Universidade Federal de Pernambuco

Centro de Informática

Monitoramento de Aplicações com Orientação a Aspectos

Trabalho de Graduação

Por

João Paulo Sabino de Moraes

Recife, Junho de 2009

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

João Paulo Sabino de Moraes

Monitoramento de Aplicações com Orientação a Aspectos

Este trabalho foi apresentado ao Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Paulo Henrique Monteiro Borba

"*One sweet dream*

came true, today."

Paul McCartney

À minha mãe Selme Maria,

Agradecimentos

Engraçado pensar como são tantas as pessoas extraordinárias que a gente conhece e, num piscar de olhos, “desconhece” ao longo da vida. Hoje, algumas delas podem nem mesmo estar lembradas da gente, outras nos vêem todo santo dia, algumas vêem a gente e fazem de conta que não vêem, às vezes as vemos e fazemos o mesmo e quando estamos sozinhos olhando a rua vazia pela janela, nos pegamos tendo dúvidas de saber como elas estão, como deixaram de ser e como serão, o que estão fazendo e o que pensam em ser. No entanto, em meio a todos esses universos de encontros e desencontros, lembranças, esquecimentos e conseqüentemente evolução, é fácil e ao mesmo tempo bastante sutil notar como cada momento passado por cada pessoa de valor que conhecemos na vida contribuiu para sermos o que somos.

Impossível não agradecer primeiramente à minha mãe como grande responsável pelo que sou hoje, detentora de uma sabedoria inata, dessas de desbancar gente graduada, ela me ensinou tudo que sei para seguir a diante na vida com minhas próprias pernas. À meu pai que com seu grande talento e criatividade me fez ver que às vezes podemos enxergar a vida através de rabiscos de lápis de cor e pipas avisadas. Ao meu irmão que me ensinou que tudo na vida deve ser dividido para uma melhor convivência. Por todos esses ensinamentos que eles nem sabem que me deram, agradeço à minha família. À minha prima Maria Helena que sempre nos ajudou com palavras de conforto e de esperança nos momentos difíceis. Ao meu primo Antônio Manuel, que por muitos é conhecido como Fidel, mas que desde criança aprendi a chamar de Toninho, ensinou me que nem tudo é o que parece ser e que através da razão podemos chegar a respostas impossíveis de serem encontradas nos melhores livros.

Ao pessoal da Ávila que me ajudou bastante compreendendo minhas faltas e saídas repentinas para fazer o TG. Ao meu orientador Paulo Borba que, sempre paciente, me ensinou a pensar objetivamente na defesa de uma idéia. À Felype Santiago e Leopoldo Teixeira que me ajudaram bastante no desenvolvimento deste trabalho. E ao pessoal da lista *Aspect J Users* sempre com respostas rápidas às dúvidas reportadas.

E claro como não agradecer aos meus inadjetiváveis amigos pós entrada de faculdade. Agradeço a todos eles, Caio, Apebão, Mário, Lhama, Xumiga, SoSo, Leila, Tiago, Riffa e João Doido. Impressionante como todos nós somos absurdamente parecidos em algumas coisas, apesar de sermos totalmente diferentes em outras. Como se diz por aí, tava escrito que eu tinha que fazer amizade com essa galera =) Mas faço um agradecimento especial, claro, aos meus quase irmãos Lupicínio, Apebão (Bruno Marques) e Caio César (Peter). Muitos bons momentos vivi com essa galera!

Todas essas pessoas nem imaginam o quanto são importantes na minha vida, mas aqui vão meus sinceros e silenciosos agradecimentos por elas serem quem realmente são.

Resumo

O Monitoramento de aplicações é um requisito cada vez mais comum em sistemas hoje em dia. No entanto, ao adicionarmos características de monitoramento diretamente nas partes da aplicação a serem monitoradas, problemas de espalhamento ou entrelaçamento de código poderão facilmente aparecer, causando grande perda de modularidade no desenvolvimento do sistema como um todo. O uso de Programação Orientada a Aspectos é uma ótima alternativa a esse problema, visto que não há necessidade de o código de monitoramento estar próximo ao código da aplicação. Porém o uso de aspectos em aplicações orientadas a *plug-ins* em Java que garanta a modularidade de fato, torna-se mais viável quando seu desenvolvimento se dá sobre plataforma *Equinox Aspects*. Este trabalho descreve, portanto, a implementação de um *plug-in* de monitoramento baseado em aspectos no *framework Equinox Aspects* avaliando os resultados obtidos através de comparações a nível de código com abordagem em OO.

Palavras-Chave: Monitoramento, TaRGeT, Programação Orientada a Aspectos e Modularidade

Abstract

The application monitoring is an increasingly common requirement in nowadays systems. However, adding monitoring features directly to parts of applications to be monitored, problems related to scattering or tangling of concerns may easily appear, causing great loss in modularity on system development as a whole. The use of Aspect Oriented Programming is a great alternative to this problem, since there isn't necessity to the monitoring code to be close to the application code. However the use of aspects in Java plug-ins oriented applications which guarantee modularity, in fact, becomes more viable when the development occurs on Equinox Aspects platform. So this work describes an implementation of an aspect based monitoring plug-in on Equinox Aspects Framework and measures its results by comparing at source code level with OO approach.

Key-Words: Monitoring, TaRGeT, Aspect Oriented Programming, Modularity

Sumário

Índice de Figuras

Índice de Listagens

Índice de Tabelas

1. Introdução

O monitoramento de aplicações para avaliar o uso da aplicação e detectar problemas, antes que o usuário final os perceba, é um requisito comum em sistemas [1]. Uma maneira bastante simples e eficaz de se realizar monitoramento em um dado sistema é através do uso de *loggings* [1], que consiste no registro de certos passos executados pela aplicação. O modo como será feito esses registros dependerá das estratégias de monitoramento a serem utilizadas na aplicação a partir da identificação de componentes a serem monitorados.

A implementação de sistemas a serem monitorados via *loggings* pode se dar paralelamente ao desenvolvimento de módulos ou pequenos sub-fluxos monitores que façam parte do sistema como um todo. Neste caso, seria necessária a identificação de funcionalidades da aplicação a serem monitoradas e posterior escolha de abordagens de implementação do monitoramento. No entanto, o desenvolvimento de sistemas com esta característica pode acarretar alguns problemas dependendo da abordagem escolhida para sua implementação.

Adicionar características de monitoramento é algo bastante factível na maioria dos sistemas como já mencionado, porém, a partir do momento em que códigos de interesses particulares são misturados ao código do sistema, a aplicação como um todo passa a ter grandes riscos de perda de modularidade. A falta de modularidade em sistemas estruturais ou orientados a objeto (OO) pode ser diagnosticada quando são percebidas características de comportamento transversal (*crosscutting concerns)* que normalmente ficam espalhadas (*scattered)* e entrelaçadas (*tangled)* no código-fonte de tais sistemas, diminuindo a possibilidade de reuso e qualidade do mesmo [2].

Buscando obter modularização no desenvolvimento de sistemas com um módulo monitor, uma alternativa inteligente e eficaz é o uso de Aspectos. A Programação Orientada a Aspectos (POA) cresceu bastante nos últimos tempos devido ao reconhecimento de que boa parte das aplicações não possui módulos independentes ou vários módulos altamente relacionados, dando margem ao aparecimento dos interesses de comportamento transversais (*crosscutting concerns)* [2]. No desenvolvimento de sistemas baseado em POA é possível eliminar este comportamento transversal resgatando as possibilidades de reuso e modularização do sistema transportando os *crosscutting concerns* para os Aspectos.

Há algum tempo, foi identificada a necessidade de desenvolver um módulo monitor para a ferramenta TaRGeT com o objetivo de gerenciar a maneira como as funcionalidades e componentes da TaRGeT estavam sendo utilizados. A TaRGeT é uma aplicação da Motorola utilizada para geração de *Test Cases* voltados a aparelhos móveis, foi implementada sobre a plataforma *Eclipse RCP* [3] e é formada por diversos *plug-ins* que compõem os diversos módulos desta aplicação. O componente de monitoramento seria mais um *plug-in* a ser

adicionado e, de acordo com os possíveis cenários de desenvolvimento voltados a aplicações com módulo monitor, a Programação Orientada a Aspectos seria uma solução satisfatória neste caso. Contudo, o uso de aspectos em aplicações *Eclipse RCP* não necessariamente garante a modularização do código, pois, caso o aspecto criado tenha que interceptar mais de um *plug-in,* ele terá que ser replicado em todos os *plug-ins* que precisarão ter partes do código interceptadas *(weaving)* pelos aspectos, comprometendo significativamente o trabalho de reuso e modularidade da aplicação.

Objetivando modularização no desenvolvimento de aspectos em aplicações compostas por diversos *plug-ins*, há alguns meses atrás foi desenvolvido por membros da *Eclipse Foundation* [14] um componente do *Equinox Framework* denominado *Equinox Aspects* [4]. A perda de modularidade em aplicações RCP é factível porque, na maioria dos casos, os aspectos devem ser replicados nos *plug-ins* a serem interceptados por eles. Este problema ocorre porque o AJDT (*Aspect J Development Tool)* sozinho não oferece suporte à interceptação (*weaving)* de códigos contidos em *plug-ins* JDT (Java *Development Tools)* [13], sendo necessária a inclusão do código em aspectos diretamente no *plug-in* a ser interceptado. No caso de aplicações simples com apenas um *plug-in* pode ser que não acarrete tantos problemas, porém caso a aplicação seja composta por diversos *plug-ins* que devam ter seus códigos interceptados por aspectos, a replicação dos aspectos nos *plug-ins* é inevitável. No entanto, *Equinox Aspects* provê suporte ao AJDT para a realização de *weaving* em JDT *plug-ins* fazendo com que os aspectos, que podem estar contidos em apenas um *plug-in,* interceptem códigos contidos em qualquer *plug-in* JDT contido no *workspace* da aplicação.

1.1 Objetivos

O objetivo deste trabalho é, portanto, apresentar o desenvolvimento de um componente de monitoramento que gera informações de *logging* no uso dos principais componentes da ferramenta TaRGeT garantindo as características de modularidade e reuso da ferramenta. Ao longo do trabalho serão apresentados problemas relacionados a abordagens em OO que causam perda de modularidade e como soluções em aspectos, notadamente *Equinox Aspects,* anulam tais problemas.

As principais contribuições deste trabalho são:

- Implementação de um *plug-in* em aspectos para monitoramento de funcionalidades da ferramenta TaRGeT.
- Avaliação da solução em aspectos em relação à abordagem em OO
- Registro dos dados resultantes do monitoramento no Banco de Dados *SQL Server* 2008.

- Implementação de ferramenta no *Visual Studio* 2008 para visualização dos dados armazenados.

1.2 Estrutura do Trabalho

Este trabalho é composto por 8 capítulos. No capítulo 2 será abordada a ferramenta TaRGeT onde serão demonstradas características de suas funcionalidades e desenvolvimento.

O capítulo 3 levanta as estratégias de monitoramento identificadas na ferramenta TaRGeT detalhando quais funcionalidades serão monitoradas e como se dará o monitoramento.

No capítulo 4 será descrita uma abordagem de desenvolvimento de um componente de monitoramento em Orientação a Objetos. Serão também vistos os códigos de monitoramento implementados nas diversas classes a serem monitoradas.

O capítulo 5 apresenta a implementação do *plug-in* de monitoramento em aspectos, detalhando características de sua implentação em *Equinox Aspects*, exemplos de códigos em aspectos e como eles interceptam os códigos a serem monitorados. No capítulo 6 serão avaliadas a abordagem em Orientação a Objetos e a implementação em aspectos, prós e contras e o capítulo 7 discorre sobre as análises de como os dados do monitoramento foram aproveitados.

Por fim, no capítulo 8 constam sucintas considerações finais e sugestões para trabalhos futuros.

2. A Ferramenta TaRGeT

A TaRGeT é uma aplicação da motorola desenvolvida para a geração de *test cases* de aplicações voltadas a aparelhos móveis. O processo de geração de *test cases* inicia-se com o desenvolvimento dos *use cases* em um *template* específico para as necessidades da ferramenta. Os *use cases* são compostos de fluxos principais (*main flows)* e fluxos secundários *(alternative flows)* ou de exceções (*exception flows)* definidos em documentos de requisitos a parte (*requirement codes).* Em cada fluxo no documento de *use cases,* são detalhados seus passos e um identificador do requisito ao qual ele pertence. Depois de escritos, os documentos dos *use cases (use case documents)* são importados na TaRGeT onde são chamados de *features*, sendo assim as *features* contém os *use cases* a serem utilizados pela TaRGeT. A seguir veremos um exemplo de um documento de *use case* e o *test case* gerado a partir dos seus fluxos.

Feature 11111 – My Phonebook

UC 01 – Creating a New Contact

Description

This use case describes the creation of a new contact.

Main Flow

Description: Create a new contact
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	Start My Phonebook application.	My Phonebook application is installed in the phone.	My Phonebook application menu is displayed.
2M	Select the New Contact option.		The New Contact form is displayed.
3M	Type the contact name and the phone number.		The new contact form is filled.
4M	Confirm the contact creation. [TRS_11111_101]	There is enough phone memory to insert a new contact.	A new contact is created in My Phonebook application.

Exception Flows

Description: There is no enough memory.
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	Confirm the contact creation.	There is no enough phone memory.	A dialog is displayed informing that there is no enough memory. [TRS_111166_103]

Figura 1 – Exemplo de use case para uso na TaRGeT [8]

A Figura 1 mostra um documento de casos de uso com um fluxo principal e outro de exceção. O cabeçalho é formado pelo nome da *feature* e do caso de uso; abaixo do cabeçalho é

feita uma descrição do *use case* e posteriormente seguem os passos dos fluxos. Abaixo da descrição e identificação do passo inicial e final de cada fluxo há uma tabela que detalha os passos de cada fluxo com informações do id do fluxo, ação do usuário, estado do sistema, retorno dado pelo sistema depois da execução da ação e identificação do requisito ao qual cada fluxo pertence (TRS_111166_103 e TRS_11111_101). Através dos identificadores de cada passo e casos de uso utilizados, a TaRGeT monta os fluxos a serem seguidos nos *test cases.* A Figura 2 logo abaixo, ilustra um *test case* gerado a partir do documento ilustrado na Figura 1 com a condição de que há espaço livre na memória, por conta disso o fluxo da exceção não pertence ao fluxo deste *test case.*

Case	Reg. Leve	Exe. Type	Case Description	Procedure	Expected Results
1	na	Man	11111_R4_1 : It describes the creation of a new contact in the contact list.	OBJECTIVE: Create a new contact	
			Use Cases:	11111#UC_01	
			Requirements:	TRS_11111_101	
			Setup:	None	
			Initial Conditions:	1) My Phonebook application is installed in the phone. 2) There is enough phone memory to insert a new	
			Notes:	Test case auto-generated by TaRGeT system.	
			Test Procedure (Step Number):		
			1	Start My Phonebook application.	My Phonebook application menu is displayed.
			2	Select the New Contact option.	The New Contact form is displayed.
			3	Type the contact name and the phone number.	The new contact form is filled.
			4	Confirm the contact creation. [TRS_11111_101]	A new contact is created in My Phonebook application.
			Final Conditions:	None	
			Cleanup:	None	

Figura 2 – Exemplo de test case gerado pela TaRGeT [8]

O número de *test cases* possíveis de serem gerados dependem diretamente do número de casos de uso e requisitos a serem contemplados. Para controlar este processo e dar ao usuário a possibilidade de escolha de como os *test cases* serão gerados, a TaRGeT possui um ambiente propício a seleção dos *test cases* com uma grande variedade de filtros e componentes que tornam o processo de geração bastante eficiente e fácil de ser manipulado.

