UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

IMPLEMENTAÇÃO E ANÁLISE DE UMA LINHA DE PRODUTOS DE SOFTWARE

Trabalho de Graduação

FELYPE SANTIAGO FERREIRA

Orientador: Paulo Henrique Monteiro Borba

Recife, 2009

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

IMPLEMENTAÇÃO E ANÁLISE DE UMA LINHA DE PRODUTOS DE SOFTWARE

Trabalho de Graduação

FELYPE SANTIAGO FERREIRA

Projeto de Graduação apresentado no Centro de Informática da Universidade Federal de Pernambuco por Felype Santiago Ferreira, orientado pelo PhD. Paulo Henrique Monteiro Borba, como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Paulo Henrique Monteiro Borba

Recife, 2009

"O segredo do êxito na vida do homem consiste em estar disposto a aproveitar a ocasião que se lhe depare."
Benjamin Disraeli

AGRADECIMENTOS

Nesses últimos quatro anos, pessoas extraordinárias entraram na minha vida enquanto outras permaneceram ao meu lado. Colegas com quem tive o prazer de estudar e trabalhar e que me ensinaram bem mais do que eu esperava aprender durante esse período que se passou tão depressa.

Agradeço primeiramente a minha mãe por toda a dedicação, atenção e paciência que fizeram de mim o que sou hoje e ao meu pai e a minha irmã, por todo o apoio que obtive.

Ao meu grande amigo, Pedro, com quem fiz todos os trabalhos, e que sempre esteve disposto a ajudar com inteligência, paciência e humildade, não somente nos projetos acadêmicos. A Marcos e Léo, pela parceria nos estudos e projetos, pelas piadas e brincadeiras mesmo durante os projetos mais entediantes, e pela amizade construída durante esses anos.

A todos os colegas de graduação com quem estudei junto tantas horas para provas e desenvolvi inúmeros projetos ao longo da graduação, e que me ajudaram a desenvolver a capacidade de trabalho em equipe, muitas vezes tão difícil.

Aos meus amigos, em especial, Janaína, Aristeu, Cecília e Fernanda, por permanecerem ao meu lado, mesmo os que estão longe, e por entenderem a minha freqüente ausência como conseqüência da dedicação a tantos projetos e provas no CIn.

Por fim, a Laís e Pedro (de novo!), pela colaboração na implementação deste trabalho, e ao meu orientador, Paulo Borba, pelo suporte fornecido para o sucesso deste projeto.

RESUMO

O uso de técnicas de desenvolvimento de Linhas de Produto de software (LPS) permite a construção de aplicações de forma mais eficiente através do reuso estratégico de módulos comuns a produtos similares. A fim de atender a requisitos particulares de clientes, o uso de estratégias de LPS para a implementação de variações mantendo um baixo custo de desenvolvimento e manutenção, apresenta-se como um solução vantajosa. Este trabalho descreve a implementação da linha de produtos de uma ferramenta de geração automática de casos de teste e apresenta uma avaliação das soluções alternativas para a implementação de variações.

Palavras-chave: TaRGeT, Linha de produtos de software, Eclipse RCP.

ABSTRACT

The usage of software product lines (SPL) techniques allows the build of applications on a more efficient way with the strategical reuse of common modules to similar products. In order to satisfy particular requirements of clients, the usage of LPS strategies on the implementation of variations, keeping a low cost of development and maintenance, is an effective solution. This work describes the implementation of the product line of an automatic generating tool to test cases and presents an evaluation of alternative solutions to implement the variations.

Key-words: TaRGeT, Software Product Line, Eclipse RCP.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE LISTAGENS

LISTA DE TABELAS

1. INTRODUÇÃO

A abordagem de Linhas de Produtos de Software (LPS) permite a construção de conjuntos de aplicações similares de forma mais eficiente, por meio do reuso estratégico de módulos da aplicação [6].

Além disso, a adoção de estratégias de desenvolvimento baseadas em LPS trazem vantagens como a redução do *time-to-market* e dos custos de desenvolvimento, além de uma melhor qualidade dos produtos concebidos resultando em um aumento da satisfação dos clientes [7].

Em situações em que uma mesma ferramenta é utilizada por vários clientes, pequenas modificações podem ser necessárias a fim de atendê-los adequadamente. É neste cenário que se encontra a TaRGeT (*Test and Requirements Generation Tool*) [8], uma ferramenta de geração automática de casos de teste, que pode ser utilizada para testes de software, celulares ou qualquer outro produto que tenha um documento de casos de uso especificado.

A ferramenta foi inicialmente desenvolvida pelo time de pesquisa do Brazil Test Center (BTC) - Motorola [9]. Atualmente é utilizada por mais de um time do centro, além de ser também utilizada por clientes externos, como a Qualiti [10] e o projeto INES [11]. Após a identificação de requisitos particulares de clientes específicos da ferramenta, foi percebida a necessidade de aplicar estratégias de linhas de produto de software ao desenvolvimento da aplicação.

Foram identificados requisitos particulares como, por exemplo, o formato de saída dos casos de teste da ferramenta. Alguns times do BTC utilizam um formato específico nomeado Test Central 3 (TC3), enquanto outros utilizam uma evolução do TC3, o Test Central 4. Já os clientes externos utilizam um formato distinto dos utilizados no BTC, o formato XML TestLink [12].

Requisitos como esses fazem com que seja necessário adicionar, alterar ou remover funcionalidades da TaRGeT e, para que os clientes tenham seus requisitos atendidos de forma satisfatória, sejam concebidas ferramentas distintas, ainda que com grande similaridade entre elas.

Para que aplicações customizadas sejam implementadas, técnicas simples, como o uso de um controle de versão com customizações implementadas em *branches* poderiam ser utilizadas. Entretanto os custos de manutenção do software seriam elevados, uma vez que, para cada modificação em artefatos comuns às variações armazenadas em *branches*

distintas, seriam necessárias alterações em todas as réplicas do artefato. Todos os artefatos em comum às variações estariam replicados.

A fim de manter um baixo custo de desenvolvimento e manutenção das variações da ferramenta, evitando replicações de código, técnicas de LPS são aplicadas na implementação dos requisitos particulares da TaRGeT.

Para a avaliação da solução, o trabalho traz ainda uma análise das soluções alternativas para os pontos de variação em que mais de uma técnica mostrou-se vantajosa para a implementação levantando as vantagens e desvantagens de cada uma delas.

1.1 Objetivos

O objetivo deste trabalho é, fazendo uso de estratégias de desenvolvimento de linhas de produtos de software, apresentar a implementação e a análise da LPS da TaRGeT. Ao longo do trabalho serão apresentadas as técnicas utilizadas para o desenvolvimento e análise da solução.

As principais contribuições deste trabalho são:

- Identificação de pontos de variação da LPS;
- Implementação de variações das features da LPS;
- Análise de soluções alternativas para implementação da LPS.

1.2 Estrutura do trabalho

Este trabalho é composto por 7 capítulos. No Capítulo 2, são abordados conceitos importantes sobre Eclipse RCP para o entendimento da implementação da TaRGeT e posteriormente de suas variações.

Após a visão geral do *framework* utilizado na implementação da ferramenta, o *core* da TaRGeT é detalhado no Capítulo 3. Serão descritas as principais responsabilidades dos *plugins* que compõem o *core* da aplicação e as dependências existentes entre eles.

No Capítulo 4 são descritas as técnicas utilizadas no desenvolvimento da linha de produtos da ferramenta e no Capítulo 5 é apresentada a implementação da LPS da TaRGeT, detalhando os pontos de variação identificados, features extraídas, variações implementadas e técnicas utilizadas.

No Capítulo 6 são analisadas, em detalhes, as soluções alternativas para a implementação das variações e pontos de variação em que mais de uma solução se mostrou interessante para o desenvolvimento da LPS.

Ao final, são feitas considerações finais acerca do trabalho.

2. APLICAÇÕES ECLIPSE RCP

Eclipse RCP é uma plataforma para construção e desenvolvimento de *Rich Client Applications* [5]. Ela permite que desenvolvedores utilizem a arquitetura do Eclipse para desenvolver aplicações *stand-alone* reutilizando funcionalidades e padrões de código do Eclipse [14].

Os próximos tópicos apresentam alguns conceitos básicos do Eclipse RCP, necessários para o entendimento deste trabalho.

2.1 Arquitetura do Eclipse RCP

A plataforma Eclipse RCP é baseada na arquitetura de *plugins*. Com isso, uma aplicação RCP é composta por um conjunto de *plugins* que são dependentes uns dos outros [14].

Um *plugin* do Eclipse é um componente que provê algum tipo de serviço no contexto do Eclipse Workbench, melhor detalhado nos próximos tópicos. Cada *plugin* é descrito em um arquivo XML nomeado plugin.xml, presente na pasta do próprio *plugin*. Esse arquivo descreve o que é necessário para ativar o *plugin*.

A especificação do *plugin* contida no arquivo é armazenada na memória em um repositório denominado Plugin Registry. O Eclipse constrói uma instância de cada *plugin* utilizando o registro de cada um contido no repositório. A listagem a seguir apresenta um exemplo do arquivo de configuração plugin.xml [25].

