P	
Suite_005 Defeito 1	TP2_TEST_070	F	1,12%
	TP2_TEST_071	P	
	TP2_TEST_072	P	
	TP2_TEST_073	P	
	TP2_TEST_074	P	
	TP2_TEST_075	P	
	TP2_TEST_076	P	
	TP2_TEST_077	P	
	TP2_TEST_078	F	
	TP2_TEST_079	P	
	TP2_TEST_080	P	
Suite_005 Defeito 2	TP2_TEST_070	P	0,27%
	TP2_TEST_071	P	
	TP2_TEST_072	P	
	TP2_TEST_073	P	
	TP2_TEST_074	P	
	TP2_TEST_075	P	
	TP2_TEST_076	P	
	TP2_TEST_077	P	
	TP2_TEST_078	P	
	TP2_TEST_079	F	
	TP2_TEST_080	P	
Média de blocos a se inspecionar: 19,88%			

Tabela E.6: Execução de testes manuais do TAFPlus2 (2)

E.3 Casos particulares na introdução de faltas, execução de testes e emissão de diagnósticos

Essa seção demonstra algumas situações particulares para avaliar a ferramenta para localização e diagnóstico de faltas. Foram introduzidas faltas que geram três situações particulares:

1. Uma mesma suíte de testes executar mais de um bloco com falta.

2. Refatorar um código para simular a diminuição de granularidade.

3. Modificar uma suíte de testes para aumentar a precisão do diagnóstico.

A primeira situação foi verificada no TAFStudio. As outras duas foram verificadas no TAFPlus2. A Tabela E.7 demonstra em quais blocos foram introduzidas faltas. Ela faz também uma breve avaliação sobre o diagnóstico emitido pela ferramenta após a a execução manual dos casos de teste da suíte impactada pela falta. Essa tabela também faz uma comparação entre o caso normal e o caso particular demonstrando o ganho ou perda na precisão do diagnóstico. A tabela E.8 traz informações sobre as execuções manuais de suítes de testes, mostrando os testes que foram executados para cada falta, os resultados dos testes e a porcentagem de blocos que necessitam ser inspecionados até se encontrar o bloco com falta. Com o número da suíte e do defeito da Tabela E.7, pode-se visualizar as informações sobre a execução da suíte de testes na tabela E.8.

Bloco defeituoso	Suíte	Avaliação do Resultado (Diagnóstico)
OpenEditorAction.run() + TestCaseCatalog.renameTestCase(TestCase tc, String newName)	TAFStudio Suíte 003 Defeito 1 Defeito 3	O resultado foi razoável. Um dos blocos com falta (TestCaseCatalog.renameTestCase()) aparece entre os 4 blocos com maior *spectrum* resultando em um ótimo dianóstico nesse caso. Já o outro bloco com falta (OpenEditorAction.run()) é apenas o 27º candidato a estar com falta. Seu *spectrum* foi diminuindo pois em muitos testes ele não foi executado e a falta mesmo assim ocorreu (devido ao outro bloco faltante). Ao introduzir as faltas e executar a suíte de testes separadamente era necessário inspecionar de 4 a 13% de blocos até encontrar o bloco com falta. A execução da suíte com as duas faltas habilitadas gerou um pior diagnóstico pois agora precisam ser inspecionados aproximadamente 29% de blocos até encontrar as duas faltas.
AddGroupDialog.update()	TAFPlus2 Suíte 002 Defeito 1 Menor Granularidade	O resultado foi excelente. A classe foi refatorada para diminuir a granularidade dos blocos. Cada condicional (se/senão) foi separado. Assim, apenas ao executar o condicional com falta será gerado um erro e o teste falhará. Essa mudança fez com que o diagnóstico melhorasse muito. Antes dessa modificação precisavam ser inspecionados até 58% dos blocos. Com a modificação necessita-se inspecionar aproximadamente 1% de blocos.
FileBasedAutoActionPage. showFileSelectDialog()	TAFPlus2 Suíte 001 Defeito 1 Suíte Modificada	O resultado foi excelente. Nos testes que criam uma execução foi adicionada a tarefa de abrir a aba ”Automatic Actions”e verificar que os campos de diretórios para exportação estão vazios. Esse novo comportamento fez com que o diagnóstico emitido fosse excelente. Esse novo comportamento fez blocos não faltosos executarem em quase todos os testes e não levarem a falha. Isso diminui o *spectrum* deles e deixou o bloco com falta como um dos maiores candidatos a estar com falta. Antes dessa modificação precisavam ser inspecionados até 26% dos blocos. Com a modificação necessita-se inspecionar aproximadamente 2% de blocos.

Tabela E.7: Casos particulares do TAFStudio e TAFPlus2

Suíte	Testes	Resultado	Blocos a se inspecionar
Suite_003	TS_TEST_030	F	29,35%
Defeito 1	TS_TEST_031	F	
Defeito 3	TS_TEST_032	P	
	TS_TEST_033	F	
	TS_TEST_034	P	
	TS_TEST_035	P	
	TS_TEST_036	F	
	TS_TEST_037	F	
	TS_TEST_038	F	
	TS_TEST_039	P	
	TS_TEST_040	P	
Suite_002	TP2_TEST_010	P	0,59%
Defeito 1	TP2_TEST_011	P	
Menor	TP2_TEST_012	P	
Granularidade	TP2_TEST_013	P	
	TP2_TEST_014	F	
	TP2_TEST_015	P	
	TP2_TEST_016	P	
Suite_001	TP2_TEST_001	P	1,58%
Defeito 1	TP2_TEST_002	P	
Suíte	TP2_TEST_003	P	
Modificada	TP2_TEST_004	P	
	TP2_TEST_005	P	
	TP2_TEST_006	F	

Tabela E.8: Execução de testes manuais dos casos particulares do TAFPlus2 e TAFStudio

Aplicação da técnica de diagnósticos de defeitos de software em ferramentas de execução de testes

Pedro Ghilardi[1], Sérgio Peters[1], José Otávio Carlomagno Filho

[1]Departamento Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

pghilardi@inf.ufsc.br, peters@inf.ufsc.br, jocf83@gmail.com

Abstract. *This work proposes the elaboration of a tool that automates the localization and diagnosis of faults based on the spectrum of test case suites. Some test case suites will be executed with enabled faults. The results of these executions will be stored and subsequently evaluated on four factors: similarity coefficient, test case suite, target software and block granularity. With the introduction of the tool in a test cycle, it is expected to decrease the time spent to locate the faults, because the diagnosis delivered by the tool decreases the quantity of blocks that need to be inspected manually until the faulty block is found.*

Resumo. *Este trabalho propõe a elaboração de uma ferramenta que automatize a localização e diagnóstico de faltas baseada no perfil de execução (spectrum) de suítes de casos de teste. Serão executadas algumas suítes de testes com faltas habilitadas. Os resultados dessas execuções são avaliados quanto a quatro fatores: coeficiente de similaridade, suíte de testes, programa alvo e granularidade de blocos. Com a introdução da ferramenta, espera-se diminuir consideravelmente o tempo gasto para se localizar faltas, pois o diagnóstico emitido pela ferramenta diminui a quantidade de blocos que precisam ser inspecionados manualmente até se encontrar o bloco com falta.*

1. Introdução

A elaboração e execução de testes é uma atividade essencial no ciclo de desenvolvimento de um software. Segundo (HAILPERN; SANTHANAM, 2002), testes são de grande importância devido à crescente complexidade dos programas, menor ciclo de desenvolvimento e grande expectativa de qualidade pelo cliente.