2.1 Objetivos e principais funcionalidades da TaRGeT

A TaRGeT tem como objetivo prover um ambiente que permita ao seu usuário a geração de uma *test suite* composta por um conjunto de vários *test cases* tendo como entrada um ou mais documentos de casos de uso. Os *test cases* descrevem os passos a serem seguidos pelos testadores para determinados fluxos de execução da ferramenta contidos nos casos de usos referentes às funcionalidades da aplicação. Na TaRGeT, a geração dos *test cases* cobre todos os cenários de testes possíveis encontrados em um ou mais documentos de entrada. Esta possibilidade de ter mais de um documento como entrada facilita a geração de *test cases* que possuam fluxos em mais de um documento. Para entender melhor todo este processo logo abaixo serão mostrados os passos necessários para a geração de um *test case* na TaRGeT.

2.1.1 Criando um TaRGeT Project

Primeiramente deve ser criado um projeto para iniciar o uso das funcionalidades da ferramenta. Similarmente ao *Eclipse,* a TaRGeT possui um gerenciador de projetos com possibilidades de criação, abertura e fechamento dos mesmos. Um novo projeto pode ser criado na opção *New Project* no menu *File.* Há também a tela para abertura de projetos denominada *Open Project Wizard* utilizada para abertura de projetos.

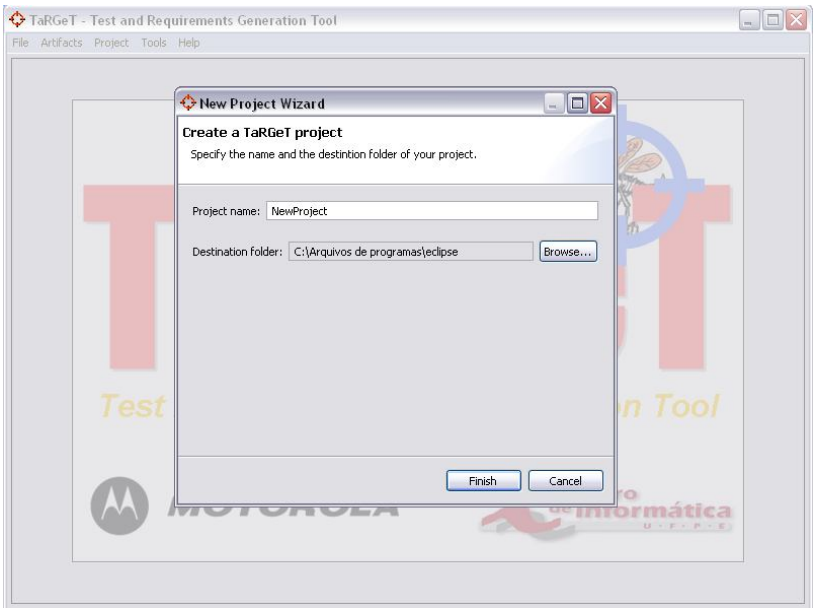


Figura 3 – Tela de criação de novo projeto

2.1.2 Importação de Documentos

Após a criação do projeto, os documentos de casos de uso podem ser importados na tela *Import Documents Wizard* localizada no menu *Artifcats*, opção *Import Use Case Documents*, como detalhado na Figura 3 logo abaixo.

Figura 4 – Tela de importação de documentos de use cases

2.1.3 Geração on the fly de test cases

A partir da importação dos documentos e por conseqüência dos *use cases*, é possível iniciar o processo de geração dos *test cases.* A geração é feita na tela *On The Fly Generation Editor* localizada no menu *tools.* A tela possui três abas. Na aba *selections* (Figura 5) podem ser escolhidos os requisitos, casos de uso, passos dos fluxos dos *use cases* e o filtro *path coverage* que possibilita a escolha de um percentual que diz respeito à quantidade de *test cases a* serem gerados dentre os contemplados pelo filtro. A característica *on the fly* permite que após a configuração dos filtros na aba *selection,* a *suite* de testes já esteja preparada na aba *Test*

Cases, aumentando bastante a eficiência na geração da *suite* de testes e o tempo de verificação entre o que foi escolhido e o que foi de fato gerado na *suíte* de testes.

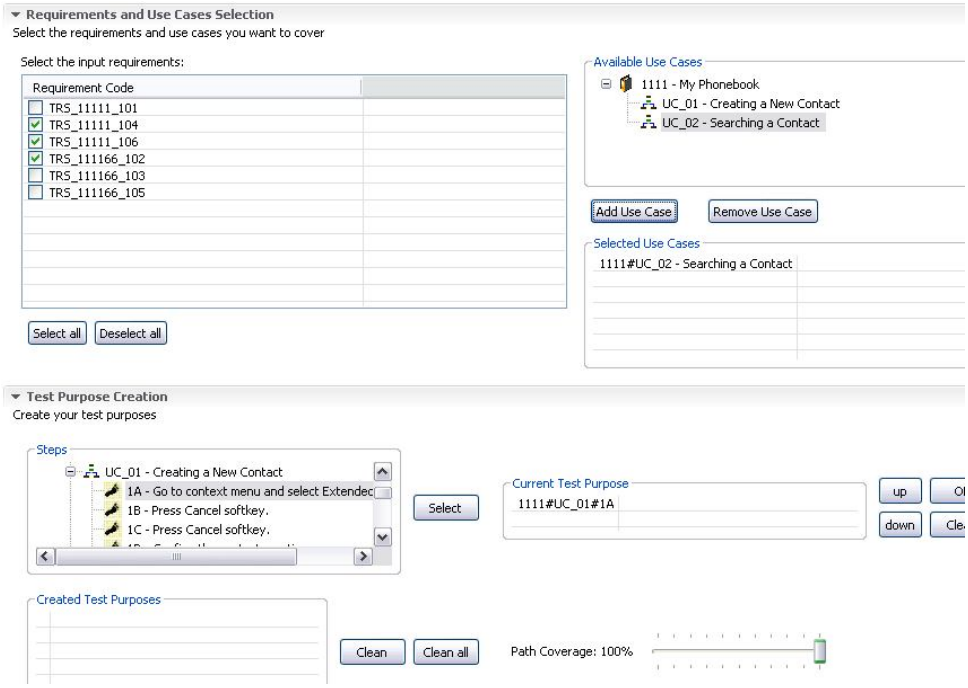


Figura 5 – Tela da aba *selections* e seus filtros para geração de *test cases*

Não é necessária a utilização de todos os filtros para a preparação da *suite* de testes, dependendo do propósito para cada projeto criado ou casos de uso envolvidos, pode ser que seja mais útil usar apenas um filtro em detrimento dos outros. Depois de selecionados os filtros, na aba *Test Cases,* mais especificamente no *grid* intitulado *Generated Test Cases* (Figura 6) é imediatamente criada a *suite* de testes, caso um *test case* seja selecionado, seu detalhamento é mostrado já modelado como *test case* no *grid selected test case* (Figura 7).

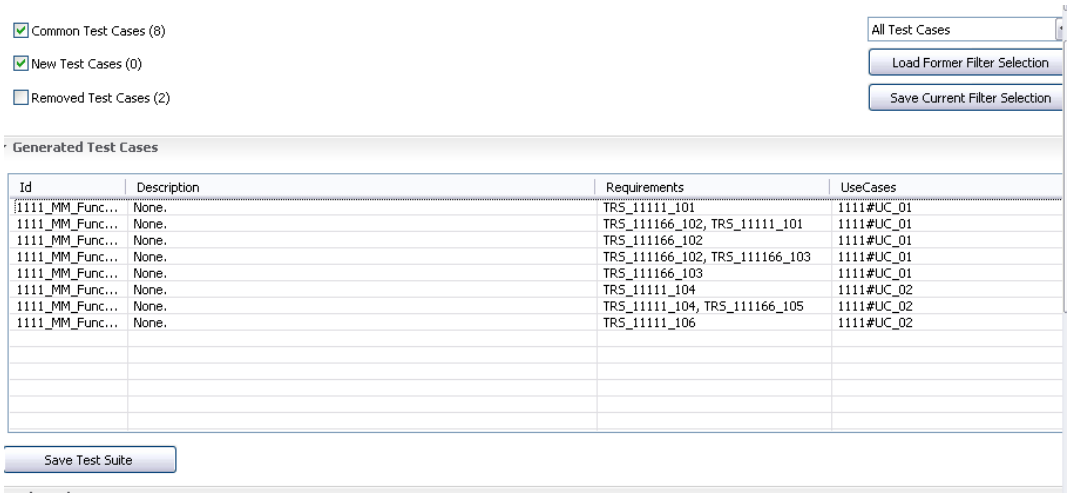


Figura 6 – Visão da *test suite* gerada

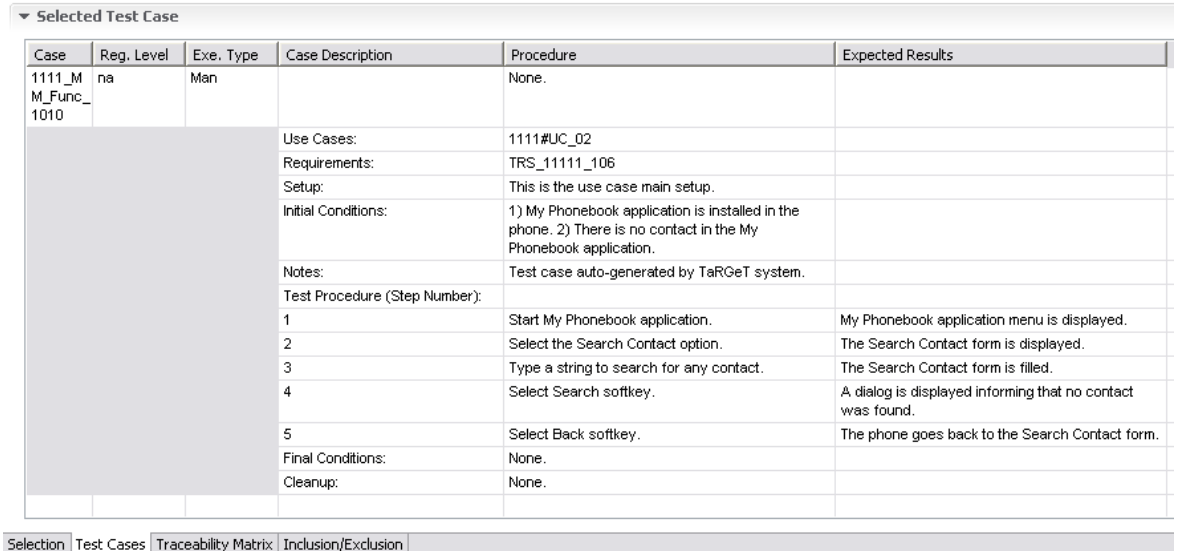


Figura 7 – Visão do *test case* selecionado na *suite* de testes

Como se pode observar na Figura 6, há ainda os filtros *Common, New* e *Removed Test Cases* que através de comparação com o número total de *test cases,* calcula quantos *test cases* são novos, foram removidos ou são comuns em relação ao total. Os botões *Save Filter* e *Load Filter* salvam e carregam respectivamente configurações de filtragem e conseqüentemente *test suites.* O botão *Save Test Suite,* gera um arquivo *excel* com a *suite* de testes utilizada discriminando também informações sobre os casos de uso e requisitos utilizados na *suite.*

Nos tópicos abaixo serão vistas informações a nível de desenvolvimento da TaRGeT, detalhando seu ambiente de desenvolvimento e alguns artefatos necessários para o seu funcionamento.

2.2 Aplicações RCP

Desenvolvido sobre a plataforma *Eclipse RCP (Rich Client Plataform),* a TaRGeT é formada por um conjunto de diversos *plug-ins* que compõem os módulos da aplicação. Cada *plug-in* é responsável por um conjunto de funcionalidades relacionadas a certo interesse da aplicação, podendo então ser caracterizados como módulos da aplicação. Todos esses *plug-ins* determinam inúmeras funcionalidades da TaRGeT como componentes de filtros para seleção de *test cases* e *use cases*, importação de arquivos, gerenciadores de criação de projeto, geração de *test cases* etc.

O desenvolvimento em aplicações sobre a plataforma *RCP* permite aos desenvolvedores o uso da arquitetura do Eclipse para implementação de aplicações flexíveis e *stand-alone*, com a possibilidade de reuso das funcionalidades herdadas do *Eclipse* [3]. Estas funcionalidades são os *plug-ins*, considerados menor parte do Eclipse possível de ser instalada ou implantada como software [3].

Os *plug-ins* são compostos por unidades menores, as extensões (*extensions*) e os pontos de extensão (*extensions-points*) [7]. Os *extension-points* definem funcionalidades que podem ser utilizadas em *plug-ins*, enquanto as *extensions* utilizam *extensions-points* para de fato implementarem novas funcionalides aos *plug-ins*. Dessa forma, o desenvolvimento em *plug-ins* torna-se bastante flexível e extensível trazendo possibilidades de reuso para a aplicação.

2.3 Principais *plug-ins* da Target

O fato de a TaRGeT ter sido desenvolvida sobre a plataforma *Eclipse RCP,* permitiu que seu desenvolvimento fosse totalmente orientado a *plug-ins.* Dessa forma, novas funcionalidades podem ser “plugadas” ou facilmente “desplugadas” o que evidencia a característica *plug and play* de tais aplicações. A TaRGeT atualmente é composta por cerca de 16 *plug-ins,* no entanto, neste tópico, serão apresentados apenas os *plug-ins* que tiverem alguns de seus componentes monitorados, o que será visto na seção 3.