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin
  name="JUnit Testing Framework"
  id="org.junit"
  version="3.7"
  provider-name="Eclipse.org">
  <runtime>
    <library name="junit.jar">
      <export name="*"/>
    </library>
  </runtime>
</plugin>
```

Listagem 2.1 - Exemplo de arquivo plugin.xml [25].

Cada plugin pode definir *extension points* que definem possibilidades de contribuição por outros *plugins*. Um *plugin* pode utilizar as *extensions* de outros para prover suas funcionalidades [14]. A figura 2.2 apresenta a organização dos componentes do Eclipse RCP. Os principais itens são detalhados a seguir [15]:

- Eclipse Runtime: Concede suporte aos *plugins*, *extension points* e *extensions*;
- SWT e JFace: O Standard Widget Toolkit provê acesso a interface do usuário, utilizando elementos da GUI do sistema operacional em que a aplicação foi implementada. A figura a seguir mostra uma mesma interface, implementada com SWT, executada em diferentes sistemas operacionais. O JFace trabalha sobre o SWT, manuseando itens da interface do usuário;

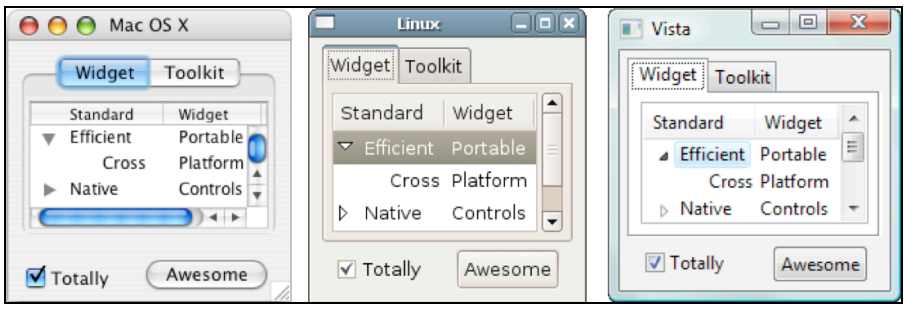


Figura 2.1 - Interface multi-plataforma desenvolvida com SWT.

- Workbench: É construído sobre o Runtime, SWT e JFace e provê o gerenciamento de perspectivas, *views*, editores, *wizards*, entre outros.

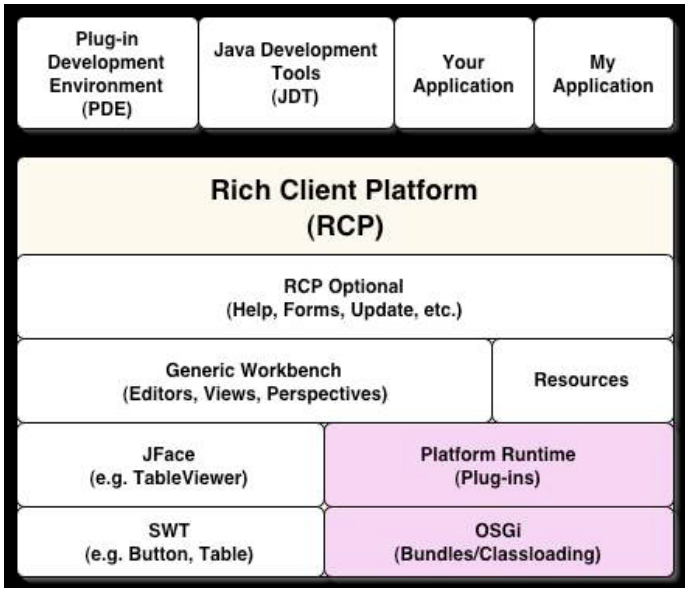


Figura 2.2 - Organização dos componentes do Eclipse RCP.

2.2 Principais componentes de uma aplicação RCP

Os itens a seguir compõem o conjunto básico de elementos necessários em uma aplicação Eclipse RCP [14]:

- Programa principal: A classe principal de uma aplicação Eclipse RCP deve implementar a interface IApplication, estendendo o *extension point* org.eclipse.core.runtime.application;
- Perspectiva: A aplicação deve conter, no mínimo, uma perspectiva que estenda o *extension point* org.eclipse.ui.perspective;
- *Workbench Advisor*: Componente invisível que controla a aparência da aplicação (menus, barras de ferramenta, perspectivas, etc.).

Todos os *extension points* implementados pela aplicação e dependências de *plugins* devem ser listados no arquivo de configuração plugin.xml. O conjunto mínimo necessário para criar e executar uma aplicação RCP são os *plugins* "org.eclipse.core.runtime" e "org.eclipse.ui".

Para que um programa desenvolvido com Eclipse RCP seja executado com um conjunto de artefatos necessários para sua execução e que o caracterizem, como ícones, *splash*, *jars*, entre outros, pode ser definido um arquivo de configuração <nome do produto>.product. Este arquivo encapsula todas as configurações necessárias para a correta execução da aplicação e, através dele, pode ser gerada uma aplicação executável do projeto.

2.3 Eclipse Workbench

O Workbench do Eclipse coordena e apresenta elementos do JFace. Ele define um poderoso paradigma de interface do usuário com janelas, perspectivas, *views*, editores e ações, organizando-os em *layouts* particulares [16].

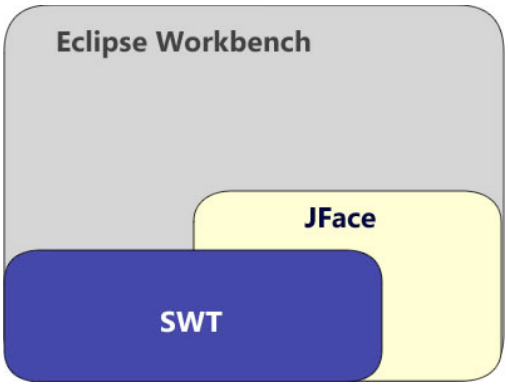


Figura 2.3 - Eclipse Workbench [17].

Para o usuário, o Workbench se apresenta como uma coleção de janelas. Em cada janela ele permite que o usuário organize os componentes como se estivesse organizando sua mesa de trabalho.

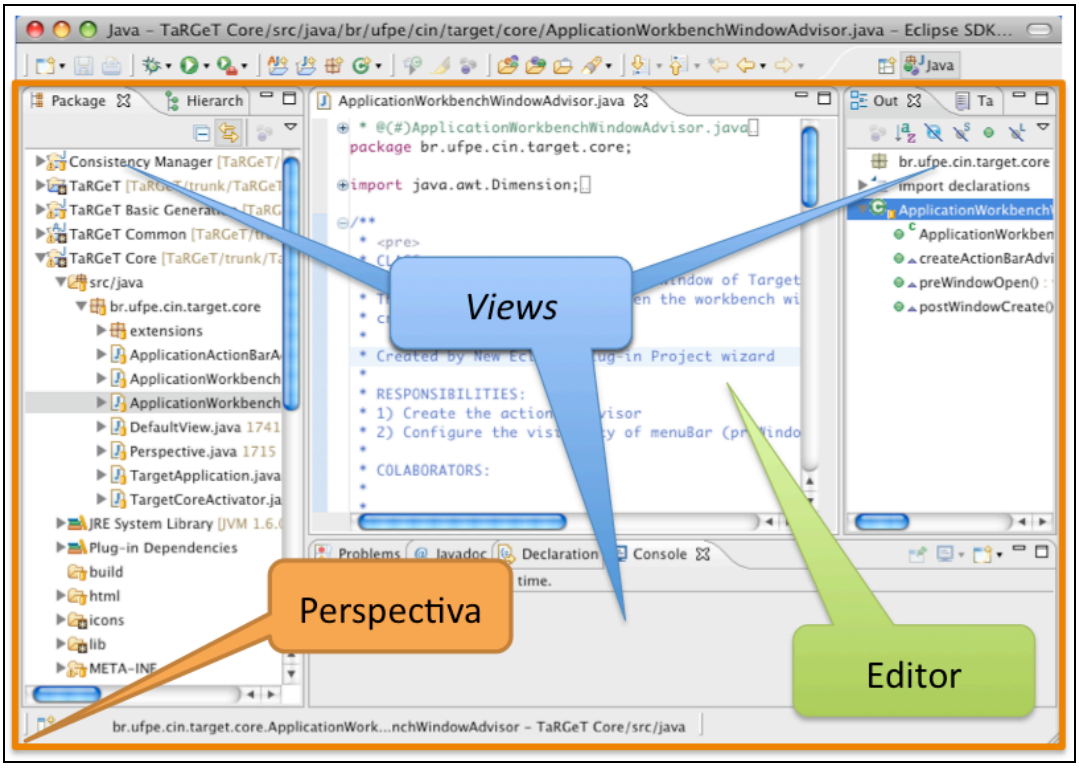


Figura 2.4 - Elementos do Workbench.

A figura acima apresenta os componentes do Workbench do Eclipse. Uma perspectiva é um container para um conjunto de *views* e editores. Os editores focam a função principal da aplicação e são compartilhados por múltiplas perspectivas em uma mesma janela, enquanto as *views* provêem suporte a uma dada tarefa, podem ser mostradas sem título e até mesmo extraídas da janela do Workbench [16].

3. A TARGET

A TaRGeT [8] (*Test and Requirements Generation Tool*) é uma ferramenta de geração de casos de teste que tem por objetivo automatizar uma abordagem sistemática para lidar com os artefatos de requisitos e de testes de uma forma integrada. A ferramenta gera automaticamente casos de teste a partir de cenários de casos de uso escritos em linguagem natural em documentos que devem seguir *templates* especificados nos formatos Microsoft Word (.doc), Microsoft Excel (.xls) ou XML (.xml). Os *templates* contêm as informações necessárias para que os casos de teste possam ser gerados automaticamente.

Essas informações incluem identificação, nome, breve descrição, fluxo principal e fluxos de exceção do caso de uso. O fluxo principal contém uma seqüência de passos e um identificador para o passo inicial e para o passo final da seqüência que o compõe. Cada passo contém um identificador, uma ação do usuário e um estado e resposta do sistema.

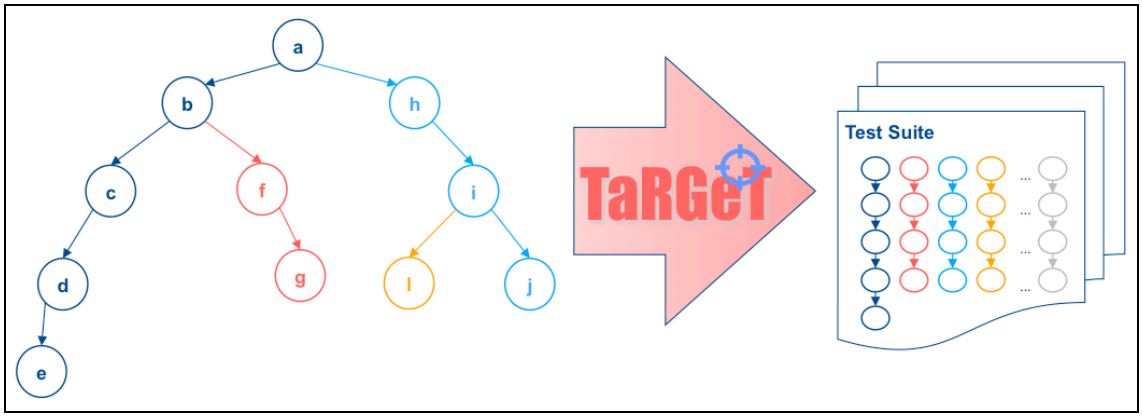


Figura 3.1 - Geração de suíte de testes a partir de fluxos de caso de uso [24].

A partir dos casos de uso do documento de entrada, o algoritmo de geração de casos de teste da TaRGeT gera testes para cobrir todos os cenários descritos nos casos de uso. Para a geração da suíte, podem ser selecionados testes gerados a partir de requisitos específicos, casos de uso específicos, propósito dos testes ou ainda grau de similaridade [24].

A ferramenta pode ser utilizada para geração de casos de teste de aplicações para qualquer domínio, desde de que seja fornecido um documento de casos de uso em um dos formatos suportados pela TaRGeT.

As figuras a seguir apresentam exemplos de casos de uso e de teste dos documentos de entrada e saída da ferramenta.

Feature [illegible]

UC 01 – [illegible]

Description

[illegible]

Main Flow

Description: [illegible]
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1M	[illegible]	[illegible]	[illegible]
2M	[illegible]		[illegible]
3M	[illegible]		[illegible]
4M	[illegible]	[illegible]	[illegible]

Exception Flows

Description: [illegible]
From Step: 3M
To Step: END

Step Id	User Action	System State	System Response
1A	[illegible]	[illegible]	[illegible]

Figura 3.2 - Exemplo de documento de entrada (formato Microsoft Word) [24].

Case	Reg. Leve	Exe. Type	Case Description	Procedure	Expected Results
1					
			Use Cases:		
			Requirements:		
			Setup:		
			Initial Conditions:		
			Notes:		