Durante o ciclo de desenvolvimento de um programa, apenas algumas das faltas que afetam o usuário final são localizadas e diagnosticadas (ABREU; ZOETEWEIJ; GEMUND, 2007). Esse cenário implica em uma menor confiabilidade no programa, pois ele provavelmente apresentará alguns erros que não foram localizados durante o desenvolvimento. Isso ocorre devido à grande pressão por parte do cliente para que o programa seja entregue em um curto espaço de tempo. Assim, muitas faltas passam despercebidas, não sendo detectadas com as técnicas de localização e diagnóstico manuais.

Para diminuir o tempo gasto com esse ciclo de testes, pode-se utilizar alguma técnica automatizada de localização e diagnóstico de faltas. Neste trabalho pretende-se

utilizar a técnica baseada no perfil de execução (*spectrum*) de algumas suítes (coleções) de testes. O perfil de execução indica quais partes (blocos) de um programa estavam ativas durante uma determinada execução (ABREU; ZOETEWEIJ; GEMUND, 2007). Observando execuções que finalizaram com sucesso e outras que falharam, é possível identificar quais partes estão correlacionadas com os erros.

Neste trabalho, a parte de monitoramento de uma execução será implementada utilizando Programação Orientada a Aspectos (POA). A POA permite resolver alguns problemas que a Programação Orientada a Objetos (POO) não consegue solucionar adequadamente (KICZALES et al., 1997). Utilizando apenas POO, para registrar a passagem por cada bloco de código seria necessário introduzir esse novo comportamento bloco por bloco, o que tornaria a implementação difícil. Utilizando aspectos, cria-se um módulo separado que adicionará essa funcionalidade em tempo de compilação, gerando um código mais elegante.

A proposta principal deste trabalho é a elaboração de uma ferramenta que faça a localização e diagnóstico de faltas monitorando execuções manuais de casos de teste para dois programas: TAFStudio e TAFPlus2. Esses dois programas criam ou executam testes automatizados para telefones celulares da Motorola Industrial Ltda. Para criar a ferramenta, será feita a análise e projeto do novo comportamento a ser adicionado com programação por aspectos. Após a criação do aspecto será criado um *plug-in* para gerenciar e controlar o rastreamento e também emitir os diagnósticos de uma execução.

Com a ferramenta (aspecto de rastreamento e *plug-in* de controle) implementada, serão introduzidas algumas faltas e executadas algumas suítes de testes para posteriormente avaliar os resultados obtidos. Os resultados provêm de um diagnóstico calculado por um coeficiente de similaridade que informa quais partes do programa têm a maior probabilidade de estarem apresentando faltas (ABREU; ZOETEWEIJ; GEMUND, 2007). Utilizando os resultados obtidos, serão avaliados quatro fatores para verificar se eles afetam a precisão de diagnóstico da técnica. Os fatores são os seguintes:

- **Coeficiente de similaridade**: Serão utilizados dois coeficientes: Ochiai, que é utilizado na biologia molecular (MEYER et al., 2004) e Jaccard, proveniente de algoritmos de *data clustering* (JAIN; DUBES, 1988). Avalia-se se algum dos coeficientes obtém melhores resultados.
- **Suíte de testes**: Avaliar o impacto no diagnóstico causado por testes que não levam a falhas e outros que executam mais de um bloco do programa com defeito.
- **Programa Monitorado**: Serão monitorados dois programas: TAFPlus2 e TAFStudio. Será verificado se existe diferença na precisão do diagnóstico variando o programa sob análise.
- **Granularidade**: Refere-se ao tamanho dos blocos rastreados. Nesse trabalho um método ou construtor será um bloco. Verifica-se se a diminuição ou aumento da granularidade modifica a precisão da técnica.

Assim, após introduzir a ferramenta em um ciclo de desenvolvimento de testes, espera-se diminuir o tempo gasto para se localizar e diagnosticar faltas, resultando em

um menor ciclo de testes, um programa com maior confiabilidade e um menor tempo de entrega ao mercado (*time-to-market*).

Na seqüência deste artigo, tem-se no Capítulo 2 uma breve introdução sobre Programação Orientada a Aspectos (POA). No Capítulo 3 são expostos conceitos sobre testes e é contextualizado o ambiente de testes utilizado. O capítulo 4 expõe a metodologia para localização e diagnóstico de faltas. O Capítulo 5 descreve o projeto e implementação da ferramenta e no Capítulo 6 são criadas e executadas algumas suítes de testes e avaliados os resultados em relação aos quatro fatores citados anteriormente. No Capítulo 7 resumem-se as conclusões e possibilidades para trabalhos futuros.

2. Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) é um paradigma que pertence ao grupo de paradigmas *Post-Object Programming* (POP). Esses paradigmas pretendem superar as limitações da Programação Orientada a Objetos (POO).

A POO não possibilita uma maneira eficiente de separar áreas de interesse (*concerns*) que ficam dispersas em vários componentes. Segundo (JACOBSON; NG, 2005), uma área de interesse pode ser definida como algo que é de interesse do cliente, usuário final, desenvolvedor ou patrocinador do projeto. Já um componente, no contexto de POO, pode ser uma classe. Essas áreas dispostas em vários componentes são chamadas de áreas de dispersão (*crosscutting concerns*). A POA é utilizada como um complemento à POO para implementar áreas de dispersão gerando um código separado em módulos, reusável e elegante.

Segundo (COLYER et al., 2004), uma área de dispersão necessita de certo comportamento para ocorrer em um ou mais pontos de um fluxo de execução. Mas, o foco principal do programa não é completar o comportamento requerido nesses pontos. Ao programar uma área de dispersão utilizando aspectos, três benefícios chaves serão obtidos: **modularidade**, **encapsulamento** e **abstração**. A modularidade será obtida porque essa funcionalidade estará implementada em um único local: o aspecto. Além disso, os detalhes da implementação estarão encapsulados dentro desse aspecto. Nomeando o aspecto, obtêm-se a abstração desse conceito no sistema. Esses três benefícios diminuem a complexidade de um programa, facilitam a manutenção e possibilitam a reusabilidade de componentes.

2.1. Conceitos Gerais

Nesta seção serão apresentados alguns conceitos importantes de programação por aspectos para a linguagem **AspectJ** que é uma extensão da linguagem Java para se trabalhar com aspectos. O conteúdo desta seção é baseado no livro **Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools** (COLYER et al., 2004) e também no guia de programação do **AspectJ** (XEROX, 1998).

Um conceito importante é o de **pontos de junção** (*join points*). Pontos de junção são eventos no fluxo de execução de um programa. Um exemplo é a instanciação de uma classe ou a execução de um método. Assim, a linguagem AspectJ define o modelo de pontos de junção, o qual define quais eventos serão expostos como pontos de junção

e poderão ser capturados pelo desenvolvedor. Os pontos de junção que serão capturados neste trabalho são a execução de construtores e métodos.