2.3.1 TaRGeT *Project Manager* (TPM)

Plug-in responsável pelo gerenciamento dos processos de criação, abertura e fechamento de projetos. Como são utilizados *extension-points* do próprio *Eclipse*, esta funcionalidade é bastante semelhante ao processo de gerência de projetos no *Eclipse.* Nele estão contidos todas as telas (*wizards),* eventos (*actions*) e códigos de controle e estruturas de armazenamento das propriedades do projeto como nome, localização e usuário do projeto.

2.3.2 TaRGeT TC *Generation Gui* (TCG)

O TCG é o *plug-in* responsável pelo gerenciamento das funcionalidades da tela *On the Fly Editor,* onde ocorrem os processos de seleção, e início da geração de *test suite* e documento excel com todos os *test cases* contidos na *test suite*.

2.3.3 TaRGeT *Test Center* 3 *OutPut Plug-in*

Este *plug-in* é reponsável pela geração da *suite* de t*est cases* no *excel.* Esta funcionalidade foi implementada na classe *ExcelFileFormatter* cujas propriedades possuem muitas informações sobre os *test cases* gerados, fato que definiu a escolha de tal classe.

2.3.4 TaRGeT *MSWord Input*

Plug-in responsável pelo processamento dos documentos de *use cases* contidos em arquivos *word* (.doc). Esse processamento consiste na conversão dos dados contidos no documento *word* em xml através de uma aplicação .NET, o que facilita a transformação final do documento em um Java *object.*

Vistas as principais funcionalidades da TaRGeT e alguns detalhes do seu desenvolvimento, no próximo capítulo, a TaRGeT será abordada num contexto de monitoramento, serão explicados os motivos e como os benefícios trazidos pelo monitoramento a enriquecerão tanto em questões de uso quanto em desempenho. Serão também demonstradas as estratégias de monitoramento utilizadas nos *plug-ins* supracitados e como os dados serão aproveitados para posterior análise.

3. Monitoramento de funcionalidades da TaRGeT

Há algum tempo foi identificada a necessidade de monitoramento para que fosse analisado o desempenho da TaRGeT e o modo como os usuários estavam utilizando suas principais funcionalidades. As funcionalidades identificadas para o monitoramento vão desde o registro de criação e abertura de projetos, até interceptação das informações dos filtros utilizados, registro de exceções causadas por erros da aplicação ou seu uso e dos casos de uso contidos nos fluxos de cada *test case.*

Uma abordagem de monitoramento favorável à captação de informações que ofereçam suporte à análise da ferramenta é através do uso de *logging que* é o processo de registro de informações de passos executados, resultados ou mensagens levantadas pela aplicação.

Tais informações quando observadas a nível geral ou por cada usuário da TaRGeT, propiciam uma visão que informa a assiduidade dos usuários no uso da TaRGeT, erros mais freqüentes cometidos pelos usuários e principalmente como os *test cases* estão sendo gerados. A partir daí, é possível tomar decisões de projeto mais seguras e identificar novas funcionalidades ou melhorias objetivando um melhor uso, aumento do desempenho e robustez da ferramenta.

Ao longo deste capítulo serão apresentadas estratégias de monitoramento a serem implementadas através do *logging* de informações geradas na utilização da TaRGeT.

3.1 Monitoramento de filtros e geração dos *test cases*

As funcionalidades relacionadas aos *test cases* são o foco principal da TaRGeT, pois tratam do objetivo final da ferramenta. Sendo assim, o monitoramento do modo como os filtros estejam sendo utilizados para geração da *suite* de testes como um todo é bastante importante. Estes procedimentos são realizados na tela *On The Fly Generation* que foi contemplada com monitoramento na escolha básica de filtros, armazenamento e salvamentos dos filtros como serão mostrados a seguir.

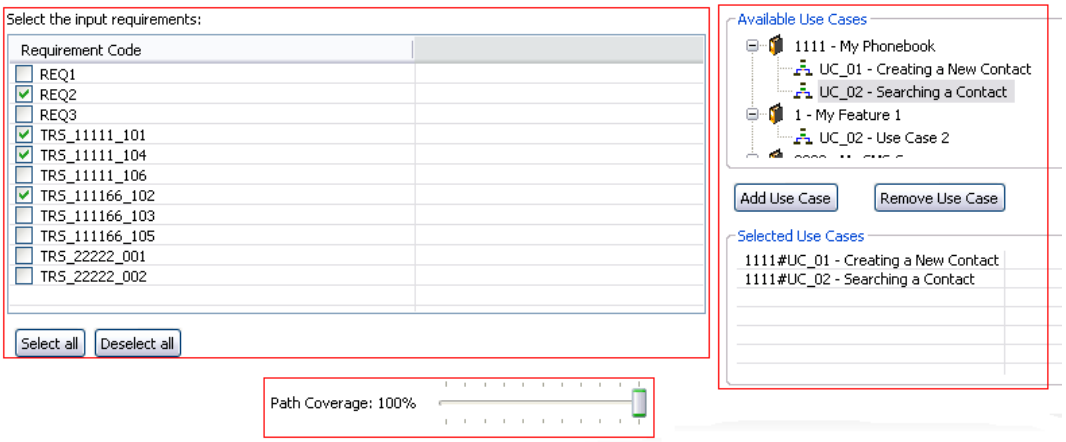


Figura 8 – Componentes de seleção a serem monitorados

A Figura 8 ilustra os componentes de seleção que serão monitorados. As informações de seleção dos três filtros poderão ser interceptadas no evento de "clique" da aba *Test Cases* que registrará a escolha básica dos filtros, no "clique" do botão *Save Current Filter* onde as configurações de filtro são salvas e no botão *Load Former Filter* onde uma configuração de filtro é carregada ou aberta. Ao serem interceptadas, essas informações ainda serão associadas ao usuário que as esteja utilizando, dessa forma será possível verificar quais usuários, ao gerarem a *test suite,* preferem criar ou salvar um ou mais filtros de seleção para posteriormente gerar os *test cases.*

Já no botão *Save Test Suite* inicia-se o monitoramento da geração de *test cases* registrando-se dados de casos de usos relacionados aos *test cases* da *suite* de teste. Análises desse monitoramento podem evidenciar se casos de usos não estão sendo usados por erros de usuários ou mesmo por estarem obsoletos. A partir daí, inúmeras decisões de projeto e novas funcionalidades poderão ser identificadas.

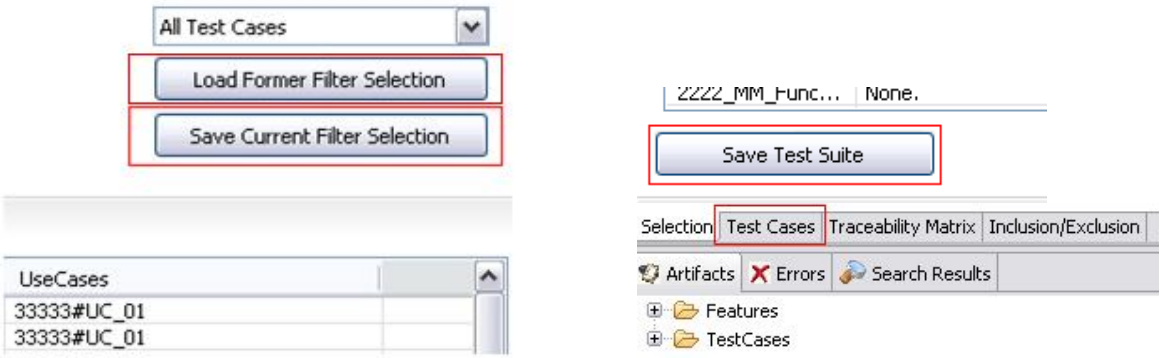


Figura 9 – Detalhamento dos componentes monitorados

O *plug-in* responsável por salvar os filtros e a *suite* de teste é o TaRGeT TC *Generation Gui (TCG).* Sendo assim, ele será monitorado afim de que sejam registradas as informações dos filtros utilizados pelo usuário e os casos de usos contemplados por cada *test case* contido na *suite* de testes.

3.2 Monitoramento das exceções

À medida que novas funcionalidades vão sendo adicionadas aos sistemas, maior as chances de acontecerem erros causados por mau uso dos usuários ou mesmo erros internos da aplicação. O *logging* das informações relacionadas ao contexto onde ocorrem as exceções esclarece os pontos da aplicação que estão sendo alvos de erros freqüentes. Caso sejam erros causados pelo mau uso da aplicação, o usuário poderá ser identificado facilitando assim a existência de um maior direcionamento na orientação de uso da aplicação. A identificação de freqüentes erros internos da aplicação será útil ao desenvolvedor que poderá verificar o erro sabendo onde e quando ocorreu e qual usuário o gerou, deixando o trabalho de correção ainda mais eficiente e efetivo.

O registro de tais informações é importante também porque nem todas as exceções levantadas pelas aplicações são mostradas ao usuário, sendo assim erros aparentemente imperceptíveis podem ocorrer, o que podem acarretar sérias conseqüências aos possíveis resultados obtidos pela aplicação. Até o momento, as principais exceções a serem levantadas pela TaRGeT são as de má formação de documentos *word* e xml. A primeira ocorre caso o documento de casos de uso tenha sido escrito em inconformidade em relação ao modelo de *template* do documento de casos de uso, já a segunda pode ocorrer quando o xml resultante da conversão do documento *word* tenha sido gerado de forma incorreta pela aplicação de conversão de *word* para XML, na maioria das vezes por erros no documento de entrada. Ambas as exceções ocorrem no *plug-in* TaRGeT *Project Manager* que terá as funções de processamento de documentos *word* e de *parser* xml monitoradas.

3.3 Monitoramento dos Projetos (TaRGeT Projects)

Diante das estratégias mencionadas, viu-se que eram necessárias maneiras de identificar os dados gerados pelo *logging* que não se limitassem apenas à associação aos usuários que os gerou, mas também aos projetos nos quais eles foram gerados. Dessa forma,

seria possível verificar além do usuário, o projeto em que foram acometidos os dados de *logging*. Esta característica permite a verificação de assiduidade dos usuários no uso da ferramenta, pois será possível observar os usuários que criam projetos e aqueles que continuam o trabalho manipulando projetos já criados.

Apresentadas as maneiras como será implantado o monitoramento por *loggings* na TaRGeT, serão vistas no próximo capítulo soluções que as implementam. No capítulo a seguir será vista uma abordagem de monitoramento em OO para as funcionalidades da TaRGeT vistas ao longo das seções acima. Serão analisados principalmente os problemas associados à implementação de estratégias de monitoramento nesta abordagem, notadamente verificados através da perda de modularidade causada por espalhamento e entrelaçamento de códigos de interesses adversos.

4. Abordagem de monitoramento em OO na TaRGeT

Depois de identificadas as estratégias de monitoramento úteis à TaRGeT, é possível implementá-las adicionando códigos interceptadores nos trechos de código que estejam em contextos importantes para o monitoramento. Os códigos interceptadores são chamadas a funções que capturam estruturas relevantes ao monitoramento para serem processadas e aproveitadas de acordo com o interesse do monitoramento em questão. Dessa forma, é possível capturar valores de objetos, atributos ou demais estruturas no momento em que as mesmas tenham dados que definam informações úteis ao monitoramento.

Nesta seção serão mostrados exemplos de abordagens de monitoramento em OO das funcionalidades vistas nos *plug-ins* apresentados no capítulo anterior (TaRGeT *Project Manager,* TaRGet *MSWordInput,* TaRGeT *Test Center* 3 *OutPut* e TaRGeT *TC Generation Gui)* com exemplos de código que interceptam tais *plug*-ins, notadamente seus interesses a serem monitorados. Porém, a abordagem em OO terá como objetivo evidenciar problemas associados à implementação de monitoramento em tal paradigma.

4.1 Monitoramento no TaRGeT *Project Manager*

Como visto anteriormente, o TaRGeT *Project Manager* é o *plug-in* responsável pelo gerenciamento das ações relacionadas aos projetos da TaRGeT. As funcionalidades compreendem criação e abertura de projetos que têm como objetivo a criação de um ambiente em que seja possível importação de documentos de casos de uso e posterior geração de *test cases* processados com base nesses documentos como já mencionado em outras seções. Há ainda a possibilidade de fechamento de projetos na TaRGeT, no entanto tal característica não faz parte das estratégias de monitoramento identificadas.

Analisando o *plug-in* foi possível identificar as classes responsáveis pela criação e abertura de projetos, são elas respectivamente *NewProjectWizard* e *OpenProjectWizard.* Ambas fazem parte do pacote de *wizards* do *plug-in* TaRGeT *Project Manager.* O objetivo do monitoramento nestas classes é identificar o momento em que o usuário cria ou abre um determinado projeto, por isso fez se necessário a interceptação em estruturas responsáveis pelo suporte a tal ação.