			Test Procedure (Step Number):		
			1		
			2		
			3		
			4		
			Final Conditions:		
			Cleanup:		

Figura 3.3 - Exemplo de documento de saída (Test Central 3) [24].

3.1 Estrutura da aplicação

As principais funcionalidades da TaRGeT encontram-se distribuídas em quatro *plugins* distintos, cada um com responsabilidades bem definidas. A figura a seguir ilustra a organização das dependências entre os *plugins* básicos da TaRGeT e os resultantes da implementação da LPS. Os componentes do *core* da ferramenta serão discutidos em mais detalhes nos próximos tópicos deste capítulo.

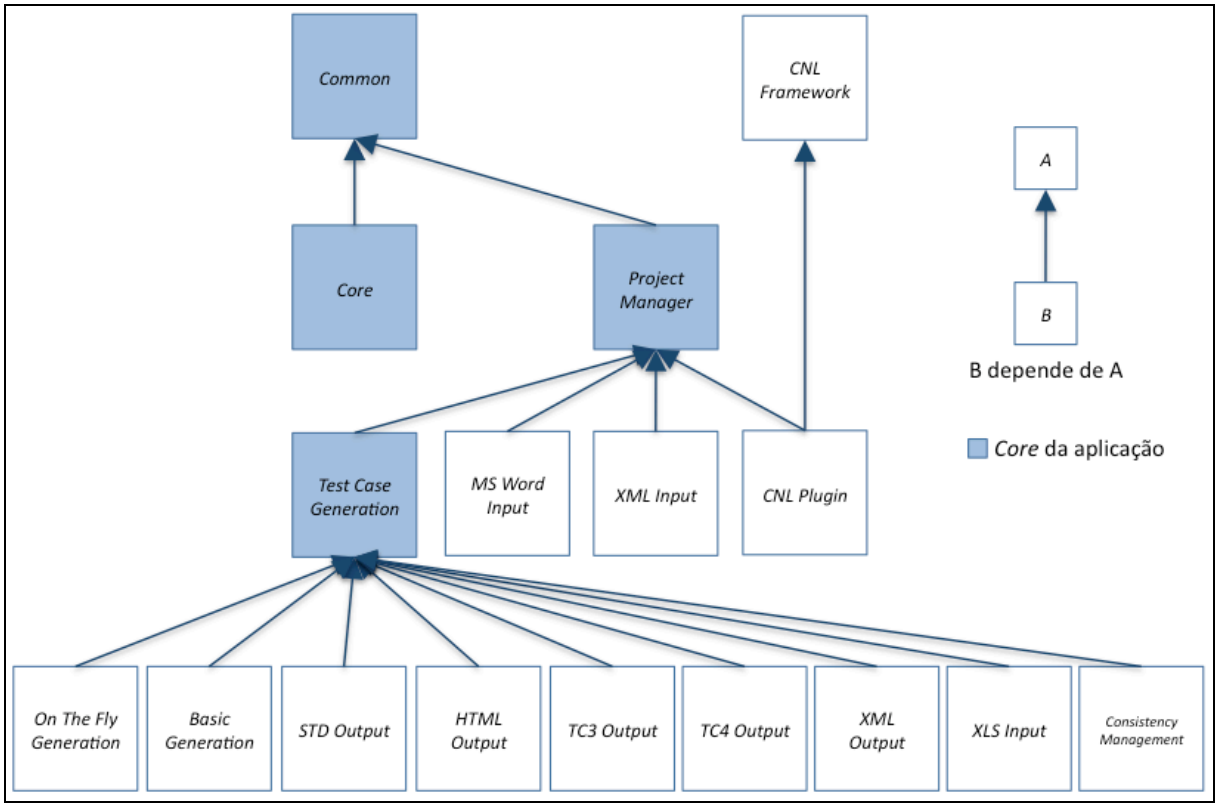


Figura 3.4 - Dependências entre *plugins* da TaRGeT.

3.1.1 TaRGeT Core

O *plugin* Core é responsável pela inicialização da ferramenta e configuração básica da janela, barra de título, *view* e perspectiva padrão da aplicação RCP.

3.1.2 TaRGeT Common

O módulo Common reúne as classes básicas necessárias para a implementação dos serviços da aplicação. Os seguintes itens podem ser encontrados neste módulo:

- Classes necessárias para a representação de documentos de caso de uso na ferramenta;
- Classes necessárias para a representação dos casos de teste gerados pela TaRGeT na aplicação;
- Classes de exceção e ação básicas que devem ser estendidas pelas demais classes de tratamento de erros e ações da TaRGeT;
- *Parser* para o XML especificado para a descrição dos documentos de casos de uso;
- Implementação do algoritmo de geração de casos de teste;
- Classes com funcionalidades básicas necessárias em vários módulos da aplicação como filtros de extensão de arquivos, gerenciadores de arquivos de propriedades e funções básicas para manipulação de arquivos, casos de uso e strings.

3.1.3 TaRGeT Project Manager

O *plugin* Project Manager é responsável pelo controle e gerenciamento da aplicação e inclui as regras de negócio para as ações do sistema. Neste módulo podem ser encontrados:

- Ações para o gerenciamento de projetos da TaRGeT como criação, abertura, atualização, busca e importação de documentos de casos de uso com os seus respectivos controladores responsáveis pela efetivação da ação. Todas as ações estendem a classe TargetAction presente no módulo TaRGeT Common, que é uma das dependências do plugin Project Manager;
- Implementações básicas de TableView, TreeView, View, TreeObject e TreeViewContentProvider, que são componentes de GUI do Eclipse RCP. Esses itens são utilizados como base para implementação dos componentes de GUI da TaRGeT;
- Implementação de *views* e editores para visualização de casos de uso e casos de teste e *wizards* para manipulação de projetos;

- Implementação de busca para documentos de casos de uso;
- Exceções para comunicar possíveis erros encontrados no formato de documentos de casos de uso e nas operações dos controladores;
- *Extension points* para implementação de formatos novos de entrada para documentos de casos de uso.

3.1.4 TaRGeT Test Case Generation GUI

O *plugin* Test Case Generation é responsável pela geração das suítes de testes com seus conjuntos de casos de teste. Nele estão implementados:

- *Extension points* para as *features* de Output, Import Template e Test Suite Extractor;
- Controladores para geração de casos de teste;
- Filtros para seleção de casos de teste;
- Exceções para tratamento de possíveis erros na avaliação da similaridade de casos de teste e na geração deles;
- *Wizards* para seleção e visualização de casos de teste.

4. LINHAS DE PRODUTO DE SOFTWARE

Uma linha de produtos de software é um conjunto de sistemas de software que compartilham um conjunto comum de features gerenciadas, que satisfazem necessidades específicas de um segmento particular do mercado ou missão e que são desenvolvidos a partir de um conjunto comum de artefatos de forma pré-estabelecida [1][2].

Dentre as estratégias para implementação de LPS, podemos citar a abordagem extrativa, que consiste de *bootstrapping* de produtos já existentes em uma LPS, a abordagem reativa, que nada mais é do que a extensão de uma LPS já implementada para que um produto novo seja gerado a partir dela, ou ainda uma combinação de ambas estratégias [3].

Com o reuso de artefatos em diferentes sistemas, há uma redução nos custos de desenvolvimento deles, uma vez que não é necessário reimplementar o mesmo módulo para cada sistema. Além disso, como os artefatos são reusados em muitos produtos, eles são testados várias vezes. Com isso, há um aumento nas chances de erros serem detectados, o que implica na melhoria da qualidade dos produtos. Há ainda uma redução do *time-to-market* para produtos da LPS, pois, ainda que no início da implementação ele seja alto porque os artefatos precisam ser inicialmente implementados, ele torna-se menor para produtos subseqüentes porque componentes previamente implementados podem ser reutilizados em cada produto novo [4].

Diversas técnicas já amadurecidas para implementação de LPS são utilizadas em fábricas de software, tais como compilação condicional, programação orientada a aspectos (POA), componentes, Extension Points do Eclipse, herança, entre outros. Estas técnicas beneficiam o processo de desenvolvimento de LPS diminuindo os custos de desenvolvimento e manutenção, além de reduzindo o *time-to-market* para o lançamento de produtos novos no mercado.

Como a *TaRGeT* foi desenvolvida utilizando o *framework* Eclipse RCP [5], grande parte de suas variações foram implementadas utilizando Extension Points do Eclipse, o próprio mecanismo de extensão do *framework*. Porém, para algumas situações particulares, outras técnicas foram utilizadas por proverem melhor suporte à implementação desejada.

As principais técnicas utilizadas na implementação e na análise da linha de produtos da TaRGeT serão apresentadas nos próximos tópicos.

4.1 Extension points do Eclipse RCP

A técnica de Extension Points do Eclipse RCP [5] define a possibilidade de adição de uma funcionalidade, também conhecida como contribuição, a um *plugin*. O *plugin* que define um *extension point* define um contrato que estabelece como outros *plugins* poderão adicionar uma contribuição através do extension point pré-estabelecido [21].