A linguagem também permite a seleção de pontos de junção através de **pontos de corte** (*pointcuts*). Um ponto de corte atua como um filtro aceitando os pontos de junção desejados e bloqueando todos os outros. Uma boa prática é nomear um ponto de corte para proporcionar uma melhor compreensão de código. Na elaboração de pontos de corte utiliza-se também ***wildcards***. Um *wildcard* funciona como uma expressão regular. Ele permite uma maior expressividade na definição de pontos de corte, geralmente representando uma seqüência de caracteres capturada pelo ponto de corte

Depois de capturar os eventos necessários através de pontos de corte, deve-se definir o que fazer nesses pontos. Para definir o comportamento executado antes, depois ou durante os pontos de junção agrupados pelo ponto de corte, a linguagem disponibiliza os **avisos** (*advices*).

Utilizando essas definições da linguagem define-se um **aspecto**, que pode ser utilizado para programar áreas de dispersão modularizadas. Após implementar o aspecto deve-se compor o mesmo com o programa desejado. Neste trabalho essa composição será feita em tempo de compilação.

3. Testes

3.1. Conceitos Gerais

Segundo (MYERS, 2004), testar é o processo de executar um programa com a intenção de localizar erros. Existem três conceitos básicos que devem ser explicados para entender este trabalho. O primeiro é **falha** que é um evento que ocorre quando um serviço difere do esperado. Outro conceito importante é **erro** que é um estado do sistema que pode causar uma falha. Por fim existe a **falta** que é a causa de um erro.

Para se detectar a maior parte dos erros em um programa, existem duas estratégias de testes que são muito utilizadas:

- **Caixa Branca**: Permite que o testador examine a estrutura interna de um programa, isto é, a lógica do código. Os dados a serem utilizados no teste são derivados da lógica do programa. Gera testes exaustivos.
- **Caixa Preta**: Não considera a estrutura interna de um programa, mas busca encontrar circunstâncias pelas quais o programa não se comporta de acordo com sua especificação. Essa estratégia de teste deriva os dados de entrada somente da especificação do programa. Gera testes que verificam apenas as entradas mais prováveis e improváveis. É a estratégia utilizada neste trabalho.

Outro conceito importante sobre testes são os **testes automatizados**, que podem ser reaproveitados, têm uma maior precisão e diminuem os custos de um ciclo de testes a longo prazo, pois não necessitam ser executados manualmente. A grande desvantagem da automatização são os custos iniciais e a não apresentação de resultados imediatos, pois é preciso implementar algum programa para criar e executar os casos de teste.

3.2. Ambiente de testes

Os dois programas que serão monitorados pela ferramenta de localização e diagnóstico de faltas foram criados para possibilitar a automatização de casos de teste para telefones celulares produzidos pela Motorola Industrial Ltda. Nesta seção será contextualizado o ambiente de testes utilizado.

3.2.1. TAF

O **Test Automation Framework (TAF)** é um *framework* criado para automatizar casos de teste dos *softwares* embutidos em telefones celulares produzidos e desenvolvidos pela Motorola Industrial Ltda (KAWAKAMI et al., 2007). Esse *framework* permite a reutilização de testes e a execução automatizada dos mesmos, diminuindo o tempo gasto no ciclo de desenvolvimento. Assim, não é necessário criar um novo teste para cada modelo de telefone a ser testado. Os testes podem ser reaproveitados e apenas algumas modificações são necessárias.

3.2.2. TAFPlus2

O **TAFPlus2** é um programa para facilitar a execução de casos de teste automatizados criados pelo TAF. Esse programa dispõe de uma interface gráfica e permite controlar a execução de vários casos de teste simultaneamente. O TAFPlus2 foi criado pois a execução de casos de teste utilizando apenas o TAF é um pouco complexa e necessita de um certo conhecimento do *framework* e de linguagens de programação. Logo, o objetivo do TAFPlus2 é permitir que usuários sem conhecimento do TAF possam executar casos de testes criados no TAF. Por isso, uma preocupação importante do TAFPlus2 é a usabilidade. O TAFPlus2 é um dos programas que será monitorado pela ferramenta para se localizar e diagnosticar faltas em seus componentes.

3.2.3. TAFStudio

Com o TAFPlus2, a execução de casos de teste automatizados tornou-se mais simples. Mas os testes ainda precisam ser criados utilizando o TAF, o que demanda um conhecimento detalhado do *framework*. O objetivo do **TAFStudio** é permitir que usuários sem nenhum conhecimento em linguagens de programação possam criar casos de teste simplesmente executando os passos diretamente no telefone celular. Assim, o TAFStudio grava os passos executados manualmente e essa sequência de passos pode ser salva como um caso de teste que depois poderá ser executado. O TAFStudio é o outro programa que será monitorado pela ferramenta de localização e diagnóstico de faltas.

4. Metodologia

4.1. Localização e diagnóstico de faltas pelo perfil de execução

A localização e diagnóstico de faltas manual é uma tarefa que demanda um bom tempo para o testador. Após executar um caso de teste ou uma suíte de testes, o testador deve analisar o erro obtido e procurar exaustivamente em cada trecho de código pela falta que causou esse erro. Um programa com muitas linhas de código torna essa tarefa muito custosa, pois examinar cada linha de código demanda muito tempo.

Para diminuir o tempo gasto na localização e diagnóstico manual, pode-se utilizar alguma técnica automatizada. Neste trabalho se utiliza a técnica de **localização e diagnóstico de faltas baseada no perfil de execução** (*Spectrum-Based Fault Localization*). Segundo (REPS et al., 1997), o **perfil de execução** (*spectrum*) é uma coleção de dados que representa o comportamento dinâmico de um programa. Esses dados são coletados em tempo de execução e posteriormente podem ser utilizados para fins de diagnóstico.

Neste trabalho, utiliza-se o **perfil de execução por bloco** (*block hit spectra*). Para cada bloco rastreado se tem um valor associado ao mesmo, indicando se uma execução passou ou não por esse bloco. Um bloco que foi executado terá um valor um, enquanto um bloco que não for executado terá o valor zero. Esses valores serão coletados para um número de suítes de testes executadas e serão armazenados em uma matriz do seguinte formato:

$$MS_{m,n} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & \cdots & x_{m,n} \end{bmatrix} VE_{m,1} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_m \end{bmatrix}$$

Nessa matriz, as colunas representam os diferentes blocos dos programa monitorados. Já as linhas representam as diferentes execuções de testes. Além dessa matriz, existe um vetor de erro, que registra quando uma execução falhou. Se uma execução falhar existirá um número um nesse vetor. Caso contrário, existirá um número zero. Uma execução será considerada falha se o resultado esperado for diferente do resultado obtido. Essa detecção de erros baseada em falhas é uma forma rudimentar de detecção de erros, pois muitas vezes alguns erros nunca levam a uma falha e permanecerão indetectáveis. Neste trabalho, cada caso de teste será considerado uma execução e cada método ou construtor será um bloco.

Após coletar esses dados, a técnica de localização e diagnóstico de faltas tentará encontrar qual bloco mais se assemelha com o vetor de erro. Para achar a semelhança entre a matriz e o vetor de erro pode-se utilizar **coeficientes de similaridade** (JAIN; DUBES, 1988). Neste trabalho, serão utilizados dois coeficientes: **Jaccard e Ochiai**. Segundo (ABREU; ZOETEWEIJ; GEMUND, 2006), o coeficiente de Ochiai apresenta melhores resultados que o coeficiente de Jaccard. Essa conclusão foi obtida submetendo os mesmos blocos defeituosos e testes executados para quatro coeficientes diferentes: Jaccard, Tarantula (JONES; HARROLD; STASKO, 2002), Ample (DALLMEIER; LINDIG; ZELLER, 2005) e Ochiai. Assim, ao avaliar os resultados, concluiu-se que o coeficiente de Ochiai é de 2,4% a 10% mais preciso que o coeficiente de Jaccard (o segundo coeficiente com maior precisão). Em geral, o coeficiente de Ochiai diminui o percentual de blocos que precisam ser inspecionados em até 5%.