As classes *NewProjectWizard* e *OpenProjectWizard* fazem parte de estruturas básicas encontradas em aplicações *RCP* denominados de *Wizards*, componentes que dão suporte a maneiras flexíveis de prover ações do usuário sistematicamente para posterior validação das mesmas [3]. Tal característica explica o uso das classes citadas, já que ambas são acionadas no momento em que são requisitadas pelo usuário as tarefas de criação e abertura de projetos. Na Listagem 1, logo abaixo, segue um exemplo de trecho de código do método *init()* localizado na

classe *NewProjectWizard* com um código de monitoramento destacado interceptando alguns dados do projeto logo após sua criação. A chamada ao método *TargetMonitoring.interceptProjectCreation(...)* na linha 118 realiza a interceptação do nome e localização do projeto prestes a ser criado no método na linha 117. Este trecho está localizado no método *init()* que é responsável por iniciar o processo de criação de projetos acionado no evento de clique no botão *finish* do *New Project Wizard* (Figura 3, Cap. 2).

```
96  /* checks if exists a 'residual' project in an existing directory */
97  if (ProjectManagerController.getInstance().existsProjectInDirectory(destinationFolder))
98  {
99      if (new File(destinationFolder).canWrite())
100     {
101         result = GUIUtil
102                 .openYesOrNoError(
103                         getShell(),
104                         "Overwrite existing project",
105                         "An existing project was found in the specified directory. Do you wa
106         if (result)
107         {
108             File file = FileUtil.deleteAllFiles(dir + FileUtil.getSeparator() + name);
109             if (file != null)
110             {
111                 MessageDialog.openError(getShell(), "Error while creating project",
112                         "Cannot delete the file " + file.getAbsolutePath() + "");
113                 result = false;
114             }
115             else
116             {
117                 result = this.createProject(name, dir);
118                 TargetMonitoring.interceptProjectCreation(nome, dir);
119             }
120         }
121     }
122     else
123     {
124         MessageDialog.openError(getShell(), "Error while creating project",
```

Listagem 1 – Monitoramento de criação de projeto no método *init()*

O mesmo procedimento é feito para a interceptação na abertura de projetos. A classe *OpenProjectWizard,* também possui um método *init()* executado após evento escolha do projeto e posterior clique no botão *finish* do *Open Project Wizard.* Similar ao anterior este monitoramento também é realizado num método *init()* mas neste caso está localizado na classe *OpenProjectWizard.* Ao abrir o projeto é possível atingir sua instância através da chamada *controller.getInstance()* e de posse da instância, a obtenção dos dados do projeto se dá nas chamadas *getCurrentProject().getName()* e *getCurrentProject().getRootDir().* Estes dados são posteriormente interceptados pelo método *TargetMonitoring.interceptProjectOpening()*, como se pode ver na Listagem 2.

```
93  String projectFile = projectSelectionPage.getProjectFile();
94
95  /* Opens the project */
96  try
97  {
98      controller.openProject(projectFile);
99      TargetMonitoring.interceptProjectOpening(controller.getInstance().getCurrentProject().getName(), controller.getInstance().getCurrentProject().getRootDir());
100     /* Manages the project opening progress */
101     this.progressBar = new OpenProjectProgressBar();
102     ProgressMonitorDialog dialog = new ProgressMonitorDialog(this.parentShell);
103     dialog.setCancelable(false);
104     dialog.run(true, true, this.progressBar);
105
106     /* verifies the exception */
107     if (this.progressBar.getException() != null)
108     {
109         throw this.progressBar.getException();
110     }
111
112     GUIUtil.showPerspective(RequirementPerspective.ID);
113
114     /*
115      * the code below updates the tree views, the code must be here and not in the
116      * respective progress bar because when the progress bar is executed the perspective has
117      * not been created yet, so there is no how to update the views
118      */
119     GUIManager.getInstance().refreshViews();
120
121     GUIManager.getInstance().updateApplicationTitle();
122
123     result = true;
124
```

Listagem 2 – Monitoramento de abertura de projeto no *init()* do *Open Project Wizard*

4.2 Monitoramento do uso dos Filtros na TaRGeT

Como visto no capítulo 3, o monitoramento dos filtros acontecerá no evento do botão *Save Current Filter* para que sejam registrados os filtros salvos pelo usuário e no evento de "clique" da aba *Test Case* onde se inicia o processo de montagem da *suite* de testes e é possível interceptar o filtro atual. Na Página seguinte, seguem as listagens referentes ao trecho de código que monitora a utilização dos filtros a partir do evento do botão *save filter* e aba *Test Cases* respectivamente.

```
301
302         header.addSaveFilterButtonListener(new SelectionListener()
303         {
304
305             @Override
306             /**
307              * Sent when default selection occurs in the control.
308              * @param e an event containing information about the default selection
309              */
310             public void widgetDefaultSelected(SelectionEvent e)
311             {
312             }
313
314             @Override
315             /**
316              * Sent when selection occurs in the control.
317              * @param e an event containing information about the selection
318              */
319             public void widgetSelected(SelectionEvent e)
320             {
321                 TestSuiteFilterAssembler assembler = selectionPage.getFilterAssembler();
322                 header.addFilter(assembler);
323                 TargetMonitoring.interceptSaveFilter(assembler);
324             }
325
```

Listagem 3 - Detalhamento do monitoramento no evento do save current filter

A Listagem 3 ilustra a criação do *Listener* (linha 302) do botão *save filter* que dispara o evento para salvar o filtro corrente. Tal evento quando acionado, causa a chamada imediata do método *widgetSelected()* onde as configurações do filtro (*TestFilterAssembler*) são resgatadas (linha 321). Depois de resgatado no método *getFilterAssembler(),* o filtro pode ser interceptado pelo código de monitoramento destacado. O objeto *TestFilterAssembler* contém o detalhamento de todos os filtros da TaRGeT, sendo assim a instância assembler criada já possui todos os dados necessários para registro das informações dos filtros a serem salvos.`

A Listagem 4 ilustra a interceptação do filtro no momento em que o usuário "clica" na aba *Test Case.* Neste momento é executado o método *setFocus(),* reponsável por dar o foco à aba *test case.* Como pode ser visto no código, foram necessárias chamadas a métodos que resgatassem o filtro corrente (linha 640) para que seu resultado fosse interceptado

```
632     /**
633      * @Override
634      * @see IWorkbenchPart#setFocus()
635      */
636     public void setFocus()
637     {
638         super.setFocus();
639         this.updateTestCaseListTable();
640         TestSuiteFilterAssembler assembler = header.getFormerAssembler();
641         formerAssembler.assemblyFilter().filter(
642                 this.getEditor().getRawTestSuite());
643         TargetMonitoring.interceptSaveFilter(assembler);
644
645     }
646
```

Listagem 4 – Interceptação de filtro no "clique" da aba Test Cases

4.3 Monitoramento na geração dos *Test Cases*

Os *test cases* serão monitorados a partir do momento em que o usuário "clicar" no botão *Save Test Suite.* A partir do "clique" neste botão, a execução passa por dois trechos do código relevantes à interceptação dos *Test Cases*. Primeiramente a interceptação poderia ser feita no evento *widgetSelected()* onde é levantado o evento do *save test suite* e criada a *suite* de testes na atribuição ao objeto do tipo *TestSuite*. A outra opção seria a interceptação do método *mountTestCaseInfo()* localizado no *plug-in* TaRGeT *Test Center* 3 *OutPut.* O método *mountTestCaseInfo()* é responsável por gerar o arquivo *excel* final dos *test cases,* ele é útil pelo fato de utilizar como parâmetro instâncias do tipo *TextualTestCase* que possuem bem mais informações em relação ao objeto *TestSuite* interceptado no evento *widgetSelected()*, no entanto o método *mountTestCaseInfo()* é chamado para cada *test case* na *test suite.* Abaixo são mostradas as Listagens referentes a estas abordagens.

```
402 saveCurrentTestSuite.setLayoutData(gridData);
403 saveCurrentTestSuite.addSelectionListener(new SelectionListener()
404 {
405
406     public void widgetDefaultSelected(SelectionEvent e)
407     {
408     }
409
410     public void widgetSelected(SelectionEvent e)
411     {
412         TestSuite<TestCase<FlowStep>> testSuite = selectionPage.getCurrentTestSuite();
413         TargetMonitoring.interceptTestCaseGeneration(testSuite);
414
415         List<TextualTestCase> textualTestCases = new ArrayList<TextualTestCase>();
416         for (TestCase<FlowStep> testCase : testSuite.getTestCases())
417         {
418             textualTestCases.add(getEditor().getTextualTestCase(testCase.getId()));
419         }
420
```

Listagem 5 – Interceptação de suite de test cases no evento do save test

```
151   public void mountTestCaseInfo(int tcCount, TextualTestCase testCase)
152   {
153
154     TargetMonitoring.interceptTestCaseGeneration(testCase);
155     this.excelGenerator.createNextRow();
156
157     // Case number
158     this.excelGenerator.createCellTextAreaStyleWithBorders(0, testCase.getTcIdHeader()+ "-
159
160     // Reg Level
161     this.excelGenerator.createCellTextAreaStyleWithBorders(1,testCase.getRegressionLevel()
162
163     // Exe Type
164     this.excelGenerator.createCellTextAreaStyleWithBorders(2,testCase.getExecutionType());
165
166     // Description/Name
167     this.excelGenerator.createCellTextAreaStyleWithBorders(3,testCase.getDescription());
168
169     // Procedure/Objective
170     this.excelGenerator.createCellTextAreaStyleWithBorders(4,testCase.getObjective());
171
```

Listagem 6 – Interceptação de suite de test cases no mountTestCaseInfo()

4.4 Monitoramento das Exceções

Por fim será apresentado um trecho de código em que se podem usar monitoramento para capturar exceções levantadas pelo usuário. As principais exceções que ocorrem na TaRGeT são as de documento ou xml mal formado, no Listagem logo abaixo há um exemplo de interceptação de exceção que ocorre na classe *WordDocumentProcessing.*

```
315  try
316  {
317       UseCaseDocumentXMLParser ucDocParser = new UseCaseDocumentXMLParser(xmlFile);
318       phone = ucDocParser.buildPhoneDocument();
319       phone.setDocFilePath(docFileNames.get(i));
320       phone.setLastDocumentModification(lastModified);
321  }
322  catch (UseCaseDocumentXMLException e)
323  {
324       phone = new PhoneDocument(docFileNames.get(i), lastModified);
325       errorMessage = "An error occurred while parsing the XML content extracted from
326                + FileUtil.getFileName(docFileNames.get(i))
327                + ". The XML may be malformed.";
328
329       TargetMonitoring.interceptException(errorMessage);
330  }
```

Listagem 7 – Monitoramento de Exceções

4.5 Problemas em abordagens de monitoramento em OO

A partir das soluções em OO mostradas acima é possível tomar conclusões importantes acerca delas. Primeiramente, é notório o espalhamento dos código de monitoramento (destacados em vermelho) ao longo das classes, ou seja, códigos de monitoramento estão localizado em várias classes, métodos e *plug-ins* do projeto como um todo. Com o acréscimo de mais estratégias de monitoramento, o nível de espalhamento só tende a aumentar ainda mais, o que traria sérios problemas de modularidade e reuso da aplicação. Outro problema percebido é o entrelaçamento do código, pois códigos de interesses particulares, no caso, monitoramento, estão entrelaçado aos códigos comuns da TaRGeT.

Problemas de surgimento de códigos entrelaçados ou espalhados em diversas classes denotam uma característica maior que é a perda de modularidade no desenvolvimento de interesses específicos das aplicações. A perda de modularidade na abordagem em OO traz consigo dificuldades no desenvolvimento paralelo dos diversos interesses das aplicações fazendo com que, neste caso, o desenvolvimento da estrutura de monitoramento seja dependente da TaRGeT. Com isso o projeto acaba perdendo muito em eficiência e capacidade de reuso o que dificulta também o processo de evolução do *software.*

Diante de tais problemas, o uso de aspectos neste contexto pode vir como uma solução eficiente ao processo de desenvolvimento da TaRGeT. No próximo capítulo será apresentada a implementação de um *plug-in* em aspectos que provê soluções que anulam os problemas de modularidade identificados na abordagem OO.

5. Solucionando problemas de modularidade com Implementação de abordagem em Aspectos

Esta seção detalha a implementação de um *plug-in* em aspectos capaz de realizar monitoramento das funcionalidades da TaRGeT apresentadas no capítulo 3. Tal implementação objetivou a criação de um *plug-in* que anulasse os problemas identificados no capítulo anterior, de modo a resgatar características de modularidade e reuso ao código da TaRGeT.

Ao longo deste capítulo são mostrados conceitos relacionados à Programação Orientada a Aspectos (POA) e os motivos que fazem com que seu uso possa resgatar modularidade no desenvolvimento de aplicações. Posteriormente, serão apresentadas detalhadamente as soluções em aspectos implementadas para cada interesse de monitoramento identificado na TaRGeT e a motivação para o uso de *Equinox Aspects* no desenvolvimento do TaRGeT *Monitoring Plug-In*.

5.1 Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) surgiu como solução aos problemas causados por perda de modularidade em aplicações em geral. Tais problemas ocorrem por conta do acúmo de interesses transverssais (*crosscutting)* no código da aplicação como um todo que pode ser percebido através do espalhamento (*scattering)* e entrelaçamento (*tangling)* de trechos de código de interesse particular por uma ou mais classes ou métodos de códigos de interesses próprios da aplicação.

Com o uso de Programação Orientada a Aspectos é possível eliminar possíveis problemas de modularidade transportando os interesses transversais à nova unidade de encapsulamento denominada aspecto. O código contido nos aspectos é combinado ao código orientado a objetos diretamente no *bytecode,* através de um processo denominado *weaving.*

Para utilizar POA no desenvolvimento de aplicações é necessário o uso da linguagem *AspectJ* que é uma extensão orientada a aspectos da linguagem Java. A seguir serão mostradas as principais construções de *AspectJ* [15].

5.1.1 *Pointcuts*

Construção sintática do AspectJ capaz de agrupar um conjunto de pontos de junção (*join points).* Neste conjunto podem estar inclusos execuções ou chamadas a métodos, construtores, inicializadores estáticos, referências a um determinado atributo, entre outros. Todos eles podem ser compostos, utilizando os operadores | | (or), && (and) e ! (not) [16].

Nos *pointcuts* é bastante comum o uso de estruturas que expõem o contexto de objetos em diversas maneiras, podendo ser interceptados objetos passados como parâmetro, retornados ou que são alvo de chamadas a métodos. As palavras reservadas mais utilizadas para este intuito são *target(), this() e args()* [17]. Há também as estruturas de controle de fluxo como *cflow(), cflowbelow()* etc.

5.1.2 *Advices*

Advices especificam código adicional a ser executado e quando ele deve ser executado. Existem três possíveis tipos de *advice*: *before*, *after* e *around*. O código definido nestes *advices* poderá ser executado respectivamente antes, depois ou durante a execução dos *join points* capturados nos *pointcuts* [16].

5.1.3 *Inter-Type* Declarations

As *inter-type declarations* representam mudanças estruturais no código do programa, diferentemente do uso de *pointcuts* e *advices*, que modificam a dinâmica de execução. Estas mudanças estruturais podem ser tanto a introdução de novos métodos e atributos a classes ou interfaces existentes, quanto modificações da hierarquia de classes [16].

5.1.4 Instâncias de Aspectos

Em POA é possível criar instâncias de aspectos referentes a cada *join point* interceptado por determinado *pointcut* através das chamadas *perClauses.* O uso de instâncias de aspectos é útil notadamente em aspectos que possuam atributos, pois, no caso de um atributo ser modificado por mais de um *advice,* o seu valor pode ficar inconsistente ao longo da aplicação, já que poderá ser modificado em diferentes momentos. O uso de instância de aspectos preserva o valor do atributo que esteja contido na instância criada. Os *poincuts* a serem interceptados são passados como parâmetros na *perClause.* As contruções de perclause mais comuns são *perthis(pointcut), pertarget(pointcut), percflow(pointcut) e percflowbelow(pointcut)* [17].