Em tempo de execução, os *plugins* do Workbench buscam os registros para todos os seus *extension points*, integrando as funcionalidades implementadas através dos *extension points* à aplicação. Esse mecanismo é o alicerce da extensibilidade do Eclipse [22].

4.2 Herança

Herança é um mecanismo que permite especialização de classes para redefinir a estrutura ou comportamento de classes mais gerais. Os refinamentos podem incluir adição de atributos e operações, e também redefinir o comportamentos de operações já implementadas na classe herdada [7].

A principal vantagem do uso da técnica é o reuso de código, que pode ser herdado por classes de comportamento semelhante [19]. Entretanto a técnica não provê boa modularidade para preocupações transversais (*crosscutting concerns*).

4.3 Arquivos de propriedades

Arquivos de propriedades podem ser usados para armazenar alternativas de valores utilizados pelo programa. Os atributos armazenados podem representar variabilidade em parâmetros utilizados durante a execução do programa. Os arquivos de propriedades podem ser lidos em tempo de execução [7].

Arquivos de propriedades são interessantes porque permitem que clientes configurem alguns itens da aplicação de acordo com suas necessidades sem que nenhuma linha de código-fonte seja alterada [4].

4.4 Compilação condicional

Compilação condicional, ou pré-processamento de código, permite que blocos de código fonte relacionados a uma variação sejam delimitados por diretivas especiais. Os artefatos de código fonte que contêm essas diretivas são pré-processados antes da compilação [7].

A funcionalidade requisitada é selecionada pela definição do símbolo condicional apropriado. Uma vantagem da técnica é o encapsulamento de múltiplas implementações em um único módulo [18].

Por outro lado, preocupações espelhadas, presentes em várias unidades do código-fonte da aplicação, precisarão de diretivas em todos os fragmentos da feature espalhada para adicioná-la ou removê-la do produto.

4.5 Programação Orientada a Aspectos (POA)

Programação Orientada a Aspectos (POA) é uma técnica de desenvolvimento que permite a modularização de preocupações transversais (*crosscutting concerns*) [18].

Preocupações transversais são características funcionais ou não do programa cuja implementação está naturalmente espalhada através do código fonte do programa. Exemplos de preocupações transversais são: controle de persistência, controle de concorrência, controle de transação, entre outros [7].

AspectJ [20] é uma extensão do paradigma de orientação a aspectos para a linguagem de programação Java. Cada aspecto pode definir *features* que afetam diversas partes do sistema identificadas através do uso de *pointcuts*. Eles casam com pontos de junção (*join points*), que são um conjunto de pontos identificados durante o fluxo de execução do programa onde se deseja adicionar partes de código. A técnica permite ainda a introdução de *inter-type declarations*, que são estruturas que permitem a introdução de campos e métodos a classes, além da mudança da hierarquia dos tipos [7].

A principal vantagem do uso da técnica é a o alto de grau de modularidade que pode ser aplicado a *features* transversais. Em contrapartida, com a definição de *pointcuts*, pode-se fortalecer o acoplamento de código, o que prejudica a manutenção do software [19].

4.6 Reflection

Reflection é comumente utilizado por programas que precisam examinar ou modificar o comportamento de aplicações em tempo de execução [23]. Pode ser combinado com o carregamento dinâmico de classes para módulos desconhecidos até o momento da execução, permitindo que operações deste módulo sejam invocadas dependendo do contexto da aplicação [18].

5. IMPLEMENTAÇÃO DA LINHA DE PRODUTOS DA TARGET

A linha de produtos da TaRGeT surgiu com a finalidade de atender a requisitos particulares de clientes distintos, mantendo um baixo custo de desenvolvimento e manutenção das variações.

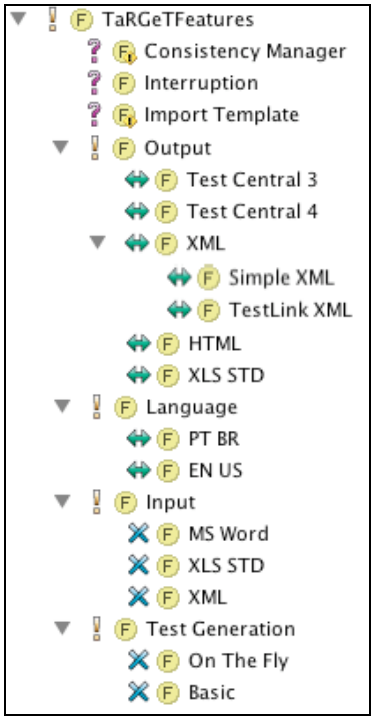


Figura 5.1 - *Feature model* da LPS.

A figura acima apresenta o *feature model* da TaRGeT após a implementação da LPS. Os próximos tópicos detalharão a implementação dos pontos de variação desta linha e de suas respectivas variações.

5.1 TaRGeT Output

A feature de Output da TaRGeT tem por objetivo conceber um arquivo com os casos de teste resultantes da geração automática, após a filtragem e edição realizadas pelo usuário.

Originalmente, a TaRGeT continha apenas um padrão de saída, o Test Central 3. Para atender a requisitos de outros times do BTC-Motorola e a clientes externos, tornou-se necessária a implementação de outros padrões. Os padrões de saída diferem tanto pelo formato quanto pelo conteúdo.

Com isso, foi identificado um ponto de variação da ferramenta e o padrão de saída anteriormente implementado foi extraído do plugin TaRGeT Test Case Generation (TCG) para um *plugin* novo e classificado como uma das variações de *Output*.

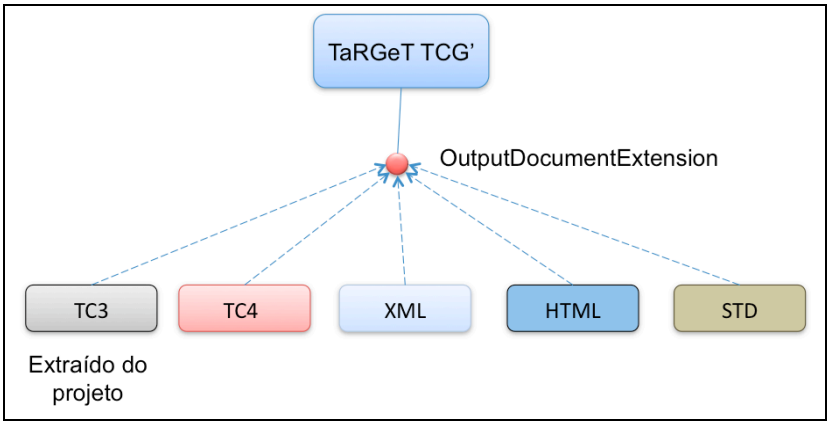


Figura 5.2 - Extração da variação TC3 Output.

A figura a seguir apresenta a *feature* do Output e suas variações no *feature model* da TaRGeT após a implementação da LPS.

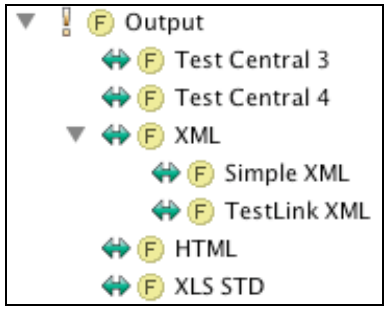


Figura 5.3 - *Feature Model* (TaRGeT Output).

Como o comportamento de alguns métodos da classe responsável pela geração dos arquivos contendo os casos de teste apresentavam um comportamento particular para cada formato especificado, as variações da *feature* foram implementadas utilizando o mecanismo de Extension Points do Eclipse, fazendo uso da técnica de herança. Para isso, foi adicionado ao plugin TaRGeT Test Case Generation um Extension Point contendo apenas uma classe abstrata (OutputDocumentExtension) com os métodos que devem ser implementados pelos *plugins* que adicionarem variações novas para *feature* de Output à linha de produto.

Além disso, a chamada ao método de geração da saída (método principal da variação) foi precedida por linhas contendo uma chamada à *factory*, que é a classe que identifica a variação presente na configuração que está sendo executada, implementada para o ponto de extensão.