Mesmo assim, optou-se por utilizar os dois coeficientes, pois assim pode-se verificar se no âmbito deste trabalho a diferença entre eles é significativa. Mas, para uma dada execução, sempre será considerado o coeficiente de Ochiai como resultado com maior confiabilidade.

A seguir estão sendo mostradas as fórmulas para cálculo dos coeficientes:

$$s_J(j) = \frac{a_{11}(j)}{a_{11}(j)+a_{10}(j)+a_{01}(j)} \quad (1)$$

$$s_O(j) = \frac{a_{11}(j)}{\sqrt{(a_{11}(j)+a_{01}(j))*(a_{11}(j)+a_{10}(j))}} \quad (2)$$

A fórmula (1) refere-se ao coeficiente de Jaccard, utilizado nos algoritmos de *data clustering*. Enquanto a segunda (2), refere-se ao coeficiente de Ochiai, proveniente da biologia molecular. A seguir serão explicados os elementos da fórmula. O elemento a_{11}(j) indica que no bloco j para uma determinada execução, o bloco foi executado e a execução falhou. Já o elemento a_{01}(j) indica que o bloco j não foi executado em uma execução que falhou. Por fim, o elemento a_{10}(j) indica que o bloco j foi executado e a execução passou. Nota-se que para um determinado bloco ter um *spectrum* diferente de zero, o elemento a_{11}(j) deve ser diferente de zero. Isso significa que deve existir pelo menos alguma situação em que o bloco foi executado e o teste que o executou falhou.

Após o cálculo do coeficiente de similaridade, os blocos que obtiverem o maior resultado são os candidatos com a maior probabilidade de estarem apresentando uma falta.

4.2. Vantagens e desvantagens

Uma das vantagens da técnica utilizada neste trabalho é que ela pode ser facilmente integrada em qualquer programa, e não causa perda de desempenho mesmo com recursos limitados de processamento e memória. Outra vantagem é que não é necessário elaborar um modelo do programa, pois ela é uma estratégia de teste do tipo caixa preta.

Segundo (ABREU; ZOETEWEIJ; GEMUND, 2006), essa técnica é muito boa para ser integrada em *software*. Para se localizar faltas em *hardware* deve-se utilizar alguma técnica baseada em modelo. No âmbito de *software*, não se pode utilizar uma técnica baseada em modelo, pois elaborar o modelo de um programa é uma tarefa extremamente complexa.

A grande desvantagem da técnica baseada no perfil de execução é que algumas vezes blocos são classificados com grande probabilidade de estarem com falta, quando na verdade não estão. Isso pode acontecer quando um erro não leva a uma falha. Existem outros pontos importantes a serem avaliados a respeito da técnica, os quais serão explicados em detalhes no Capítulo 6 e Capítulo 7.

5. Análise, Projeto e Implementação

Este capítulo trata da análise, projeto e implementação da ferramenta para localização e diagnóstico de faltas. Essa ferramenta é composta por um **aspecto de rastreamento** (desenvolvido utilizando a IDE Eclipse (ECLIPSE, 2009a) e AspectJ (KICZALES et al., 1997)) e de um ***plug-in* de controle** (desenvolvido utilizando a IDE Eclipse).

Primeiramente será feita a análise, projeto e implementação do aspecto de rastreamento na seção 5.1. Na seção 5.2, os blocos dos programas monitorados serão analisados para verificar se alguns blocos podem ser retirados do rastreamento sem impactar no diagnóstico. A seção 5.3 trata da análise, projeto e implementação do *plug-in*. Ao final, na seção 5.4 mostra-se como foi feita a composição entre o aspecto e os

programas monitorados. Após a composição, a ferramenta está pronta e já pode ser utilizada para localizar e diagnosticar faltas.

5.1. Análise, Projeto e Implementação do aspecto

A análise e projeto do aspecto a ser desenvolvido neste trabalho são baseados no livro **Aspect-Oriented Software Development with Use Cases** (JACOBSON; NG, 2005).

O primeiro passo ao se analisar um novo comportamento, é compreender o que o cliente deseja inserir no sistema. Ao conversar com o cliente, conclui-se que ele deseja inserir o registro de toda chamada de método ou construtor em determinadas classes do programa. Assim, esse é um **requisito não funcional**. Segundo (JACOBSON; NG, 2005), um requisito não funcional representa um atributo de qualidade do sistema. Já um **requisito funcional**, representa algo que o usuário pode fazer no sistema.

Para modelar os requisitos serão utilizados **casos de uso**. Um caso de uso ajuda a entender o comportamento do sistema, pois demonstra o que o mesmo deve fazer como resposta as ações do usuário. Muitos autores não modelam requisitos não funcionais com casos de uso, mas, neste trabalho os requisitos não funcionais serão modelados como caso de uso pois demandam algum processamento.

Foram criados dois casos de uso para implementar a funcionalidade de registrar a execução de alguns blocos: **<Executar Transação>** e **Gerenciar Rastreamento de Blocos**. O diagrama de casos de uso está sendo mostrado na figura 5.1.

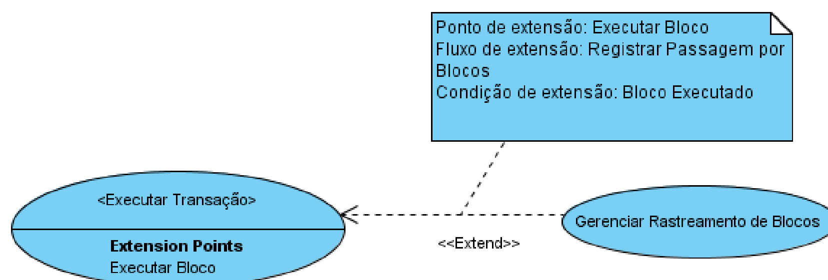


Figura 5.1. Diagrama de casos de uso do aspecto

Pode-se observar nessa figura, que o caso de uso Gerenciar Rastreamento de Blocos estende <Executar Transação> no ponto de extensão Executar Bloco. Quando um bloco for executado será registrada a passagem pelo mesmo. Observa-se também que esse caso de uso não tem nenhum fluxo principal. Isso significa que ele é um caso de uso puramente extensivo e não pode ser instanciado independentemente. Um caso de uso puramente extensivo sempre executará no contexto de um outro caso de uso base.

Avançando um pouco mais e pensando em um nível de projeto, deve-se tomar cuidado para continuar mantendo as áreas de interesse separadas. Com técnicas tradicionais de modelagem, a introdução de casos de uso de extensão em casos de uso base causa emaranhamento de código na fase de implementação. Utilizando-se aspectos, **fatias de caso de uso** podem ser utilizadas para separar o novo comportamento. Uma fatia de caso de uso tem esse nome pois "corta" o modelo de projeto. Uma fatia de caso de uso é composta por uma colaboração, que descreve o comportamento do caso de uso; classes específicas para o caso de uso; e extensões de classes existentes para realização

do caso de uso. A fatia do caso de uso Gerenciar Rastreamento de Blocos pode ser visualizada na figura 5.2.