5.2 *Equinox Aspects*

A utilização comum de aspectos em aplicações *Eclipse RCP* nem sempre pode garantir o modularidade. Isto ocorre porque o *AspectJ Development Tool* (AJDT) [11] até então não era capaz de prover interceptação em outros *plug-ins* JDT [10] em tempo de execução (*load time weaving),* ou seja, os *plug-in* JDT não poderiam ser interceptados por aspectos definidos em *plug-ins* AJDT. Se os JDT-*Plug-Ins* forem compilados utilizando os aspectos internamente, seria possível interceptar dados apenas do *plug-in* em questão, no entanto esta solução não é muito satisfatória em projetos com mais de um *plug-in*, visto que, caso o aspecto implementado tivesse que interceptar mais de um *plug-in* ele teria que ser replicado em cada um.

Buscando uma solução que eliminasse a necessidade de replicação dos aspectos no desenvolvimento de aplicações *RCP,* foi desenvolvido por membros da *Eclipse Foundation* [14] um *framework* que permite ao AJDT a realização de *load time weaving* em JDT-*Plug-Ins* [13] [12]. Denominado de *Equinox Aspects,* este *framework* permite ao AJDT a possibilidade de ser mais um *plug-in (bundle)* na pilha de *plug-ins* que compõe as aplicações *RCP* e interceptar qualquer trecho de código de qualquer *plug-in* de determinada aplicação [13]. A TaRGeT se encaixou nesse perfil por ser uma aplicação *RCP* e possuir diversos *plug-ins* com necessidades de monitoramento. No entanto, por ter sido desenvolvido recentemente, algumas funcionalidades ainda não funcionam adequadamente ou de fato não funcionam como será demonstrado ao longo desse capítulo. O Apêndice A na Página 55 explica resumidamente alguns passos para a instalação de *Equinox Aspects* no *Eclipse*.

5.3 Monitoramento da TaRGeT com *plug-in* em Aspectos

Objetivando a adição de características de monitoramento que garantam a permanência do desenvolvimento modular da TaRGeT, foi desenvolvido um *plug-in* em *Equinox Aspects* denominado TaRGeT *Monitoring Plug-In* como parte integrante deste trabalho de graduação. Com o TaRGeT *Monitoring Plug-In,* será possível monitorar as principais funcionalidades da TaRGeT evitando o entrelaçamento de códigos de interesse do monitoramento ao código dos outros *plug-ins* da TaRGeT. O fato de o aspecto estar definido num *plug-in* independente, permite também a vantagem da característica *plug and play*, permitindo facilmente que o *plug-in* seja “desplugado” da aplicação quando necessário, sem a necessidade de comentar códigos ou qualquer medida manual que impeça a sua inicialização.

`

5.4 TaRGeT *Monitoring Plug-in*

O TaRGeT *Monitoring Plug-in* foi desenvolvido com o objetivo de registrar informações de *logging* no uso da TaRGeT atuando como um componente monitor das funcionalidades relacionadas a geração de *test cases,* configuração de filtros, manipulação de projetos e levantamento de exceções. A interceptação de tais funcionalidades é realizada por aspectos direcionados a cada funcionalidade. Portanto, foram criados os seguintes aspectos: *ExceptionMonitoring para interceptação de exceções, FilterMonitoring* para interceptação de configurações de filtros *TestCaseMonitoring* para interceptação de dados dos *test cases, ProjectMonitoring* responsável pela interceptação na criação e abertura de projetos e *AbstractListener,* aspecto abstrato herdado nos aspectos que interceptam códigos contidos em *listeners.*

Os dados interceptados pelos aspectos estão sendo registrados num banco de dados SQL *Server* através do *driver* JDBC*.* Para que fosse realizada a comunicação com o banco foi criada uma classe denominada *JdbcConnection* que contém os métodos de transação. Foi também criado o aspecto denominado *BdConnectionControl* (Listagem 9) para interceptação dos métodos da *JdbcConnection* chamados nos advices dos aspectos apresentados mais acima. O aspecto *BdConnectionControl* possibilita um controle maior nas chamadas de tais métodos que ao serem executados nos *advices* do *BdConnectionControl* evitam o espalhamento dos códigos de abertura (*connect())* e fechamento de conexão (*closeConextion())* nos aspectos de monitoramento. Como pode ser visto ainda neste aspecto, foram utilizadas as contruções *around* nos *advices.* Isso foi necessário para que pudesse ser realizado o tratamento das possíveis exceções a serem levantadas pelos métodos da *JdbCConnection*. *Outra* vantagem trazida pelo *BdConnectionControl* foi o tratamento das exceções de banco. O trecho de código a seguir (Listagem 8), pertence à classe *JdbcConnection* onde pode-se notar que não houve necessidade de tratamento de exceção*.*

```
public static void generateTestCase(String header, String useCases) throws SQLException {
    PreparedStatement pstmt;
    pstmt = con.prepareStatement("{ call sp_generateTestCase(?, ?, ?, ?) }");
    pstmt.setString(1, header);
    pstmt.setString(2, useCases);
    pstmt.setInt(3, userId);
    pstmt.setInt(4, projectUserId);
    pstmt.execute();
}

public static void generateException(String name) throws SQLException {
    PreparedStatement pstmt;
    pstmt = con.prepareStatement("{ call sp_generateException(?,?) }");
    pstmt.setString(1, name);
    pstmt.setInt(2, projectUserId);
    pstmt.execute();
}

public static void generateUser() throws SQLException {
    CallableStatement cstmt = null;
    ResultSet rs = null;
    Statement stmt = null;
```

Listagem 8 - Trechos de códigos da classe JdbcConnection sem tratamento de exceções

```
5  public aspect BdConnetionControl {
6      boolean connection = true;
7      String user = "jp";
8
9      pointcut generateUser() :
10         execution(* JdbcConnection.generateUser()) && !within(BdConnetionControl);
11
12     pointcut generateFilter(String s1, String s2) :
13         execution(* JdbcConnection.generateFilter(String, String))
14         && args(s1,s2) && !within(BdConnetionControl);
15
16     pointcut generateProject( String s1, String s2, String s3) :
17         execution(* JdbcConnection.generateProject(String, String, String))
18         && args(s1,s2,s3) && !within(BdConnetionControl);
19
20     pointcut generateTestCase( String s1, String s2) :
21         execution(* JdbcConnection.generateTestCase(String, String))
22         && args(s1,s2) && !within(BdConnetionControl);
23
24
25     pointcut generateException(String name) :
26         execution(* JdbcConnection.generateException(String)) && args(name);
27
28     void around() : generateUser() {
29         try {
30             if (connection) {
31                 JdbcConnection.connect(user);
32                 proceed();
33             }
34         } catch (SQLException e) {
35
36             e.printStackTrace();
37         }
38     }
```

Listagem 9 – Controle e tratamento de exceções dos métodos da classe JdbcConnection

O TaRGeT *Monitoring* foi desenvolvido no mesmo *workspace* dos *plug-ins* da TaRGeT e configurado para interceptar códigos contidos nos *plug-ins* TaRGeT *Project Manager* voltado a manipulação dos projetos da TaRGeT, TaRGeT *MSWord Input* responsável pelo processamento de documentos *word* e o TaRGeT TC *Generation Gui* (TCG). A Figura 9 na próxima Página ilustra o *workspace* da TaRGeT detalhando o TaRGet *Monitoring*, seus artefatos e os *plug-ins* a serem interceptados por ele.

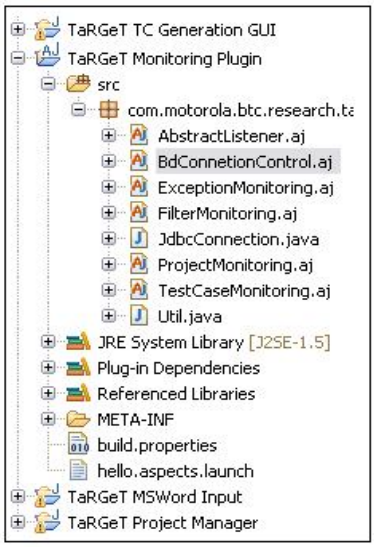


Figura 9 – Detalhamento do TaRGeT Monitoring e demais plug-ins no workspace

Nos tópicos seguintes, será detalhada a implementação de cada aspecto implementado no TaRGeT *Monitoring Plug-In.* Levando em consideração os trechos de códigos da TaRGeT interceptados por eles.

5.5 Aspecto de monitoramento de projetos

Com o intuito de registrar infomações da criação e abertura de projetos, foi criado no TaRGeT *Monitoring* o aspecto *ProjectMonitoring* que intercepta o método *init()* das classes *NewProjectWizard e OpenProjectWizard.* O método *init(),* como ilustrado respectivamente nas Listagens 1 e 2 de tais classes, foi escolhido para ser interceptado apenas por ser executado no momento em que os projetos são criados ou abertos. Sendo assim, não foram interceptadas instâncias criadas em tal método o que apenas exigiu a execução dos *advices* posterior ao *init()* pois os dados do projeto já estariam carregados. As declarações de levantamento da exceção *SQLException* nos advices foi feita devido ao aspecto *BdConnectionControl* que é o responsável

por tratar tais exceções quando os fluxos das chamadas aos métodos de transação são chamados em seus *advices.*

```
12 public aspect ProjectMonitoring {
13
14
15     pointcut openProject() : this(OpenProjectWizard) && call(boolean init());
16
17     pointcut newProject() : this(NewProjectWizard) && call(boolean init());
18
19
20     after () throws SQLException : openProject()  {
21         JdbcConnection.generateUser();
22         TargetProject proj = ProjectManagerController.getInstance().getCurrentProject();
23         JdbcConnection.generateProject("u", proj.getRootDir(), proj.getName());
24     }
25
26     after() throws SQLException : newProject() {
27         TargetProject proj = ProjectManagerController.getInstance().getCurrentProject();
28         JdbcConnection.generateUser();
29         JdbcConnection.generateProject("i", proj.getRootDir(), proj.getName());
30     }
31
32 }
33
```

Listagem 10 – Aspecto que realiza monitoramento na criação e abertura de projetos

Como pode ser visto, foram criados dois *pointcuts* (*openProject() e newProject())* que realizam respectivamente a interceptação nos métodos *init()* das classes *NewProjectWizard* e *OpenProjectWizard* respectivamente, para que seu fluxo de execução seja continuado nos *advices* definidos nas linhas 20 e 26 da Listagem 8. Os *advices* do aspecto *ProjectMonitoring* registram o usuário e os dados do projeto que esteja sendo manipulado através da conexão com JDBC a ser explicada mais adiante. Dessa forma, foi possível realizar a interceptação dos dados do projeto transportando o código de relacionado ao monitoramento para o aspecto em questão.

5.6 Aspecto de monitoramento das exceções

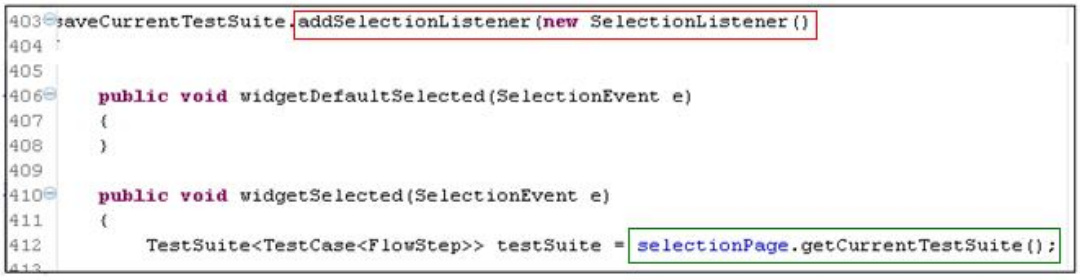
A listagem 9 ilustra a implementação do aspecto *ExceptionMonitoring* que contemplou a ocorrência de exceções no processamento dos documentos *word* no *pointcut createWordDocument*, processo de parser do documento *word* para transformação em xml no *pointcut UCDocXMLParser* e na atualização do projeto interceptada no *pointcut TargetRefresh.* Depois de interceptadas, as mensagens relacionadas às exceções são registradas. Como pode ser analisado, foi utilizada a contrução *after* e *throwing* ilustrando o que será feita após o levantamento da exceção. Apesar de existirem exceções envolvidas, a construção *around* não foi utilizada porque as exceções que acontecerem serão levantadas aos *advices* do aspecto *BdConnectionControl.*

```
14 public aspect ExceptionMonitoring {
15
16     pointcut TargetRefresh(boolean throwExceptionOnFatalError,int operation ) :
17         execution(private TargetProjectRefreshInformation TargetProjectRefresher.reloadProject(boolean , int))
18         && args(throwExceptionOnFatalError,operation);
19
20
21     pointcut createWordDocumentObj(List<String> doc, boolean e) :
22         execution(public List<PhoneDocument> WordDocumentProcessing.
23                 createObjectsFromWordDocument(List<String>, boolean)) && args(doc,e);
24
25
26     pointcut UCDocXMLParser(UseCaseDocumentXMLParser uc, File f) :
27         execution(public UseCaseDocumentXMLParser.new(File)) && args(f) && this(uc);
28
29
30     after(List<String> doc, boolean e) throwing(UseCaseDocumentXMLException e1) throws SQLException:
31         createWordDocumentObj(doc, e) {
32         JdbcConnection.generateException("Erro ao adicionar Doc. Documento Mal Formado");
33     }
34
35
36     after(UseCaseDocumentXMLParser uc, File f) throwing(UseCaseDocumentXMLException u) throws SQLException :
37         UCDocXMLParser(uc,f) {
38         JdbcConnection.generateException("Erro no XML Parser. XML mal formado");
39     }
40
41     after(boolean b,int op) throwing(TargetException t) throws SQLException : TargetRefresh(b,op) {
42         JdbcConnection.generateException(t.getMessage());
43     }
44 }
```

Listagem 11 – Aspecto de monitoramento de Exceções

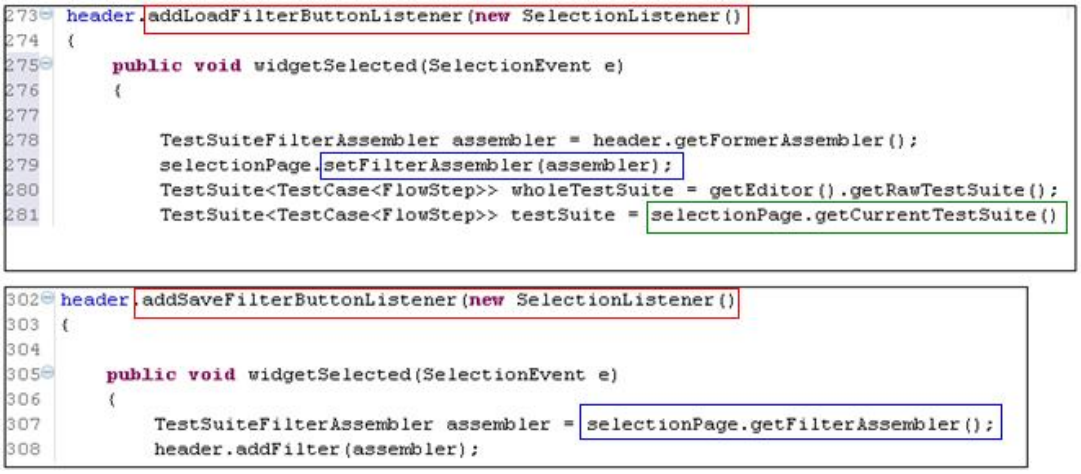
5.7 Aspecto de monitoramento de funcionalidades executadas em Observadores (*Listeners)*