```
if (outputDocumentExtension == null) {
       outputDocumentExtension = OutputDocumentExtensionFactory
              .outputExtension().getOutputDocumentExtension();
}

outputDocumentExtension.writeTestCaseDataInFile(result);
```

Listagem 5.1 - Chamada a *factory* de Output.

Para cada variação, um *plugin* novo foi desenvolvido. Dessa forma foram adicionados ao projeto os *plugins* TaRGeT Test Central 3 Output (saída compatível com o *Test Central 3* – sistema de gerenciamento de casos de teste utilizado no BTC-Motorola), TaRGeT Test Central 4 Output (saída compatível com o Test Central 4 – evolução do Test Central 3, também utilizado pelo BTC-Motorola), TaRGeT HTML Output (saída no formato HTML),

TaRGeT XML Output (saída no formato XML) e TaRGeT STD (saída desenvolvida especificamente para o time de STD da Motorola).

Como as variações Output Test Central 3, Output Test Central 4 e Output STD são implementações de saídas XLS para Microsoft Excel 2003/2007, todas apresentam similaridades no que diz respeito a geração e inicialização das tabelas e de estilos de células. Por conta disso, para essas variações, foi utilizada uma solução com herança combinada ao mecanismo de Extension Points do Eclipse.

Para isso, o código em comum às duas variações foi extraído para uma super classe abstrata (OutputDocumentExtensionExcel) que implementa os métodos da interface membro do *extension point* desenvolvido para o Output (OutputDocumentExtension). Dessa forma, as três implementações (ExcelFileFormatterTC3, ExcelFileFormatterTC4 e ExcelFileFormatterSTD) foram reconhecidas pelo mecanismo de extensão sem que fosse necessária replicação desnecessária de código.

A ilustração a seguir apresenta o esquema de herança seguido pelas classes das variações de *Output*.

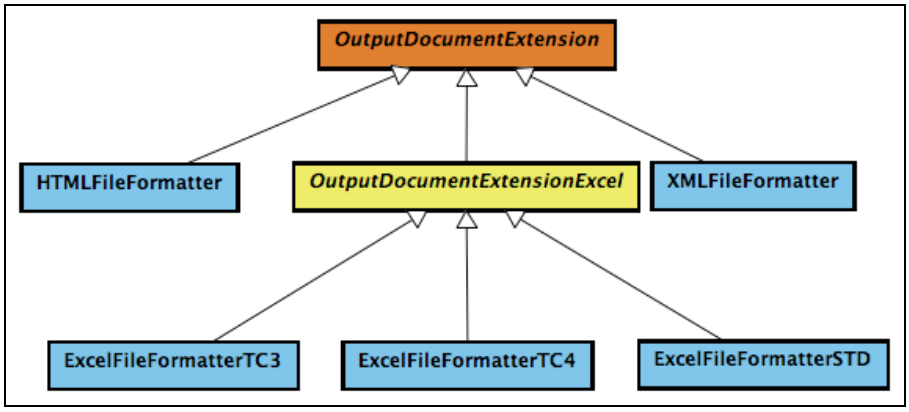


Figura 5.4 - Herança das variações de Output.

Para atender a clientes que utilizam a ferramenta de gerenciamento de testes TestLink [12], foi implementada ainda uma variação, nomeada TestLink XML, com o formato de saída compatível com o importado pela ferramenta.

Como o formato TestLink é também composto por *tags* XML, foi feita apenas uma transformação sobre a saída XML padrão da TaRGeT para que ele fosse concebido. Para que as duas variações fossem mantidas na LPS da TaRGeT, a transformação para o

formato TestLink foi feita através da implementação de um aspecto que é adicionado ao projeto quando a variação TestLink é selecionada.