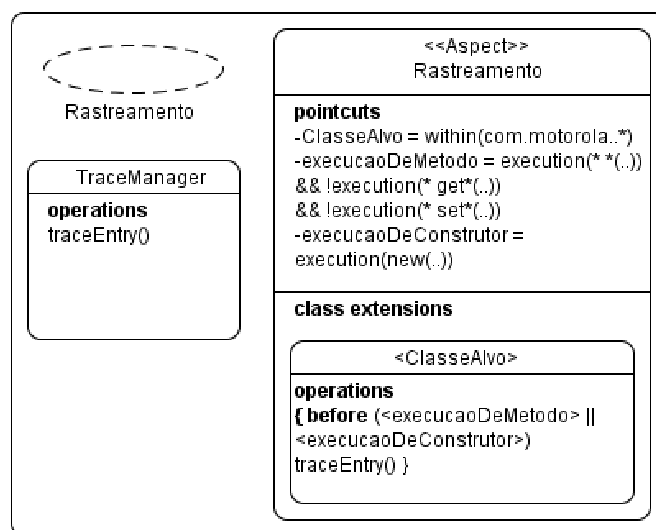


Figura 5.2. Fatia do caso de uso Gerenciar Rastreamento de Blocos

Nessa fatia de caso de uso, o aspecto de Rastreamento contém uma extensão de classe para <ClasseAlvo>. Uma classe alvo executa métodos e construtores. A execução de um construtor da classe é rastreada. A execução de métodos é rastreada, exceto os métodos que começam com *get* ou *set*. Finalmente, o novo comportamento aparece na seção de extensões de classe. Ele define que **antes** (*before*) de uma execução de método ou execução de construtor, será registrada a execução do mesmo com a chamada ao método **traceEntry()** da classe TraceManager.

A implementação do aspecto de rastreamento é derivada da fatia de caso de uso mostrada na figura 5.2. Após o projeto do *plug-in* de controle o aspecto deverá ser modificado para não monitorar alguns blocos.

5.2. Análise de blocos monitorados

Durante a análise e projeto do aspecto, foi feita uma primeira definição de quais blocos seriam monitorados. Esses blocos pertencem aos dois programas que auxiliam a criação e execução de testes automatizados: TAFStudio e TAFPlus2. Após analisar os componentes do TAFPlus2, decidiu-se eliminar o rastreamento de blocos de eventos. Esses blocos são executados na maior parte das execuções, logo, poderiam ter um *spectrum* elevado, gerando diagnósticos incorretos. Além disso, esses blocos dificilmente apresentarão erros pois tem uma complexidade de código baixa. O aspecto do TAFStudio não teve nenhuma modificação nesta fase e apenas será modificado após o projeto do *plug-in*. Após o projeto do *plug-in*, precisa-se atualizar também o aspecto do TAFPlus2.

5.3. Análise, Projeto e Implementação do *plug-in* de controle

Para começar a análise do *plug-in*, foram elaborados alguns artefatos utilizando UML. O primeiro artefato elaborado foi um **diagrama conceitual** para compreender conceitos fundamentais. Esse diagrama será refinado e transformado em um diagrama de classes, à

medida que novos artefatos forem elaborados. O diagrama conceitual pode ser conferido na figura 5.3.

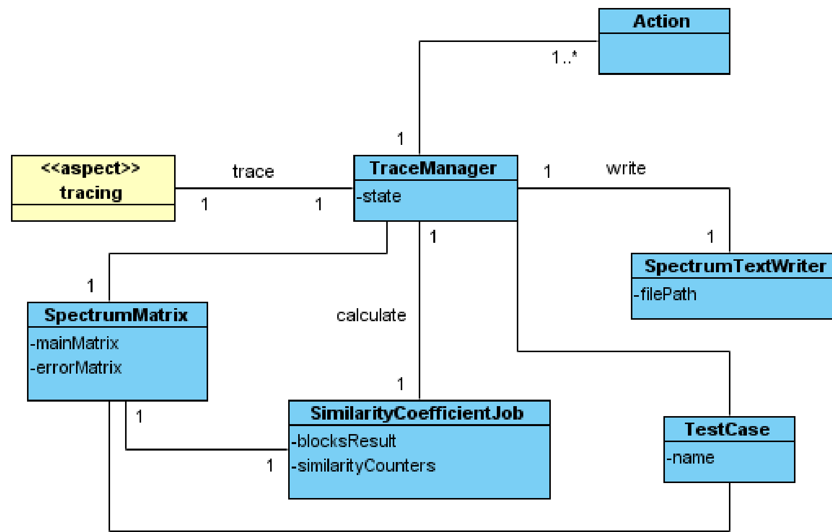


Figura 5.3. Diagrama conceitual do *plug-in*

Nesse diagrama pode-se observar as classes principais que serão necessárias para o *plug-in* de controle e emissão de diagnósticos. A classe **TraceManager** é o componente mais importante, pois ela controla o rastreamento e é a interface de comunicação com os aspectos de rastreamento. Nesse diagrama o aspecto de rastreamento está sendo representando pelo nome **Tracing**. O desenvolvedor pode ativar ações na interface que modificam o estado (executando, pausado, finalizado, etc) do rastreamento. Essas ações serão ativadas por diversas **Actions** que se comunicam com a classe TraceManager. A classe **SpectrumMatrix** é responsável por armazenar a matriz de *spectrum*, contendo os casos de teste e blocos que foram executados. Essa matriz contém também o resultado de cada caso de teste. Com os dados da matriz de *spectrum*, pode-se calcular um diagnóstico através de um coeficiente de similaridade. Essa é a responsabilidade da classe **SimilarityCoefficientJob**. Após calcular o diagnóstico, a classe **SpectrumTextWriter** emitirá o resultado em um arquivo no formato texto. Finalizando, a classe **TestCase** encapsula informações de um caso de teste manual.

Foram elaborados dois casos de uso para o *plug-in*: **Diagnosticar Casos de Teste** e **Controlar Rastreamento**. Para cada caso de uso foi criado um diagrama de atividades e um diagrama de sequência. O diagrama de atividade é um diagrama em um maior nível de abstração. Já o diagrama de atividades dá mais ênfase ao tempo e servirá como guia para implementação. O diagrama de casos de uso do *plug-in* está sendo mostrado na figura 5.4.

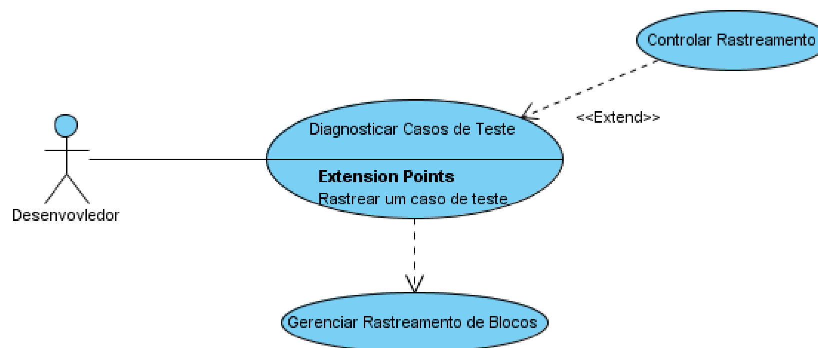


Figura 5.4. Diagrama de casos de uso do *plug-in*

Após refinar os casos de uso do sistema, atualiza-se o diagrama conceitual com os novos conceitos, atributos, relacionamentos e métodos descobertos. Assim, obtém-se o **diagrama de classes**. Segundo (SILVA, 2007),o diagrama de classes deve ser utilizado para representar os elementos de um programa orientado a objetos em tempo de desenvolvimento e não em tempo de execução.