Nas estratégias de monitoramento identificadas, há algumas funcionalidades cuja interceptação passa por fluxos de *listeners* que são classes responsáveis por disparar eventos associados a componentes como botões ou *checkboxes.* Tais funcionalidades são o salvamento do filtro (*save current filter)*, carregamento de filtros (*load current filter)* e salvamento da *suite* de *test cases (save test suite).* Ambos os eventos ocorrem no método *widgetSelected(),* definido na classe *SelectionListener* que é instanciada como *Listener* dos botões dessas funcionalidades. A seguir seguem as listagens onde são demonstradas a localização da criação das instâncias *SelectionListener* e as chamadas aos métodos de geração de *test cases (getCurrentTestSuite(),* listagem 12*)* e os métodos relacionados a carregamento e salvamento de filtros (*setFilterAssembler() e getFilterAssembler(),* listagem 13*).*



```
403 saveCurrentTestSuite.addSelectionListener(new SelectionListener()
404 {
405
406     public void widgetDefaultSelected(SelectionEvent e)
407     {
408     }
409
410     public void widgetSelected(SelectionEvent e)
411     {
412         TestSuite<TestCase<FlowStep>> testSuite = selectionPage.getCurrentTestSuite();
```

Listagem 12 – Trecho do método createSaveTestSuite onde é instanciado o evento para geração dos test cases no botão save test suite



```
273 header.addLoadFilterButtonListener(new SelectionListener()
274 {
275     public void widgetSelected(SelectionEvent e)
276     {
277
278         TestSuiteFilterAssembler assembler = header.getFormerAssembler();
279         selectionPage.setFilterAssembler(assembler);
280         TestSuite<TestCase<FlowStep>> wholeTestSuite = getEditor().getRawTestSuite();
281         TestSuite<TestCase<FlowStep>> testSuite = selectionPage.getCurrentTestSuite()
```

```
302 header.addSaveFilterButtonListener(new SelectionListener()
303 {
304
305     public void widgetSelected(SelectionEvent e)
306     {
307         TestSuiteFilterAssembler assembler = selectionPage.getFilterAssembler();
308         header.addFilter(assembler);
```

Listagem 13 – Trecho do método createHeaderSection onde são instanciados os eventos para carregamento e salvamento de configurações de filtro respectivamente

No entanto, caso o método a ser interceptado esteja sendo chamado em mais de uma instância de evento da mesma classe, como é o caso das chamadas ao *getCurrentTestSuite()* destacado na Listagem 12 (linha 412) e Listagem 13 (linha 281), não é possível identificar a origem da chamada destes método por aspectos, pois, apesar de estarem em locais diferentes, os métodos são os mesmos e são chamados em fluxos pertencentes ao mesmo método da mesma classe, o que impossibilita distinguir suas execuções como sendo pertecentes a *join points* diferentes.

Logo, para que a origem da chamada do método possa ser identificada no aspecto, é necessário verificar o valor da referência que originou o evento. Para isso, nos aspectos que interceptam chamadas contidas em *listeners,* foram implementados *pointcuts* que interceptam a criação da instância do *listener* comentadas anteriormente.

Abaixo seguem as Listagens 14 e 16 que ilustram o aspecto *TestCaseMonitoring* e *FilterMonitoring* respectivamente. O *TestCaseMonitoring* intercepta o método *getCurrentTestSuite()* no *pointcut generateTests(),* porém como ele faz parte do fluxo do *SelectionListener* que é utilizado em outros contextos, foi criado o *pointcut getListener()* para interceptar o valor da referência do *SelectionListener* com o objetivo de identificar o contexto da ocorrência do evento. O *advice* localizado na linha 31 da listagem 14, intercepta a inicialização da instância do *SelectionListener* para armazená-la no atributo *selectionListener.* Esta inicialização (linha 403, listagem 12) ocorre dentro do fluxo do método *createSaveTestSuiteButton(),* por isso este método é utilizado no *cflowbelow* do *pointcut getListener()* do aspecto *TestCaseMonitoring.*

De posse do valor da referência do *listener,* é possível interceptar o método *getCurrentTest()* no *pointcut generateTests()* identificando se ele foi chamado no evento do botão *save current test.* Esta identificação é feita no advice localizado na linha 35 que verifica se a instância *SelectionListener* (passada ao *pointcut SelectionListener)* cujo fluxo gerou a chamada ao método *getCurrentTestSuite()* é igual ao valor da referência do *SelectionListener* criada no botão *save test* e armazenada no atributo *selectonListener.* Caso sejam iguais, significa que o método *getCurrentTestSuite()* foi de fato chamado na execução do evento do botão *save test suite.*

```
17 public aspect TestCaseMonitoring extends AbstractListener {
18
19     public SelectionListener selectionListener;
20
21     public pointcut getListener(SelectionListener sl) :
22         execution(*  Button.addSelectionListener(SelectionListener))
23     && cflowbelow(execution(*  OnTheFlyGeneratedTestCasesPage.
24             createSaveTestSuiteButton(Composite))) && args(sl);
25
26     pointcut generateTests(OnTheFlyTestSelectionPage selectPage) :
27         execution(* OnTheFlyTestSelectionPage.getCurrentTestSuite())
28         && target(selectPage);
29
30
31     before(SelectionListener sl) : getListener(sl) {
32         selectionListener = sl;
33     }
34
35     after (SelectionListener stl, OnTheFlyTestSelectionPage selectPage)
36     returning (TestSuite<TestCase<FlowStep>> testSuite):
37         generateTests(selectPage) && SelectionListenerFlow(stl) {
38
39         if (stl == this.selectionListener) {
40             System.out.println("Intercepting Filters from Save Test Suite Event...");
41         }
42     }
43
44     pointcut interceptTestGeneration(TextualTestCase t) :
45         execution(* ExcelFileFormatter.mountTestCaseInfo(int,TextualTestCase)) && args(..,t);
46
47
48     before(TextualTestCase t)  throws SQLException : interceptTestGeneration(t) {
49         JdbcConnection.generateTestCase( t.getTcIdHeader()+t.getId(), t.getUseCaseReferences());
50     }
```

Listagem 14 – Interceptação de métodos de geração de test cases

O mesmo processo ocorre no aspecto *FilterMonitoring* (Listagens 15 e 16), ou seja, foram implementados aspectos que interceptam métodos chamados dentro de fluxos de *listeners* como é o caso dos métodos *getFilterAssembler()* (linha 307, listagem 12) e *setFilterAssembler()* (linha 279, listagem 12). Os *listeners* que deram origem à chamada destes métodos foram instanciados dentro do método *createHeaderSection()* que é chamado no evento de "clique" da aba *Test Cases.* Por este motivo, no *pointcut getListener()* (linha 35, listagem 15) do aspecto *FilterMonitoring*, além de, neste mesmo *poincut,* terem sido inclusas as chamadas aos métodos onde as instâncias dos *SelectionListener* são criadas*,* foi incluída também a necessidade de o fluxo estar abaixo do método *createHeaderSection()* com a construção *cflowbelow()* (linha 38, listagem 15).

Dessa forma, é possível interceptar a criação das instâncias de evento do tipo *SelectionListener* no advice localizado na linha 42 da listagem 15. Neste *advice,* ainda foi feita a verificação do *join point* interceptado, já que o *pointcut getListener()* do aspecto em questão define dois *join points* que podem ser originados respectivamente a partir de chamadas aos métodos *addSaveFilterButtonListener()* e *addLoadFilterButtonListener().* Tal verificação permite a atribuição aos atributos *saveListener* e *loadListener* referente aos *join points* citados.

De posse das instâncias de eventos que podem ser levantadas nos *join points* de salvamento e carregamento de filtros, será possível identificar o contexto da execução de tais *join points* nos *advices* localizados nas linhas 42 (listagem 15) e 57 respectivamente (listagem 14) quando os métodos definidos em tais *join points* forem executados.

```
17 public aspect FilterMonitoring extends AbstractListener   {
18
19     public SelectionListener saveListener;
20
21     public SelectionListener loadListener;
22
23     pointcut saveFilter() : execution(* OnTheFlyTestSelectionPage.getFilterAssembler())
24     && cflowbelow(execution(* widgetSelected(SelectionEvent)));
25
26     pointcut loadFilter(TestSuiteFilterAssembler assembler) :
27         execution(* TestSuiteFilterAssembler.assemblyFilter()) && target(assembler)
28         && cflowbelow(execution(* OnTheFlyTestSelectionPage.getCurrentTestSuite()));
29
30     pointcut changeTab(TestSuiteFilterAssembler assembler) :
31         loadFilter(assembler)
32         && cflowbelow(execution(* OnTheFlyGeneratedTestCasesPage.setFocus()));
33
34
35     public pointcut getListener(SelectionListener sl) :
36         (execution(* OnTheFlyHeader.addSaveFilterButtonListener(SelectionListener))
37          ||execution(* OnTheFlyHeader.addLoadFilterButtonListener(SelectionListener)))
38         && (cflowbelow(execution(* OnTheFlyGeneratedTestCasesPage.createHeaderSection(St
39         && args(sl));
40
41
42     before(SelectionListener sl) : getListener(sl) {
43         if (Util.loadFilterJoinPoint.equals(thisJoinPoint.toString())) {
44             loadListener = sl;
45         } else {
46             saveListener = sl;
47         }
48     }
49
```

Listagem 15 – Aspecto que intercepta métodos de salvamento de filtros

```
51      }
52
53      after(TestSuiteFilterAssembler assembler) throws SQLException : changeTab(assembler) {
54          Util.trackingFilters(assembler, 'c');
55      }
56
57      after(SelectionListener sfl) returning (TestSuiteFilterAssembler assembler) throws SQLException :
58          saveFilter() && SelectionListenerFlow(sfl)    {
59          if (sfl == saveListener) {
60              Util.trackingFilters(assembler, 's');
61          }
62      }
```

Listagem 16 – Aspecto que intercepta métodos de configuração de filtros (Continuação)

Estes *advices* verificam se a referência do evento que os levantou são iguais as referências armazenadas nos atributos *loadListener* ou *saveListener* anteriormente.

Ainda no aspecto em questão, há o *pointcut changeTab()* (linha 31, listagem 15) que utilizando o *pointcut loadFilter()* intercepta o método *assemblyFilter()* chamado dentro do fluxo do *getCurrentTestSuite().* O *changeTab()* tem como objetivo interceptar os filtros criados no momento em que o usuário "clica" na aba *Test Cases,* para isso ele precisa estar abaixo do fluxo do método *setFocus().* No entanto, apesar de representar um evento de "clique", não há instâncias de eventos envolvidas no *setFocus()*, o que deixou mais simples a implementação desta funcionalidade. O *advice* da linha 53, listagem 16 intercepta o *pointcut changeTab().*

5.7.1Uso de Herança

Tanto o aspceto *TestCaseMonitoring* quanto o *FilterMonitoring* interceptam chamadas a funções contidas em eventos. O fato de ambos os aspectos possuírem tal característica, motivou o uso de herança em sua implementação através da classe *AbstractListener* (Listagem 17). A classe *AbstractListener* possui o *getListener()* como *pointcut* abstrato (*abstract)* e o *pointcut SelectionListenerFlow* que intercepta quaisquer chamadas que ocorram dentro de todos os fluxos pertencentes à classe *SelectionListener.*

```
6   public abstract aspect AbstractListener {
7
8       public abstract pointcut getListener(SelectionListener sl);
9
10      pointcut SelectionListenerFlow(SelectionListener sl):
11          cflow(execution(* * (..)) && this(sl));
12
13  }
```

Listagem 17 – Aspecto Abstrato para interceptação de objetos SelectionListener

O modificador *abstract* no *pointcut getListener()* faz com que ele tenha que ser implementado nos aspectos que os estendem (*FilterMonitoring e TestCaseMonitoring)* de acordo com as necessidades de cada um. O *getListener()* foi implementado em ambos os aspectos juntamente com os *advices* que os intercepta. Tais *advices* poderiam ser implementados na própria classe abstrata já que possuem o mesmo propósito (armazenar referência de *SelectionListeners*) independente do aspecto que a esteja estendendo, no entanto, caso ele e o atributo *selectionListener* fossem transportados ao *AbstractListener,* apenas seria possível acessar o atributo através de *inter type declarations* ou diretamente em uma instância de aspecto criada, o que poderia ser realizado por estruturas que criam instâncias de aspectos para execuções de *join points* como *perthis, pertarget, percflwow* etc. O uso de tais funcionalidades seria útil para aproveitar ainda mais vantagens no uso de herança evitando a replicação dos *advices* relativos aos *poinctus* abstratos. Além do quê, caso problemas de consistência nos valores dos aspectos ocorressem, o uso de construções de instâncias de aspectos seria a única maneira de evitá-los.

Contudo, tais funcionalidades ainda se encontram em processo de maturação em *Equinox Aspects* e ainda não podem ser utilizadas em sua plenitude. Apesar desses problemas, o modo como os aspectos foram implementados funcionam corretamente, pois ambos os atributos são instanciados apenas na interceptação da criação dos *SelectionListeners* que ocorre apenas uma vez. Não podendo, portanto, sofrer problemas de consistências provocados por modificações nos atributos por outros aspectos no decorrer do uso da aplicação. No próximo capítulo estes problemas serão discutidos mais detalhadamente.

6. Avaliação

Nesta seção serão avaliados os resultados alcançados com a implementação do monitoramento em aspectos em relação à abordagem em OO e também detalhados os problemas identificados durante a implementação do *plug-in* em *Equinox Aspects.*

6.1 Análise da Abordagem em OO

Como pôde ser visto no capítulo 3, a utilização da Programação Orientada a Objetos (POO), no desenvolvimento das estratégias de monitoramento, pode ser de certa forma simples, pois é necessária apenas a criação de uma classe de monitoramento para que os dados identificados no contexto de monitoramento sejam reportados a ela. No entanto, viu se que os códigos do monitoramento ficaram espalhados e entrelaçados ao código de vários *plug-ins* da TaRGeT.