```
...
File around(File file) throws IOException : generateXML(file)
{
    File fileNew = proceed(file);

   try
    {
        return this.transform(fileNew);
    }
    catch (IOException e)
    {
        throw new org.aspectj.lang.SoftException(e);
    }
}

...
```

Listagem 5.2 - Aspecto para transformação do TestLink XML.

5.2 TaRGeT Import Template

A feature TaRGeT Import Template tem por objetivo extrair informações de cabeçalho e ID de testes de suítes geradas previamente (suítes antigas editadas manualmente ou *templates* gerados pela ferramenta Test Central 4) colocando-as em suítes novas no momento da geração. A principal finalidade é automatizar o trabalho manual de cópia de tais informações entre suítes antigas e novas, evitando que ocorram erros de cabeçalho ou de identificação dos testes.

Originalmente a feature não existia na linha de produto. O ponto de variação foi implementado após a requisição de times da Motorola que precisavam importar dados de *templates* gerados pela ferramenta de gerenciamento de casos de teste *Test Central 4*.

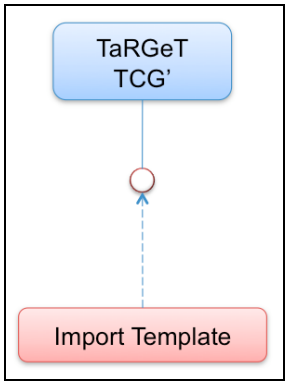


Figura 5.5 - Ponto de variação para a *feature* opcional.

A figura a seguir apresenta o *feature model* da TaRGeT para a *feature* opcional adicionada. O *feature model* inclui uma restrição para esta *feature*: ela só pode ser selecionada, caso a variação de saída selecionada para o produto seja Test Central 4 ou STD, as únicas compatíveis com formato de cabeçalho importado pelo TaRGeT Import Template.

Figura 5.6 - *Feature model* para a *feature*.

Inicialmente, a *feature* TaRGeT Import Template foi implementada com o uso isolado de Extension Points do Eclipse. Após uma análise detalhada de sua implementação e da comparação com outras opções, a solução utilizando Extension Points do Eclipse combinado com Aspectos foi identificada como sendo a melhor solução por manter uma melhor modularidade da feature opcional. Isso ocorreu principalmente por tratar-se de uma feature opcional. Para ela, a implementação com POA e Extension Points permitiu que todo o código relacionado a feature fosse removido do core da aplicação no momento em que a feature não estava mais presente. A análise pode ser encontrada nos próximos capítulos deste trabalho.

Para a implementação com Extension Points do Eclipse e Aspectos foi definido um aspecto (ImportTemplateAspect) que encapsula a chamada à *factory*

(InputDocumentTemplateExtensionFactory) e aos métodos da feature e foi adicionado um *extension point* ao *plugin* TaRGeT TCG com uma interface contendo a assinatura dos métodos necessários para que um *template* seja importado.

Como os formatos do cabeçalho das saídas STD e Test Central 4 são iguais, apenas um plugin com a implementação para o *extension point* foi criado, o TaRGeTImportTemplateTC4STD.

A implementação atual permite que a *feature* TaRGeT Import Template seja implementada facilmente para formatos diferentes das saídas STD e Test Central 4 apenas com a implementação de um plugin que implemente os serviços do extension point definido para a *feature*.

O *feature model* (figura 5.1) apresenta restrições para a *feature*. Como o cabeçalho da suíte de testes está presente apenas nas saídas Test Central 4 e XLS STD, a *feature* Import Template só pode ser selecionada quando um desses formatos é selecionado para o produto.

5.3 TaRGeT Test Generation

A TaRGeT apresenta os casos de teste gerados à medida que a seleção deles é realizada. Esta forma de geração bastante útil foi nomeada geração “On The Fly” e permite que usuários da ferramenta possam visualizar os resultados decorrentes de sua seleção sem a necessidade de gerar novamente a suíte de testes a cada edição dos critérios.

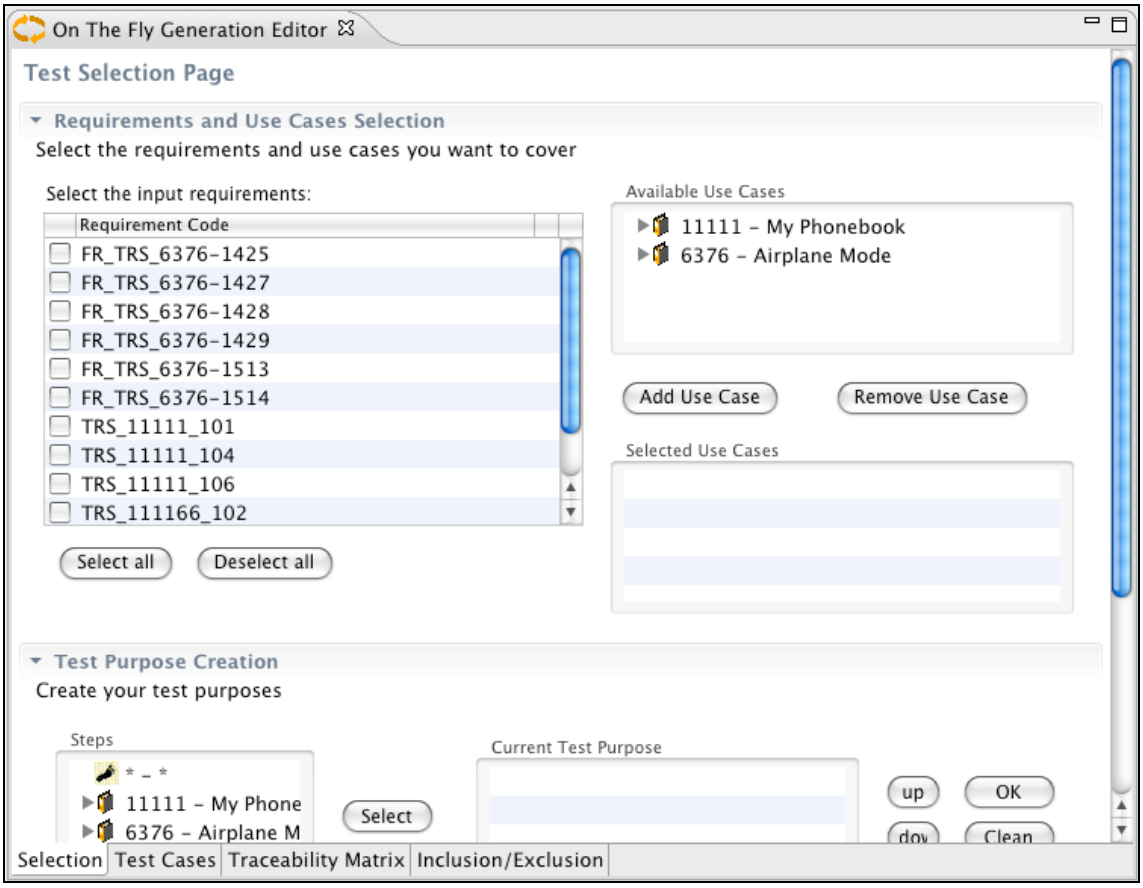


Figura 5.7 - On The Fly Generation.

Entretanto, para alguns clientes específicos, a seleção de critérios para geração dos casos de teste tornou-se desnecessária por serem requisitados sempre todos os casos de teste gerados pela ferramenta. Desta forma, foi identificado um requisito novo desses clientes que, devido a decisões estratégicas da equipe, mudaram sua política para geração de suítes.

Para atender a esse requisito novo, um ponto de variação foi criado. A variação “On The Fly” já implementada no core da ferramenta foi extraída para um plugin e uma variação nova, nomeada “Basic Generation”, foi implementada para atender ao requisito particular.

O objetivo da variação nova é gerar uma suíte com todos os testes possíveis a partir dos casos de uso fornecidos como entrada. Como, para isso, nenhuma seleção ou filtragem é necessária, apenas um item foi adicionado no menu para a geração. Quando o item é selecionado, a suíte é gerada, de forma bastante simplificada quando comparada à geração “On The Fly”.

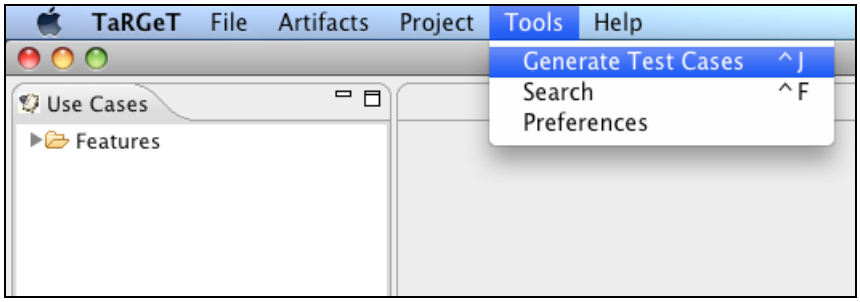


Figura 5.8 - Basic Generation.

O algoritmo de geração dos casos de teste permaneceu inalterado, apenas a seleção e filtragem dos casos de teste, implementada na variação “On The Fly”, deixou de ser aplicada na variação nova. Como a implementação das duas variações são compostas apenas por elementos de GUI, mais especificamente *actions* e *editors*, ambas foram implementadas utilizando o suporte a componentes do Eclipse RCP.

Para isso, para a variação já implementada no *core*, foi necessário extrair os componentes já implementados da geração “On The Fly” para um *plugin* separado do *core* da aplicação, que adiciona conteúdo a GUI da aplicação RCP. Nele, os seguintes *extension points* do Eclipse RCP foram implementados, compondo a GUI necessária para o “On The Fly”:

- **org.eclipse.ui.commands** - Utilizado para implementação da ação relacionada à abertura do "On The Fly" na barra de ferramentas;
- **org.eclipse.ui.bindings** - Utilizado para adicionar um atalho para a ação de abertura do "On The Fly" adicionada à barra de menu;
- **org.eclipse.ui.editors** - Utilizado para adicionar o editor "On The Fly" ao Workbench;
- **org.eclipse.ui.actionSets** - Utilizado para adição do item de menu responsável pela abertura do editor "On The Fly" ao menu "Tools" da TaRGeT.

Já para a variação nova, foi criado um *plugin* novo, que também adiciona conteúdo à GUI da aplicação RCP, onde foram adicionados os componentes de GUI necessários para o funcionamento da variação “Basic Generation”. A implementação também foi realizada com o uso de *extension points* do próprio Eclipse RCP. Os mesmos itens da variação “On The

Fly" foram implementados para a variação "Basic Generation", com exceção do *extension point* org.eclipse.ui.editors.

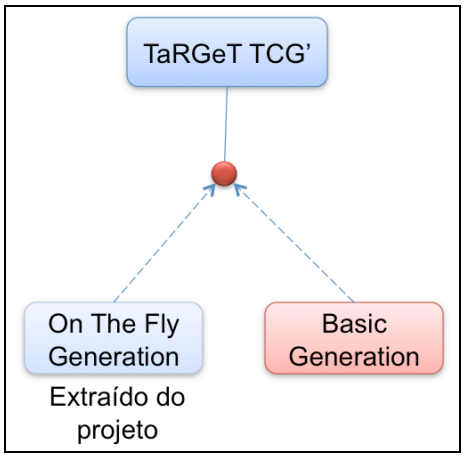


Figura 5.9 - Extração da variação On The Fly.

5.4 TaRGeT Consistency Management

A *feature* Consistency Management tem por objetivo controlar a consistência de testes previamente gerados, caso o usuário faça modificações no documento de casos de uso utilizado para a geração dos testes [13].

A *feature* apresenta o percentual de similaridade entre testes novos e antigos e os campos diferentes entre eles. O usuário deve ser capaz de associar testes novos a velhos baseado no nível de similaridade entre eles e de atribuir um novo ID a um caso de teste novo, decorrente da inclusão de novos passos ou fluxos no documento de casos de uso [13].

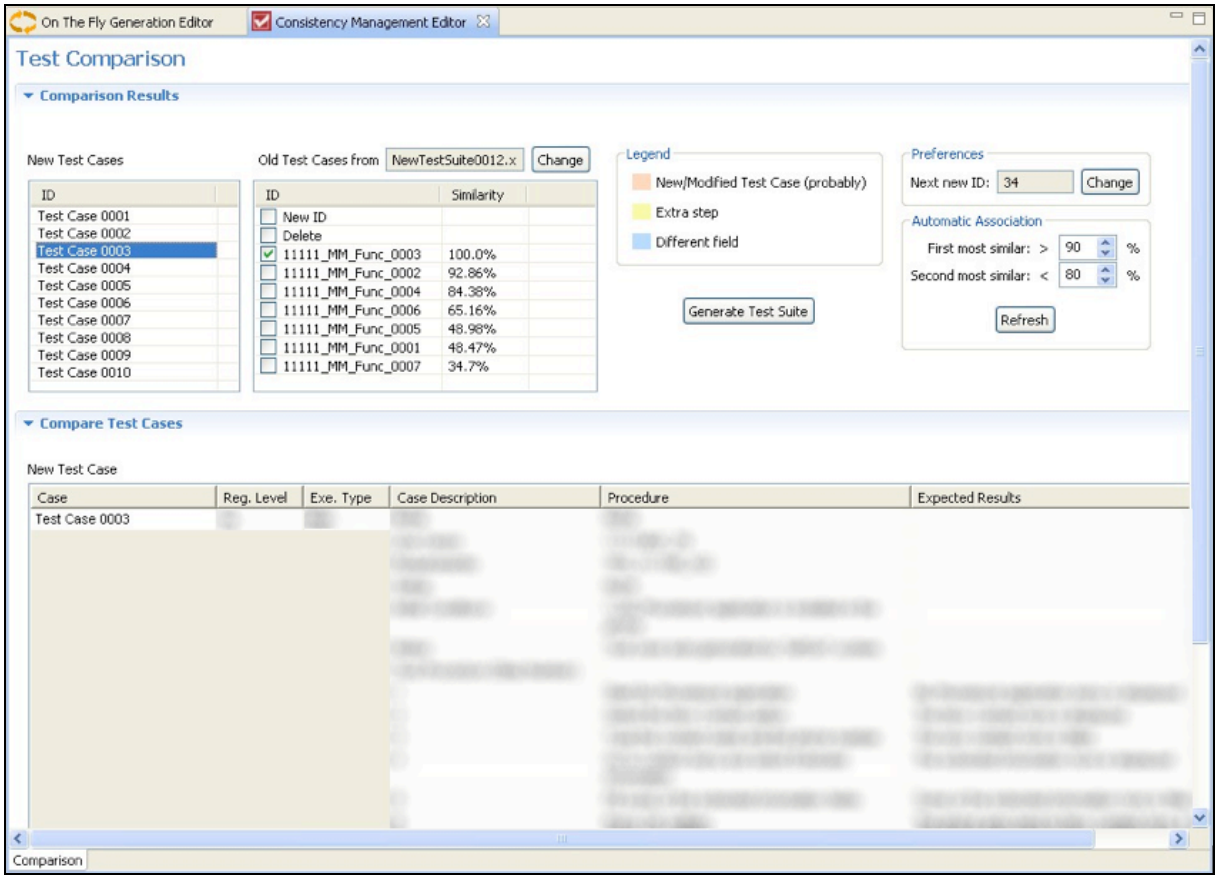


Figura 5.10 - Comparação de suítes de testes.

Originalmente a *feature* não existia na aplicação. O ponto de variação foi implementado após a requisição de times da Motorola que precisavam manter a consistência de identificadores de casos de testes de suítes antigas quando os documentos de entrada contendo casos de uso fossem alterados.

Da mesma forma que a *feature* Import Template, Consistency Management também foi inicialmente implementada com o uso isolado de Extension Points do Eclipse. Após uma análise detalhada da implementação e da comparação com outra solução, concluiu-se que a implementação mais vantajosa para ponto de variação seria a que envolvesse o uso de Aspectos. Isso ocorreu principalmente por tratar-se de uma *feature* opcional, em que desejava-se remover todo o código relacionado a preocupação do *core* da aplicação, no momento em que ela não estivesse mais presente na configuração do produto. Essa análise também pode ser encontrada no Capítulo 6 deste trabalho.

Para a implementação com POA, foi definido um aspecto que encapsula a chamada aos métodos da feature (ConsistencyManagerAspect) e que só é adicionado ao *build path* da aplicação caso a *feature* esteja na seleção do produto.