Baseando-se nos artefatos elaborados, inicia-se a programação. A programação será guiada principalmente pelos diagramas de sequência, pois eles demonstram a execução temporal de métodos das classes modeladas. Nesta fase, cria-se o *plug-in* de controle do rastreamento. Para criar esse *plug-in*, utilizou-se a IDE Eclipse (ECLIPSE, 2009a) para desenvolvimento, e o livro Eclipse: Building Commercial-Quality Plug-ins (CLAYBERG; RUBEL, 2004) como principal referência.

5.4. Composição

Com o *plug-in* de controle implementado, precisa-se atualizar o aspecto para não monitorar blocos do *plug-in* relacionados com ações de interface do usuário. Para isso, retira-se o monitoramento de classes de ações de interface do usuário (*actions*) e de janelas (*dialogs*) ativadas por essas ações de interface.

Com os aspectos atualizados, é necessário compor os mesmos com os programas alvo para começar a registrar a execução de blocos. Cada programa alvo é composto por vários projetos de *plug-in*. Logo, deve-se executar a composição com o aspecto para cada projeto que se deseja monitorar.

Após a composição dos aspectos com os programas alvo (introdução da funcionalidade de rastreamento da execução de blocos) e a implementação do *plug-in* de controle, tem-se a ferramenta finalizada. Neste momento já pode-se localizar e diagnosticar faltas do TAFStudio ou TAFPlus2. O próximo capítulo descreve o ciclo de execução de testes manuais, e avalia os resultados obtidos após utilizar a ferramenta para diagnosticar algumas faltas introduzidas propositadamente.

6. Experimentação e Resultados

6.1. Criação de testes

Os testes criados nesta seção baseiam-se na técnica caixa preta. Cada caso de teste pode ter um número limitado de passos e cada passo pode ou não ter um resultado esperado. Ao executar um teste, se o resultado obtido de um passo estiver de acordo com o

resultado esperado do mesmo, então o próximo passo será executado. No momento que o resultado obtido de um passo diferir do esperado, o caso de teste já é declarado como falho. Se todos os resultados obtidos estiverem de acordo com os esperados, então o teste passou.

Neste trabalho, existe uma planilha de casos de teste manuais para cada programa monitorado. O TAFPlus2 já tinha uma planilha com alguns casos de teste que serão aproveitados e modificados. Além desses casos de teste, novos serão criados, pois alguns testes da planilha anterior estão desatualizados. Para o TAFStudio tornou-se necessário criar uma nova planilha de testes. Esses testes cobrem diversas funcionalidades do sistema, desde casos normais de uso até entradas inesperadas pelo usuário.

6.2. Introdução de faltas

Com o intuito de verificar se a ferramenta emite diagnósticos precisos, foram introduzidas faltas em determinados blocos de código. Esses blocos são executados pelas suítes de testes. O objetivo é que cada suíte de testes execute um bloco com falta. Ao se tratar de casos de teste, geralmente se tem um pequeno número de faltas entre duas versões de um programa. Esse comportamento foi observado no ambiente de testes da Motorola. Logo, a idéia de inserir uma falta por suíte de testes é aceitável. Essa idéia deve refletir a realidade quando a ferramenta for executada para localizar e diagnosticar defeitos que não foram inseridos propositavelmente.

No total foram introduzidas oito faltas no TAFStudio e oito faltas no TAFPlus2. Estas faltas foram distribuídas em blocos que têm em funções diferentes.

6.3. Execução de testes

Com as suítes criadas e os defeitos inseridos, parte-se para a execução manual das suítes de casos de teste com a ferramenta habilitada, a fim de diagnosticar possíveis blocos com falta. Para cada falta verifica-se qual suíte de testes é a mais impactada pela mesma. Assim, associa-se a falta a uma suíte de testes e executa-se essa suíte com a falta habilitada. Ao final, calcula-se o *spectrum* dessa suíte utilizando os dois coeficientes (Ochiai e Jaccard). Obtêm-se como resultado (diagnóstico) os blocos com maior probabilidade de estarem com falta. Então, pode-se verificar se o bloco com falta foi reportado pela ferramenta como um dos candidatos a estarem com defeito. Se o bloco tiver sido reportado então a ferramenta está funcionando corretamente. A figura 6.1 demonstra o ciclo de execução manual de suítes de casos de teste.