Analisando as funcionalidades interceptadas em OO, encontra-se códigos do monitoramento espalhados (*scattared)* nas classes *NewProjectWizard, OpenProjectWizard* e *TargetProjectRefresher* do *plug-in* TaRGeT *Project Manager;* na classe *ExcelFileFormatter* do *plug-in* TaRGeT *Test Center* 3 *OutPut; WordDocumentProcessing* do *plug-in* TaRGeT *MSWord Input* e *OnTheFLyGeneratedTestCasePage* do *plug-in* TaRGeT *TC Generation Test.* Nesta última classe há ainda 4 ocorrências de chamadas a métodos de monitoramento (*tangling)*.

Dessa forma, o espalhamento e entrelaçamento dos interesses envolvidos, acabaram afetando 4 *plug-ins* da TaRGeT, o que poderia comprometer bastante o desenvolvimento da TaRGeT de maneira modular. Este problema ao longo do desenvolvimento e melhorias da aplicação pode comprometer ainda mais a modularidade em mais partes da aplicação, já que mais estratégias de monitoramento poderão ser identificadas.

6.2 Análise da Implementação em Aspectos

A implementação do *plug-in* em aspectos, notadamente *Equinox Aspects,* fez com que o desenvolvimento do *plug-in* fosse não invasivo. Todos os trechos de código que poderiam estar espalhados ou entrelaçados ao código da TaRGeT foram transportados aos aspectos. Os aspectos foram implementados em um *plug-in* totalmente independente que, por conta disso, pode ser facilmente removido ou adicionado novamente à TaRGeT, aproveitando a característica *plug and play* de aplicações *Eclipse RCP*.

No entanto, ainda existe forte dependência semântica entre o aspecto e o código interceptado. É possível, por exemplo, que o código escrito por um desenvolvedor seja interceptado por um aspecto de maneira inesperada, ou o inverso; ser desenvolvido um aspecto que intercepte um código indevidamente. Sendo assim o paralelismo no desenvolvimento do *plug-in* em aspectos e da aplicação pode causar certos problemas por conta da forte dependência existente entre eles.

6.3 Limitações do TaRGeT Monitoring causadas por Equinox Aspects

Apesar dos benefícios trazidos à TaRGeT pelo *framework Equinox Aspects*, possíveis avanços ou melhorias no TaRGeT *Monitoring Plug-In*, podem ser freados devido a algumas deficiências observadas em *Equinox Aspects.* A primeira delas já foi comentada na seção acima, que é relacionada ao mau funcionamento das construções de instâncias de aspectos. Se forem utilizadas construções de instância como *perthis*, *pertarget*, *percflow ou percflowbelow* em classes abstratas, possíveis *pointcuts* existentes nos aspectos que as implementam, não conseguem interceptar seus *join points.*

Outro problema identificado está relacionado ao uso de *Inter type declarations.* Caso seja criada uma *inter type* em um dado aspecto, o *framework* deixa o aspecto reconhecer a declaração como sendo a de uma *inter type*, no entanto ocorre erro de compilação (sem sugestões) quando a *inter type* é chamada, lida ou escrita dentro de um *advice.* Tal problema foi verificado tanto para *inter types* de métodos quanto de atributos.

Apesar de ter sido possível desenvolver os aspectos com tais problemas, algumas facilidades deixaram de ser aproveitadas. Uma delas foi a implementação de *advice* em classe abstrata. No caso, como não tinha maneira de salvar o valor da referência do *SelectionListener* na classe abstrata, este mesmo *advice* teve que ser replicado em ambos aspectos que herdaram de tal classe. Mesmo assim *pointcuts* abstratos e não abstratos podem ser herdados normalmente.

Logo, possíveis avanços do TaRGeT *Monitoring* estão bastante atrelados à evolução de *Equinox Aspects* para que a *AspectJ Language* possa ser usada com todas as suas características sem restrições. A tabela a seguir demonstra um pouco mais detalhadamente o desempenho das contruções de *aspectj* testadas em *Equinox Aspects* para este trabalho. A coluna 3 diz respeito a funcionalidades que não funcionaram em todas as condições em que foram testadas.

Nome da Construção	**Funcionou Perfeitamente**	**Funcionou Parcialmente**	**Não Funcionou**
execution(), call()	X		
this(), target(), args(),	X		
aftter(), before(), around()	X		
throwing()	X		
returning()	X		
Inter Types		X	
abstract	X		
perthis(), pertarget(), percflow(), percflowbelow()		X	
adviceexection()			X
annotations			X

Tabela 1 – Análise de desempenho das contruções de aspectos em Equinox Aspects

Porém, *Equinox Aspects* foi desenvolvido recentemente e ainda se encontra em processo de amadurecimento. Na lista de usuários *Equinox Development Mailing List* [18] ainda se encontram muitos desenvolvedores ainda não familiarizados com a ferramenta, no entanto as dúvidas surgidas são rapidamente respondidas e reportada ao *bugzilla.* O que de certa forma demonstra que *Equinox Aspects* ainda está em desenvolvimento e que possui grande potencial para agregar bastante valor à Programação Orientada a Aspectos.

7. Visualização dos dados resultantes do Monitoramento

Para visualizar os dados resultantes do monitoramento, foi desenvolvida uma ferramenta em .NET integrada ao banco de dados SQL *Server* 2008. Através dela é possível visualizar o inter-relacionamento dos dados registrados pelo monitoramento. Cada projeto possui uma ou mais execuções que por sua vez estão ligadas aos *test cases,* filtros e exceções geradas em cada execução. Cada *test case* ainda está ligado aos *use cases* utilizados na geração da *suite* de teste. As figuras abaixo demonstram rapidamente as visualizações desses dados na tela e um modelo básico das tabelas sql implementadas, são detalhadas as execuções por projeto e os *test cases,* filtros, exceções e casos de uso por *test cases*.

A Figura 10 foca no detalhe dos filtros gerados na execução do projeto denominado *New Project* executado pelo usuário jp. Os registros dos filtros na aba *Filters* mostra respectivamente que o usuário primeiramente os gerou depois salvou e carregou.

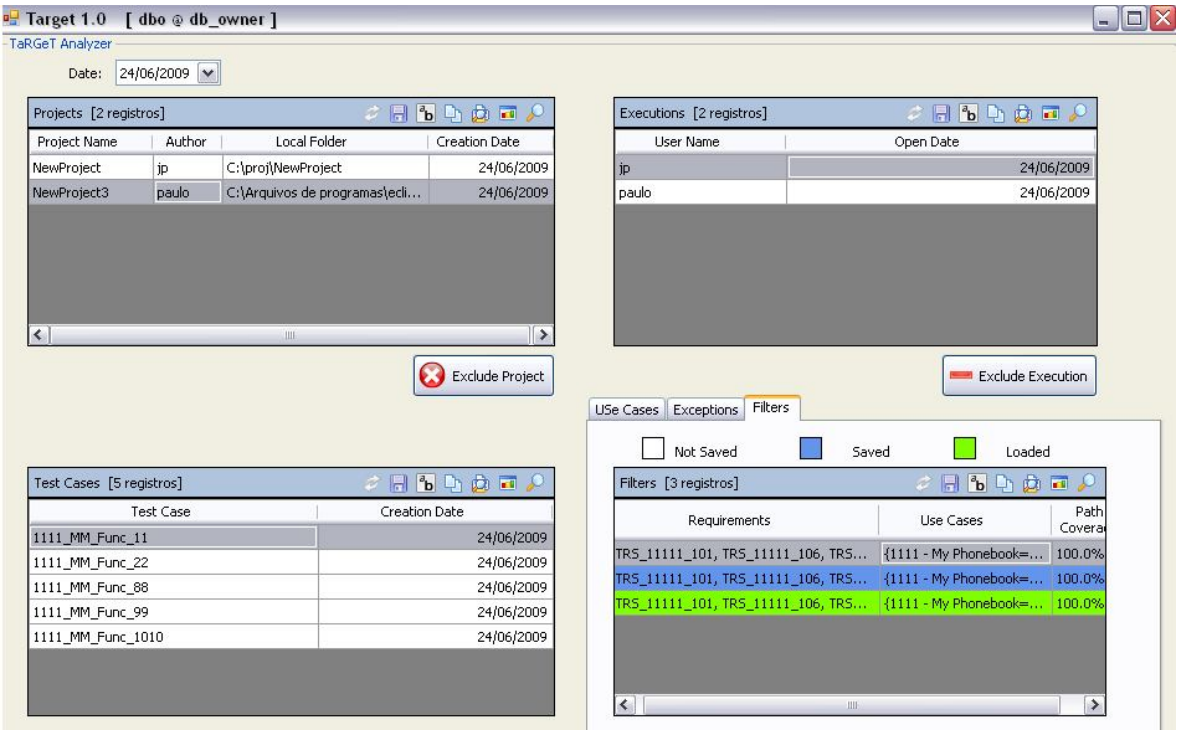


Figura 10 – Dados de projetos, test cases e filtros utilizados (geração, salvamento e carregamento)