```
privileged public aspect ConsistencyManagerAspect {

	private Button OnTheFlyGeneratedTestCasesPage.consistencyManagementcheckButton;

	pointcut consistencyManagement(
		TestSuite<TestCase<FlowStep>> testSuite,List<TextualTestCase>
		textualTestCases, OnTheFlyGeneratedTestCasesPage ontTheFlyPage) :
		call(void OnTheFlyGeneratedTestCasesPage.generateSuite(TestSuite,List))
		&& args(testSuite,textualTestCases)
		&& target(ontTheFlyPage);

	void around(TestSuite<TestCase<FlowStep>> testSuite,List<TextualTestCase>
			textualTestCases,OnTheFlyGeneratedTestCasesPage ontTheFlyPage)
			throws IOException, TargetException:
		consistencyManagement(testSuite,textualTestCases, ontTheFlyPage){

		if(ontTheFlyPage.consistencyManagementcheckButton.getSelection()) {
			ConsistencyManagerExtensionImplementation cm =
				new ConsistencyManagerExtensionImplementation();

			...
		}
		else {

			...
		}

	}
	...
}
```

Listagem 5.3 - Aspecto para adição da *feature* Consistency Management.

O *feature model* (figura 5.1) apresenta restrições para a *feature* Consistency Management. Como os clientes que requisitaram a variação de saída XLS STD não solicitaram inicialmente a feature Consistency Management, a importação dos casos de teste gerados nesse formato não foi implementada. Com isso, a *feature* Consistency Management não pode ser associada a esse padrão de saída.

5.5 TaRGeT Test Suite Extractor

Com a implementação da *feature* Consistency Management, melhor detalhada nos próximos tópicos, tornou-se necessária a extração dos casos de teste das suítes geradas pela TaRGeT. Como a linha de produto da ferramenta inclui um ponto de variação específico para a feature Output, vários formatos de saída são gerados pela ferramenta.

Para que todos os formatos diferentes de suítes geradas pela TaRGeT fossem suportados pela feature Consistency Management, tornou-se necessária a implementação de um ponto de variação novo para a feature TaRGeT Suite Extractor, responsável pela extração dos casos de teste das suítes.

Originalmente, como a TaRGeT continha apenas um padrão de saída, apenas um método de extração estava implementado no *core* da aplicação, o extrator do formato Test Central 3. Com a implementação dos padrões de saída novos, o único padrão de extração anteriormente implementado foi extraído do *core* da ferramenta e identificado como uma das variações da *feature* TaRGeT Suite Extractor.

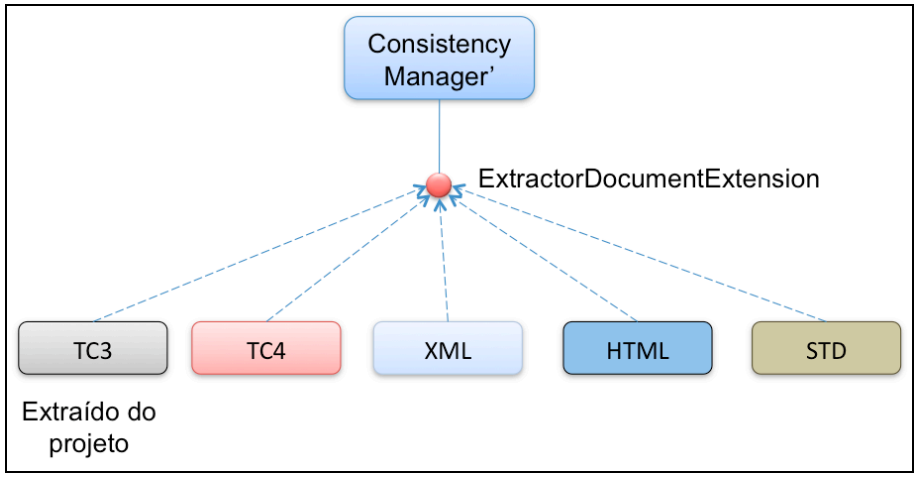


Figura 5.11 - Extração da variação TC3 Extractor.

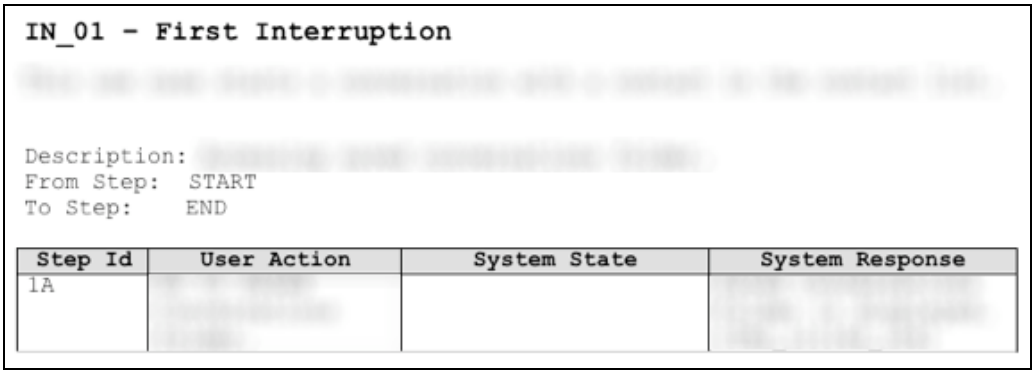
Da mesma forma que as variações de *Output* da *TaRGeT*, as variações da *feature* foram implementadas com o uso de Extension Points do Eclipse. Para isso, foi adicionado ao *plugin* TaRGeT Test Case Generation, um *extension point* contendo apenas uma interface (ExtractorDocumentExtension) com os métodos que devem ser implementados

pelos *plugins* que adicionarem variações novas para a *feature* Suite Extractor à linha de produto.

Para cada variação, um *plugin* com a implementação da interface do ponto de extensão para a *feature* foi criado.

5.6 TaRGeT Interruption

O conteúdo dos casos de uso da TaRGeT incluía, opcionalmente em sua versão inicial, para cada caso de uso, fluxos de interrupção que eram compostos por passos que poderiam ou não ocorrer entre outros já previstos para o caso de uso, e que interromperiam o fluxo esperado.


IN_01 - First Interruption

Description:
From Step: START
To Step: END

Step Id	User Action	System State	System Response
1A			

Figura 5.12 - Fluxo de interrupção.

Esta seção foi removida da especificação do documento de entrada da ferramenta por cumprir uma função muito semelhante a parte destinada aos fluxos de exceção. Com isso, os passos antes representados como fluxos de interrupção passaram a integrar os fluxos alternativos.

Entretanto, após a remoção do suporte aos fluxos de interrupção, alguns clientes requisitaram ainda a permanência da compatibilidade da ferramenta com os arquivos de entrada que possuíam fluxos de interrupção. A necessidade de adaptar documentos de

casos de uso antigos, substituindo os fluxos de interrupção por fluxos alternativos, fez com que alguns clientes não migrassem seus documentos para o formato novo.

Para atender aos clientes que desejavam utilizar a *feature* de interrupção, foi identificado um ponto de variação novo, que resultou na implementação da *feature* opcional TaRGeT Interruption. Para a implementação da *feature*, todo o código associado a ela foi extraído de uma versão anterior da ferramenta, que ainda continha o suporte a interrupções, e transferido para um aspecto fazendo uso das técnicas de POA.

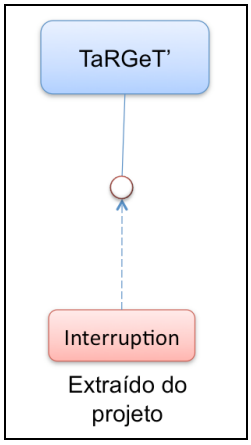


Figura 5.13 - Extração da *feature* TaRGeT Interruption.

A solução foi implementada com POA porque o código relacionado a *feature* estava presente em diversas camadas da aplicação como GUI, controle e negócios, o que caracterizava um alto grau de espalhamento de código. Com POA, foi possível modularizar o código associado à *feature*, adicionando-o apenas quando necessário.

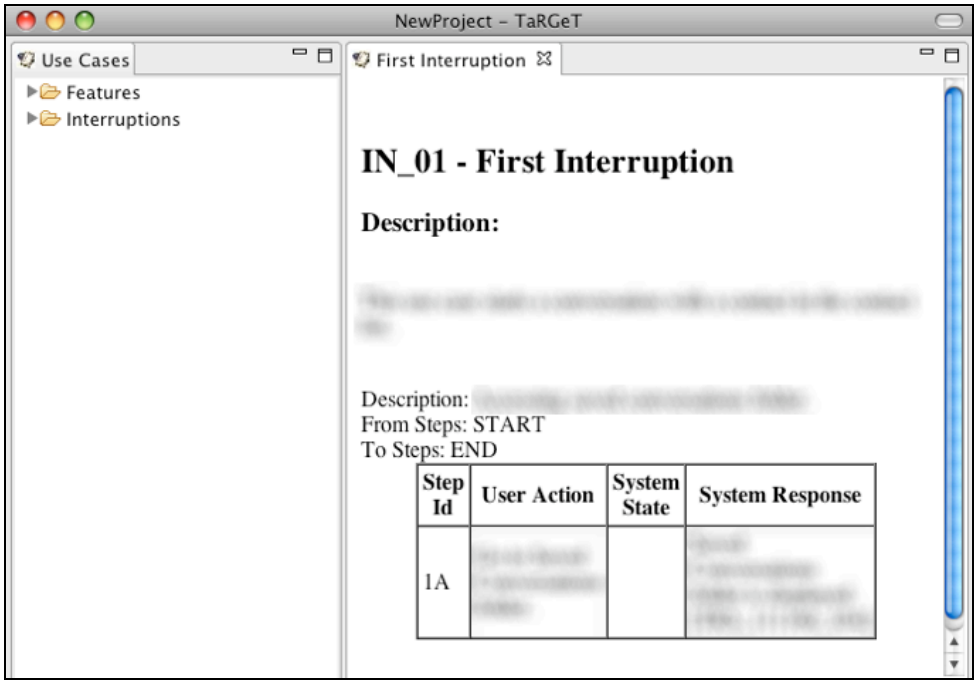


Figura 5.14 - Presença da *feature* TaRGeT Interruption na GUI.

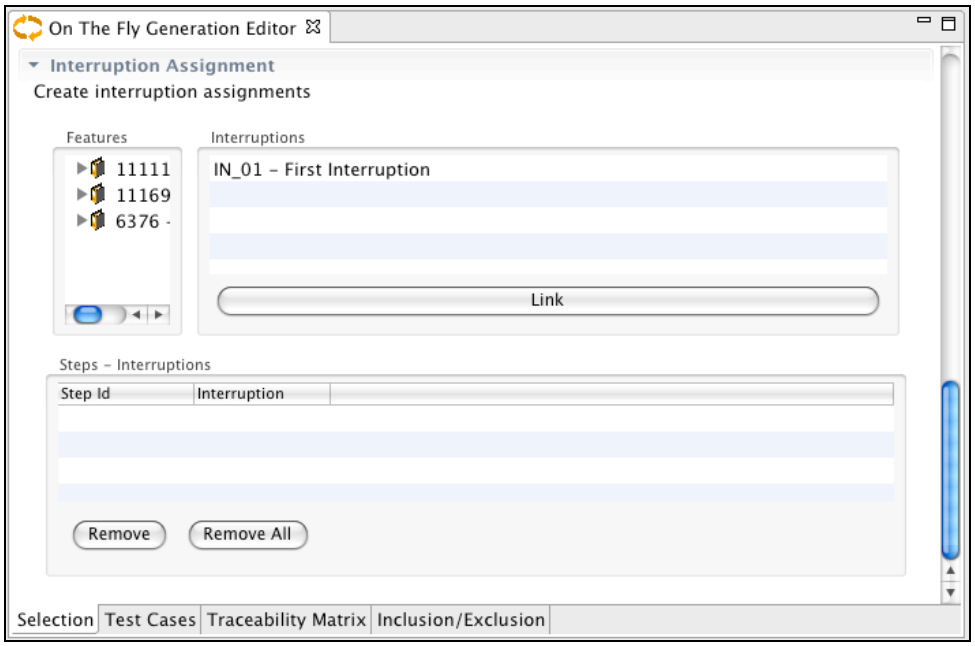


Figura 5.15 - Presença da *feature* na seleção dos casos de teste.

5.7 TaRGeT Internationalization Support

A interface do usuário da TaRGeT foi inicialmente implementada apenas em Inglês Americano. Desta forma, mensagens de sucesso, erro e aviso eram apresentadas sempre no mesmo idioma.

A fim de aumentar a aceitação da ferramenta e tornar desnecessário o conhecimento no idioma estrangeiro, todas as mensagens de GUI da ferramenta foram substituídas por chamadas à classe que representa o arquivo de propriedades de idioma conforme as listagens 5.4 e 5.5. Com isso, todo o texto da GUI passou a ser armazenado em arquivos de propriedades que podem ser facilmente trocados por outros contendo as mensagens em idiomas diferentes, como o apresentado na listagem 5.6. Desta forma, a LPS da TaRGeT ganhou um ponto de variação novo, que corresponde a *feature* obrigatória TaRGeT Internationalization Support.