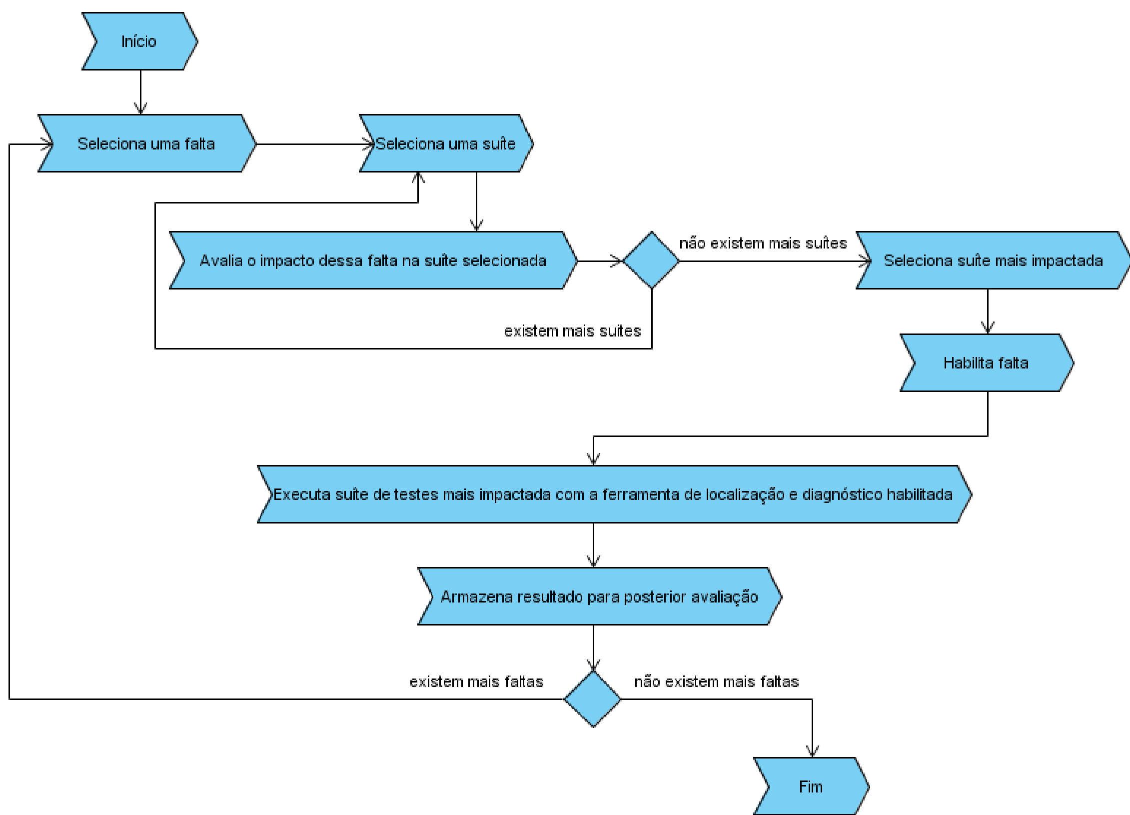


Figura 6.1. Ciclo de execução de testes

6.4. Avaliação

6.4.1. Coeficiente de similaridade

Nas execuções manuais de suítes de testes para os dois programas monitorados, não se percebeu grande influência do coeficiente no resultado. Em todos os experimentos, o número de blocos que necessitam ser inspecionados foi o mesmo para ambos coeficientes. No entanto, notou-se que em algumas execuções outros blocos foram classificados com probabilidades diferentes pelos coeficientes. Se o defeito estivesse em algum desses blocos, poderia existir uma diferença no resultado. Logo, precisa-se saber qual coeficiente tem a maior confiabilidade. Segundo (ABREU; ZOETEWEIJ; GEMUND, 2006), o coeficiente de Ochiai apresenta melhores resultados, pois após uma bateria de testes com diversos coeficientes este coeficiente superou os outros em relação a precisão do diagnóstico em até 10%. Assim, optou-se por utilizar o coeficiente de Ochiai como base para avaliação dos resultados.

6.4.2. Suíte de testes

A suíte de testes pode afetar a precisão do diagnóstico da ferramenta. Essa situação ocorre em ambos os coeficientes. Esse comportamento pode ocorrer quando um bloco com falta não leva a uma falha em determinados testes. Também pode acontecer quando temos mais de uma falta em uma suíte de testes.

No primeiro caso, esse problema poderia ser evitado se todo teste que passasse por um bloco com falta falhasse. Infelizmente, esse tipo de comportamento é impossível

de se obter, pois em qualquer programa sempre existirá situações que não levam a um erro e a falta passará despercebida.

No segundo caso, quando existe mais de uma falta em uma suíte de testes, a ferramenta emite resultados pouco precisos. Isso acontece pois em alguns casos um bloco com falta não é executado em um determinado teste e esse teste falha. Esse comportamento faz com que o *spectrum* desse bloco diminua. Para prevenir esse problema recomenda-se agrupar os casos de teste que executam funcionalidades semelhantes. Essa é uma forma de prevenir que se tenha mais de uma falta por suíte de testes. Além disso, foi observado no ambiente de testes da Motorola, que entre duas versões de um programa poucas faltas são introduzidas. Logo, a suposição de existir apenas uma falta por suíte de testes é válida.

6.4.3. Programa alvo

Outro objetivo deste trabalho é verificar se a ferramenta emite resultados diferentes dependendo do programa monitorado. No TAFPlus2 e no TAFStudio foram introduzidas oito faltas para cada programa, resultando em dezesseis execuções diferentes. Em média, 20% dos blocos necessitam ser inspecionados após a execução da ferramenta no TAFPlus2. No TAFStudio, em média necessita-se inspecionar 15% dos blocos para encontrar o bloco com falta. Conclui-se então que programas diferentes submetidos à ferramenta têm uma precisão de diagnóstico semelhante. A pequena diferença de precisão deve-se principalmente à diferença entre as suítes de testes para os dois programas. Vale ressaltar que o TAFPlus2 contém quatro vezes mais blocos que o TAFStudio.

6.4.4. Granularidade de blocos

A primeira implementação da ferramenta tinha uma granularidade maior, utilizando classes como blocos. Foram executadas algumas suítes de testes para verificar se os resultados eram satisfatórios. Ao final, concluiu-se que a ferramenta não emitia bons resultados, pois ela não apontava com precisão em qual parte do bloco suspeito estava a falta. Como o bloco poderia ser uma classe de mais de 1000 linhas, seria necessário percorrer cada uma dessas linhas até encontrar a falta.

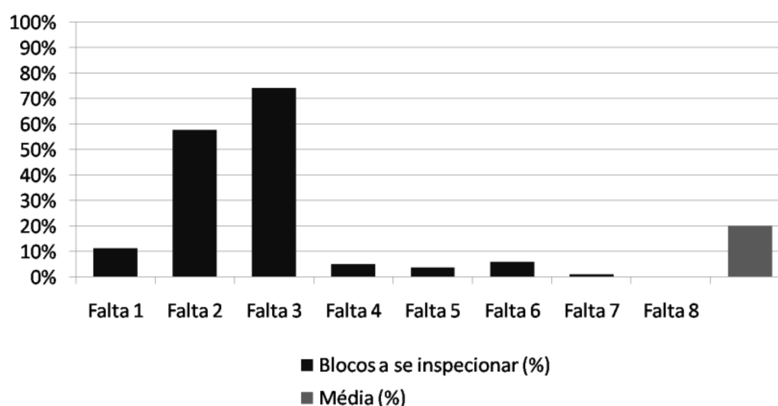
Assim, resolveu-se diminuir a granularidade e obter informações mais detalhadas de uma classe. Decidiu-se por considerar um construtor ou método como um bloco. Essa abordagem trouxe um aumento na precisão e maior rapidez na localização da falta, pois um método ou construtor não tem uma grande quantidade de linhas de código. É importante ressaltar que não é possível modificar a granularidade de blocos em tempo de execução. Essa granularidade deve ser modificada no código dos aspectos antes de executar a composição dos aspectos com os programas alvo.

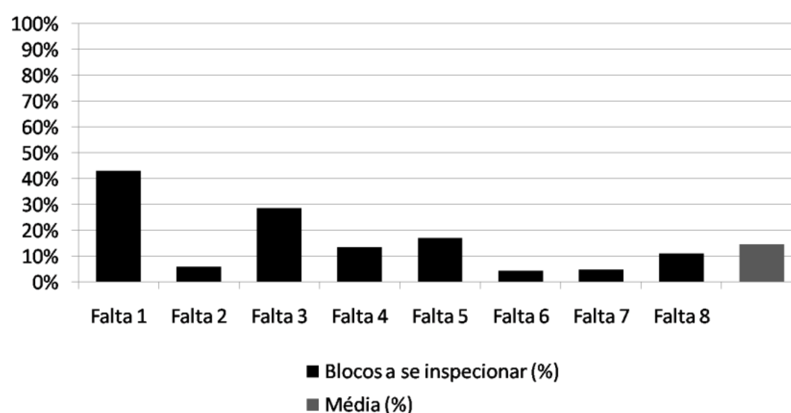
Porém, em alguns casos ainda existem problemas com a granularidade dos blocos. Esses problemas acontecem em trechos de código onde existem diversos blocos de condição (*if/else*). Por exemplo, um bloco que tem três condicionais. Vamos supor que o primeiro condicional está com defeito. Se em cinco execuções apenas uma delas entrar nesse condicional e as outras quatro executarem os outros condicionais, não será lançada uma falha. Assim, para solucionar esse problema cada condicional deveria ser considerado um bloco para a ferramenta. Poder-se-ia considerar um bloco de código

Java (*statement*) como um bloco para a ferramenta. O grande problema dessa abordagem é que existiria um número muito maior de blocos na memória e uma queda de desempenho da ferramenta.