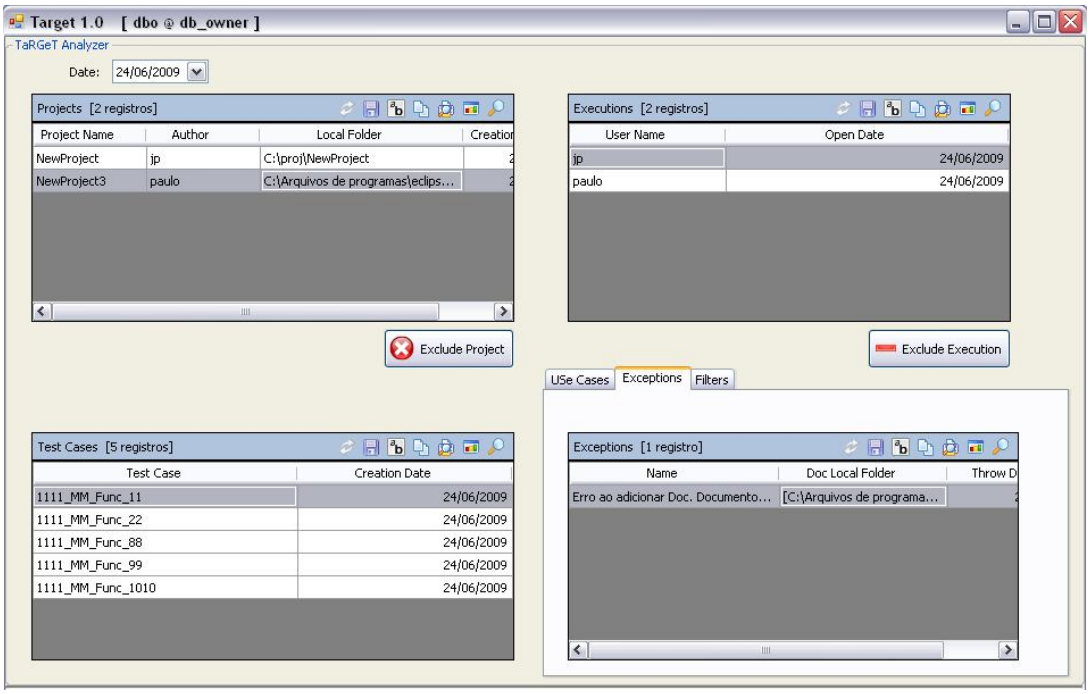


Figura 11 – Dados de projetos, *test cases* e exceções levantadas

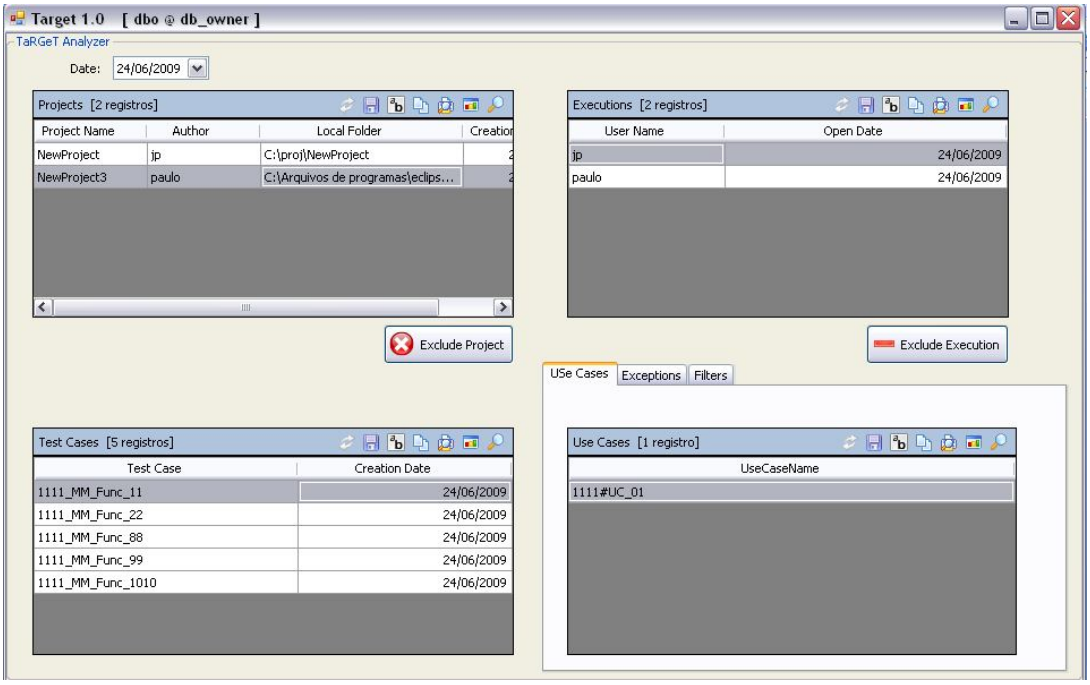


Figura 12 – Dados de projetos e use cases por test cases

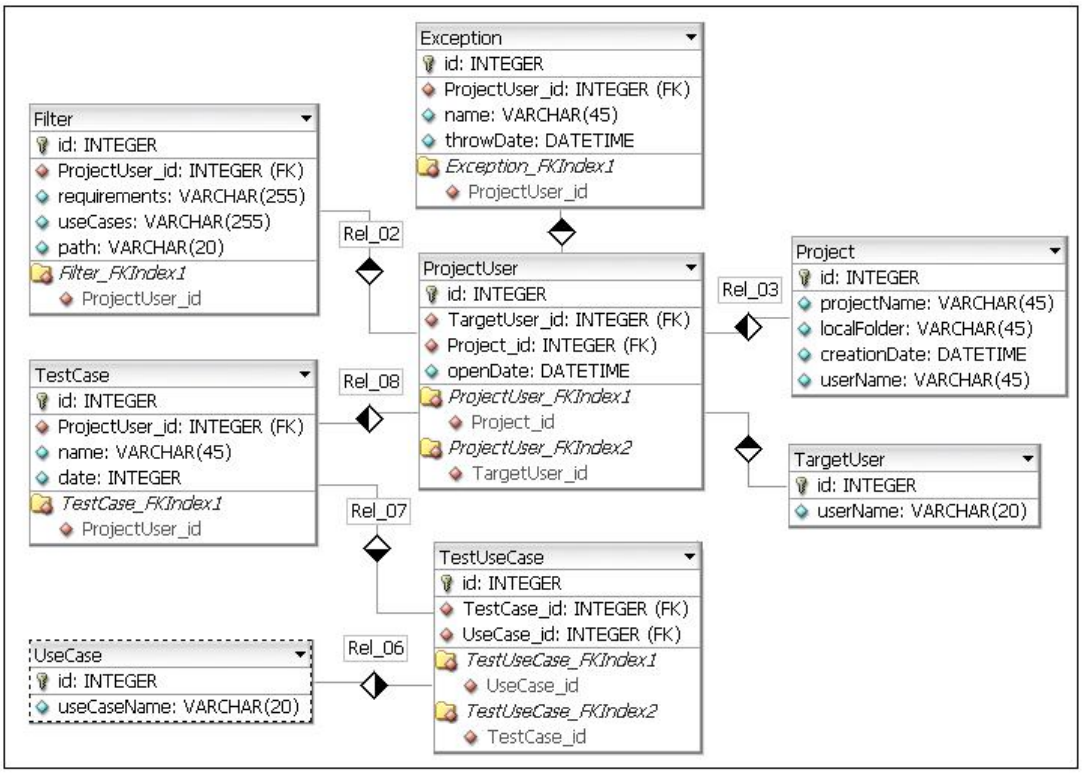


Figura 13 – Tabelas SQL utilizadas no SQL *Server* 2008

8. Conclusões

Este trabalho propôs a implementação de uma ferramenta capaz de realizar monitoramento de uma aplicação particular com o uso de aspectos na plataforma *Equinox Aspects.* Contudo, foi necessário contextualizar os motivos que de fato estimularam a sua implementação em aspectos. Esta contextualização levou em consideração a identificação de necessidades de monitoramento na TaRGeT e problemas de modularidade que poderiam ocorrer na implementação em orientação em objetos que poderiam comprometer o desenvolvimento na TaRGeT futuramente.

O desenvolvimento do TaRGeT *Monitoring Plug-In* se deu a partir da maturação deste contexto que norteou o desenvolvimento do *plug-in.* O objetivo de seu desenvolvimento mais do que trazer benefícios relacionados ao monitoramento, era mostrar que era possível anular possíveis problemas de modularidade com o advento da Programação Orientada a Aspectos na plataforma *Equinox Aspects*. Por fim, através de avaliações pôde-se ver que a utilização de aspectos de fato trouxe grandes vantagens em relação à abordagens em OO que poderiam ser limitados apenas pelos deficiências de *Equinox Aspects.*

8.1 Trabalhos Futuros

Possíveis trabalhos futuros certamente seriam a identificação de novas necessidades de monitoramento a serem interceptadas pelo *plug-in*, como monitoramento de mais exceções, funcionalidades relacionadas a outros componentes ainda não explorados. O que poderia ser ainda mais enriquecedor futuramente quando mais avanços no *Equinox Aspects* poderão ser observados. Problemas causados pela forte dependência entre o código e o aspecto poderiam ser solucionados através do uso de *Design Rules* [16] [20] que são regras de objeto a serem obedecidas em todo o processo de desenvolvimento das aplicações. Ou seja, o desenvolvimento da aplicação deverá ser baseado nas regras garantindo o estabelecimento dos requisitos mínimos necessários para o desenvolvimento paralelo dos componentes.

Existem também novidades em aspectos que poderiam agregar muito valor, uma delas seria o uso de *trace matching* [19] que permite facilmente a interceptação de caminhos de fluxos percorridos pela aplicação. Isso poderia ser usado, por exemplo, para interceptar fluxos que não devem ser percorridos pela aplicação. No entanto, qualquer tipo de novas soluções em aspectos a serem assimiladas ao TaRGeT *Monitoring*, terão que ser suportadas primeiramente por *Equinox Aspects.*

Apêndice A – Configuração de *Equinox Aspects* no Eclipse

Para utilizar *Equinox Aspects* deve ser baixado um arquivo com um conjunto de *plug-ins* em [13], posteriormente ele deve ser descompactado na pasta *plug-ins* do *Eclipse,* no entanto o mais indicado é fazer a atualização pelo próprio *update* do *Eclipse*. Na atualização do AJDT muito provavelmente estarão inclusos os *plug-ins* necessários ao *Equinox Aspects.* Após a instalação do *plug-in*, é necessário adaptar a aplicação *RCP.* Para isso, devem ser habilitados no *Eclipse launch configurations (*Figura 14), os *plug-ins org.eclipse.equinox.weaving.aspectj, org.eclipse.equinox.common, org.eclipse.update.configurator* e *plug-ins* requeridos por eles que podem ser obtidos na opção *Add Required Plug-Ins*.

Caso a versão do *Eclipse* que esteja sendo utilizada seja a *GanyMede,* será necessária a criação de um arquivo *config.ini* com uma lista de *plug-ins* nativos e da própria aplicação a serem inicializados no momento em que a aplicação for executada. Depois de criado, o *config.ini* deve ser selecionado na aba *configuration* do *Eclipse launch configurations.* Segue abaixo um exemplo do arquivo *config.ini* utilizado para a configuração do TaRGeT *Monitoring Plug-In.*

```
1  # default start level for the bundles
2  osgi.bundles.defaultStartLevel=4
3
4  osgi.bundles=org.eclipse.update.configurator@3\:start,
5  org.eclipse.core.runtime@start,org.eclipse.equinox.common@2\:start,
6  org.eclipse.equinox.weaving.aspectj@4\:start,
7  org.eclipse.equinox.weaving.caching.j9@start,
8  org.eclipse.equinox.weaving.caching@2\:start,
9  org.aspectj.runtime,org.aspectj.weaver,
10 org.eclipse.core.commands,org.eclipse.core.contenttype,
11 org.eclipse.core.databinding,org.eclipse.core.expressions,
12 org.eclipse.core.jobs,org.eclipse.core.runtime.compatibility.auth,
13 org.eclipse.equinox.app,org.eclipse.equinox.event,
14 org.eclipse.equinox.preferences,org.eclipse.equinox.registry,
15 org.eclipse.equinox.weaving.aspectj,org.eclipse.equinox.weaving.caching,
16 org.eclipse.equinox.weaving.caching.j9,org.eclipse.equinox.weaving.hook,
17 org.eclipse.help,org.eclipse.jface,org.eclipse.jface.databinding,
18 org.eclipse.osgi.services,org.eclipse.swt,
19 org.eclipse.swt.win32.win32.x86,org.eclipse.ui,
20 org.eclipse.ui.workbench,org.junit,org.eclipse.equinox.launcher,
21 org.eclipse.equinox.launcher.win32.win32.x86
22
23
24
25 # AOSGi
26 osgi.framework.extensions=org.eclipse.equinox.weaving.hook
27 aj.weaving.verbose=true
28 org.aspectj.weaver.showWeaveInfo=true
29 org.aspectj.osgi.debug=true
30
```

Listagem 18 – Arquivo *config.ini* utilizado para a execução do TaRGeT *Monitoring Plug-In*

É importante também a escolha dos níveis (*levels)* logo após o nome do identificador do *plug-in*. O *plug-in (bundle) org.eclise.equinox.weaving.aspectj,* por exemplo, é recomendado a ser executado no nível 4 como pode ser visto na Listagem 18. O mesmo serve para demais *plug-ins.* Os *plug-ins* contidos no arquivo *config.ini* são de interesse da aplicação, dessa forma ele não possui conteúdo fixo que pode ser diversificado de aplicação para aplicação.

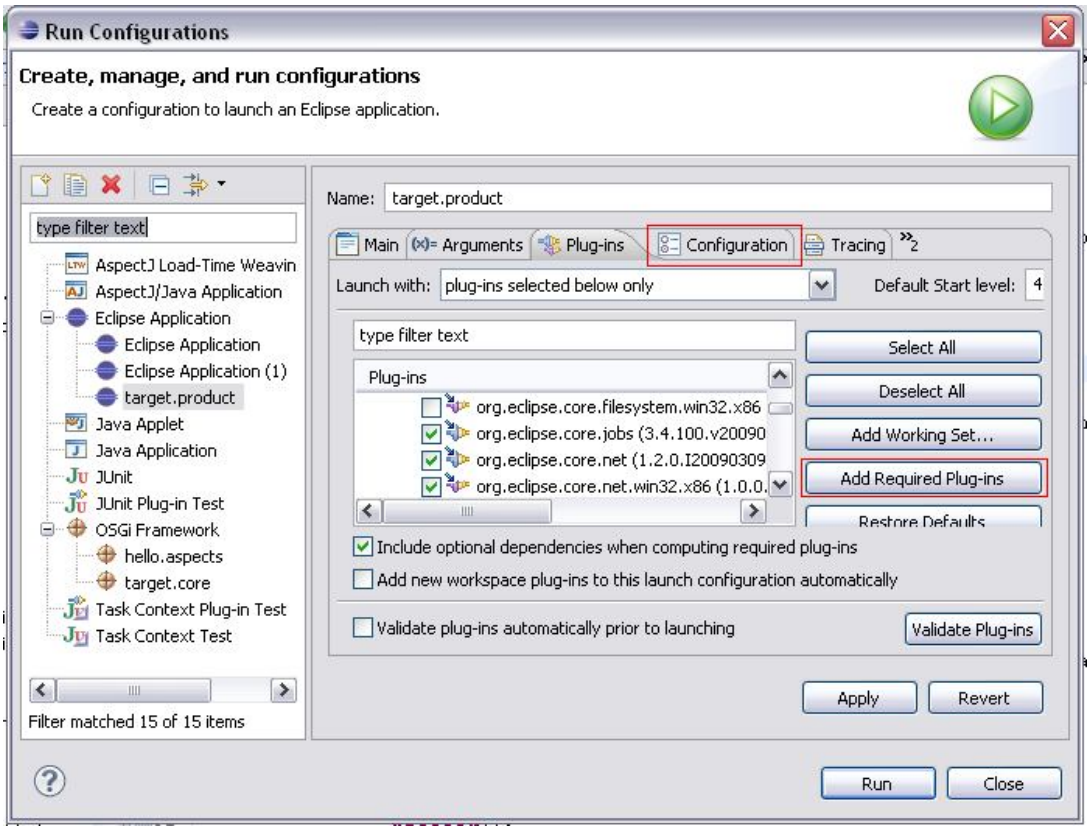


Figura 14 – Eclipse Launch Configurations

Bibliografia

[1] Polozoff, A. *Proactive Application Monitoring.* IBM Software Group, Software Services for WebSphere, Chicago, Illinois, Abril 2003. Disponível em: <http://www.ibm.com/developerworks/websphere/library/techarticles/0304_polozoff/polozoff.html>. Acesso em: 1 maio. 2009.

[2] Lesiecki, N. *Improve modularity with aspect-oriented programming.* Technical Team Lead, eBlox, Inc. Janeiro 2002. Disponível em: <http://www.ibm.com/developerworks/java/library/j-aspectj/>. Acesso em: 1 maio. 2009.

[3] Lars, V. *Eclipse RCP - Tutorial with Eclipse 3.4.* Copyright © 2008-2009 Lars Vogel. Abril 2009. Disponível em:< http://www.vogella.de/articles/RichClientPlatform/article.html>. Acesso em: 1 maio. 2009.

[4] Copyright © 2009 The Eclipse Foundation. All Rights Reserved. *Equinox.* Disponível em: <http:/www.eclipse.org/equinox/>. Acesso em: 1 maio. 2009.

[5] Chapman, M.; Vasseur, A.; Kniesel, G. *Industry Track Proceeding.* AOSD 2006 – 5th International Conference on Aspect-Oriented Software Development. Institut für Informatik III Universität Bonn. Bonn, Germany. 20-24 Março. 2006.

[6] Bilenium. *Performance de Aplicações.* Disponível em: < http://www.bilenium.com.br/?q=node/21> . Acesso em: 11 Junho. 2009.

[7] Lars, V. *Extensions and Extension Points Tutorial.* Copyright © 2008-2009 Lars Vogel. Abril 2009. Disponível em: < http://www.vogella.de/articles/EclipseExtensionPoint/article.html >.

[8] CIn - BTC Research Project. *TaRGeT 4.0, a brief tutorial.* Fevereiro. 2009.

[9] Nogueira, S.; Jucá, M. *Analysing the use of TaRGeT templates.* Novembro. 2008.

[10] Copyright © 2009 The Eclipse Foundation. *JDT Weaving Features.* Disponível em < http://wiki.eclipse.org/JDT_weaving_features >

[11] Copyright © 2009 The Eclipse Foundation. *AJDT AspectJ Development Tools.* Disponível em < http://www.eclipse.org/ajdt/>

[12] Lippert, M. *What's new in Equinox Aspects.*

[13] Copyright © 2009 The Eclipse Foundation. *Equinox Aspects.* Disponível em <http://www.eclipse.org/equinox/incubator/aspects/index.php>

[14] Copyright © 2009 The Eclipse Foundation. *Explore The Eclipse Membership.* Disponível em: < http://www.eclipse.org/membership/ >

[15] AspectJ Team. *The AspectJ Project.* Abril. 2008. Disponível em: < http://www.eclipse.org/aspectj >

[16] Arcoverde, R. *Implementação de uma linguagem de Especificação de Design Rules para projetos Orientados a Aspectos.* Trabalho de Graduação. Junho. 2008.

[17] Resende, P. M. A.; Silva, C. C. *Programação Orientada a Aspectos em Java.* Disponível em: < http://books.google.com.br/books?id=WTjfKBmlxPMC&dq=after+throwing+aspectos&source=gbs_navlinks_s >

[18] Copyright © 2009 The Eclipse Foundation. *Equinox Development Mailing List.* Disponível em: https://dev.eclipse.org/mailman/listinfo/equinox-dev

[19] Avgustinov, P.; Allan, C.; *Adding Trace Matching with Free Variables to AspectJ.* University of Oxford.

[20] C. Baldwin. e K. Clark. BB. *Design Rules, Vol. 1: The Power of Modularity*. The MIT Press, 2000.

Folha de Aprovação

Assinatura do Orientador