6.5. Resultados

Nesta seção serão mostrados os resultados do ciclo de execução de testes para as 16 faltas introduzidas nos programas monitorados: TAFStudio e TAFPlus2. Cada suíte de testes foi executada com uma falta e a ferramenta habilitada. Ao final, obtém-se o diagnóstico que mostra quais blocos tem a maior probabilidade de estar com falta. A eficiência de um diagnóstico é avaliada em relação à porcentagem de blocos que necessitam ser inspecionados manualmente até se encontrar o bloco defeituoso. A porcentagem de blocos que necessitam ser inspecionados está sendo mostrada no eixo y do gráfico de resultados. O eixo x representa as faltas introduzidas. Os resultados podem ser visualizados nos gráficos abaixo. O primeiro refere-se aos resultados do TAFPlus2. O segundo trata dos resultados do TAFStudio.





Os resultados podem ser avaliados como satisfatórios, pois em média necessita-se inspecionar uma quantidade pequenas de blocos até se encontrar o bloco defeituoso. As faltas 2 e 3 do TAFPlus2 tiveram resultados considerados ruins, pois necessita-se inspecionar mais de 55% dos blocos até se encontrar o defeito. As outras faltas do TAFPlus2 tiveram resultados muito bons. Já os resultados do TAFStudio têm uma precisão semelhante entre eles. A falta 1 tem o pior resultado onde mais de 40% dos

blocos necessitam ser inspecionados. A falta 3 teve um resultado razoável, pois quase 30% dos blocos necessitam ser inspecionados manualmente.

7. Conclusão

A ferramenta de localização e diagnóstico de faltas diminui consideravelmente o tempo gasto para se encontrar um bloco com defeito em um programa, pois o diagnóstico emitido pela ferramenta diminui a quantidade de blocos que precisam ser inspecionados manualmente até se encontrar o bloco com falta. Assim, no desenvolvimento de um novo programa, o tempo de entrega ao cliente será menor, possibilitando um menor ciclo de desenvolvimento. Na manutenção de um programa, a execução de suítes de testes com a ferramenta ajudará a localizar erros e diminuirá o esforço do testador até encontrar os blocos com faltas. Partes do programa com defeito serão diagnosticadas e corrigidas, resultando em um programa com maior confiabilidade.

Concluiu-se também que quanto menor a granularidade dos blocos monitorados, maior a eficiência da técnica de diagnóstico. Neste trabalho, foi utilizada uma granularidade média, onde um bloco é um método ou construtor. Assim, existiu um balanceamento entre precisão do diagnóstico e tempo de processamento, pois quanto menor a granularidade, maior o processamento necessário.

O programa monitorado e o coeficiente de similaridade são fatores que quase não influenciam o resultado do diagnóstico. Os dois coeficientes de similaridade geraram resultados muito parecidos. Em relação aos programas monitorados, a precisão média da técnica também é semelhante. A diferença foi de apenas 5 pontos percentuais a mais de blocos que precisam ser inspecionados no TAFPlus2.

Pode-se destacar também que a suíte de testes afeta a precisão de diagnóstico da técnica. Os testes devem executar funções semelhantes, para que se previna a possibilidade de existir muitas faltas em uma mesma suíte. Além disso, o grande problema da técnica de *spectrum* é que alguns erros podem passar despercebidos, sem causar uma falha. Esse comportamento é difícil de ser evitado, pois isso pode ser uma característica do programa.

Em geral, a ferramenta emite resultados satisfatórios e que auxiliam o testador a encontrar o bloco com falta em um menor tempo. Os resultados ruins acontecem quanto muitos testes executam blocos com falta que não levam a falhas. Mas esse tipo de problema acontece também quando um testador utiliza o diagnóstico manual e, portanto, é inerente a técnica utilizada.

7.1. Trabalhos futuros

Como trabalhos futuros, pode-se diminuir a granularidade dos blocos para quantificar o ganho de precisão obtido. Pode-se também investigar a possibilidade de automatizar os testes manuais dos programas alvo. Assim, o testador apenas elaboraria os testes em código Java, habilitaria a ferramenta e receberia os resultados quando uma suíte de testes terminasse de executar. Outro aspecto importante que poderia ser investigado é a representatividade dos casos de teste. Também seria interessante elaborar alguma maneira de detectar faltas que passam despercebidas em determinadas execuções. Essa última proposta é um pouco mais complicada, devido à complexidade de se detectar esse tipo de falta.

Referências

Abreu, R.; Zoeteweij, P.; Gemund, A. J. V. An evaluation of similarity coefficients for software fault localization. In: 12TH PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING. 12th Pacific Rim International Symposium on DependableComputing. [S.l.], 2006.

Abreu, R.; Zoeteweij, P.; Gemund, A. J. V. On the accuracy of spectrum-based fault localization. IEEE Computer, 2007.

Avizienis, A. et al. Basic concepts and taxonomy of dependable and secure computing. In: Dependable and Secure Computing. [S.l.]: IEEE, 2004. p. 11–33.

Clayberg, E.; Rubel, D. Eclipse: Building Commercial-Quality Plug-ins. [S.l.]: Addison-Wesley Professional, 2004. (Eclipse).

Colyer, A. et al. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. [S.l.]: Addison-Wesley Professional, 2004. (Eclipse).

Dallmeier, V.; Lindig, C.; Zeller, A. Lightweight defect localization for java. In: 19TH EUROPEAN CONFERENCE GLASGOW (ECOOP). 19th European Conference Glasgow (ECOOP). [S.l.], 2005. p. 528–550.

Eclipse. Eclipse Corporation. 2009. Acesso em: Fev. 2009. Disponível em: <http://www.eclipse.org>.

Eclipse. IAJC AspectJ Task. 2009. Acesso em: Mar. 2009. Disponível em: <http://www.eclipse.org/aspectj/doc/released/devguide/antTasks-iajc.html>.

Hailpern, B.; Santhanam, P. Software debugging, testing and verification. In: IBM Research. IBM Journal, 2002. p. 4–12. Disponível em: <http://www.research.ibm.com/journal/sj/411/hailpern.html>.

Jacobson, I.; Ng, P.-W. Aspect-Oriented Software Development with Use Cases. [S.l.]: Addison-Wesley, 2005.

Jain, A. K.; Dubes, R. C. Algorithms for Clustering Data. [S.l.]: Prentice-Hall, 1988.

Jones, J. A.; Harrold, M. J.; Stasko, J. Visualization of test information to assist fault localization. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING. International Conference on Software Engineering. [S.l.], 2002. p. 467–477.

Kawakami, L. et al. A test automation framework for mobile phones. In: 8TH IEEE LATW 2007. Proceedings of 8th IEEE LATW 2007. [S.l.], 2007.

Kickzales, G. et al. Aspect-oriented programming. In: Aspect-Oriented Programming. [S.l.]: European Conference on Object-Oriented Programming, 1997.

Meyer, A. da S. et al. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). Genetics and Molecular Biology, v. 27, n. 1, p. 83–91, 2004.

Myers, G. J. The Art of Software Testing. [S.l.]: John Wiley & Sons, 2004.

Reps, T. et al. The use of program profiling for software maintenance with applications to the year 2000 problem. In: 6TH EUROPEAN SOFTWARE ENGINEERING CONFERENCE HELD JOINTLY WITH THE 5TH ACM SIGSOFT INTERNATIONAL SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING. Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering. [S.l.], 1997. P. 432–449.

Silva, R. P. e. UML 2 em Modelagem Orientada a Objetos. Florian´opolis: Visual Books, 2007.

Xerox. The AspectJ Programming Guide. 1998. Acesso em: Set. 2008. Disponível em: <http://www.eclipse.org/aspectj/doc/released/progguide/index.html>.