```
...
TableColumn tc = new TableColumn(table, SWT.LEFT);
tc.setText(Properties.getProperty("use_cases"));

...
```

Listagem 5.4 - Texto na GUI antes do uso de arquivos de propriedades.

```
...
TableColumn tc = new TableColumn(table, SWT.LEFT);
tc.setText(Properties.getProperty("use_cases"));

...
```

Listagem 5.5 - Texto na GUI com arquivos de propriedades.

```
...
use_cases=Use Cases

...
```

Listagem 5.6 - Exemplo de arquivo de propriedades.

Como variações, foram implementados arquivos de propriedades para o idioma Português Brasileiro e foi extraído o idioma já existente Inglês Americano. Após a implementação da LPS, para adição de idiomas novos, é necessário apenas a implementação de arquivos de propriedades com a tradução das mensagens da GUI da ferramenta para o idioma desejado.

A ferramenta de gerenciamento de variações ficou responsável pela troca dos arquivos de propriedades correspondentes ao idioma selecionado para o produto.

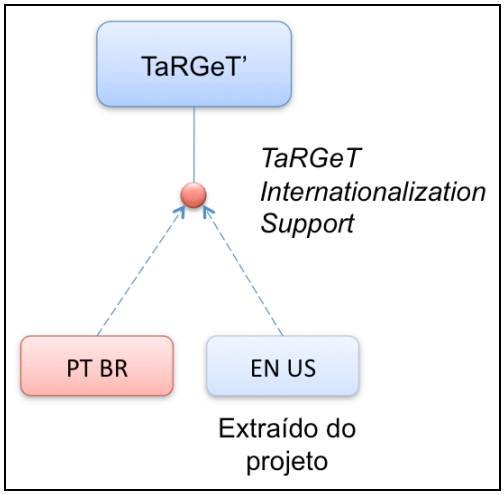


Figura 5.16 - Extração da variação EN US.

5.8 TaRGeT Input

A *feature* de Input da TaRGeT tem como finalidade a leitura de casos de uso a partir de documentos de entrada em diferentes formatos.

Originalmente, a TaRGeT continha apenas um padrão de entrada, o Microsoft Word. Para atender a requisitos de outros times do BTC-Motorola e a clientes externos, tornou-se necessária a implementação de outros padrões. Os padrões de entrada diferem-se tanto pelo formato quanto pelo conteúdo dos casos de uso.

Com isso, foi identificado um ponto de variação novo na TaRGeT, referente ao formato do documento de entrada da ferramenta. O padrão anteriormente implementado foi extraído

do *plugin* TaRGeT Project Manager (PM) para um *plugin* novo e classificado como uma das variações do Input.

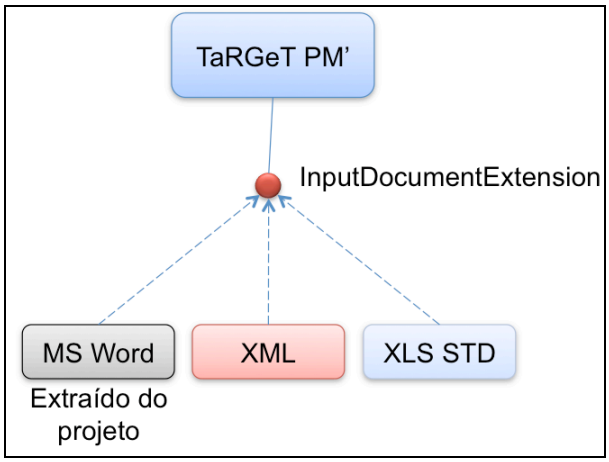


Figura 5.17 - Extração da variação MS Word Input.

Além da variação extraída da implementação, foram desenvolvidas ainda as variações XML Input e XLS STD.

Da mesma forma que a *feature* TaRGeT Output, e pelo mesmo motivo, as variações foram implementadas utilizando o mecanismo de *extension points* do Eclipse. Para isso, foi adicionado ao *plugin* TaRGeT Project Manager, um ponto de extensão contendo apenas uma classe abstrata (InputDocumentExtension) com os métodos que devem ser implementados pelos *plugins* que adicionarem variações novas da *feature* de Input à linha de produto.

Além disso, a chamada ao método de carregamento do arquivo (método principal da variação) foi precedida por linhas contendo uma chamada à *factory* (classe que identifica a variação presente na configuração que está sendo executada) implementada para o *extension point*.

```
InputDocumentData inputDocumentData = InputDocumentExtensionFactory.
       getInputDocumentDataByExtension(fileExtension);

UseCaseDocument useCaseDocument = inputDocumentData.getInputDocumentExtension()
       .loadDocument(file);
resultDocuments.add(useCaseDocument);
```

Listagem 5.7 - Chamada à *factory* de Input.

Para cada variação, um *plugin* novo foi implementado. Dessa forma foram adicionados ao projeto os *plugins* TaRGeT Word Input, TaRGeT XLS STD Input e TaRGeT XML Input.

6. ANÁLISE DE SOLUÇÕES ALTERNATIVAS

Neste tópico serão detalhadas e analisadas as implementações com diferentes combinações de técnicas para as mesmas variações realizadas com a finalidade de melhor exemplificar as diferenças ressaltando as vantagens e desvantagens das mesmas.

Nas implementações foram analisados os critérios quantidade de código, capacidade de extensão por terceiros, modularidade da feature, escalabilidade e conhecimento necessário para o uso da técnica.

A quantidade de linhas de código de cada solução foi levada em consideração na análise porque soluções que atendem bem aos demais critérios, mas exigem uma quantidade consideravelmente maior de código, que não poderá ser gerado de forma automática em sua implementação, podem não justificar suas vantagens devido ao esforço necessário para implementação.

O critério capacidade de extensão por terceiros foi adotado porque a TaRGeT, ferramenta analisada, também é utilizada por clientes externos, que não têm acesso ao código fonte da aplicação. Dessa forma, técnicas que viabilizem a implementação de variações, como um tipo de saída para suítes de testes específicas, sem que o código fonte do core seja conhecido, são mais apropriadas para o desenvolvimento da linha.

Os quesitos modularidade da feature e escalabilidade foram selecionados por serem critérios básicos para que a qualidade seja mantida em qualquer linha de produtos mantendo o baixo custo de manutenção da mesma.

Por fim, o conhecimento necessário para o uso da técnica avaliada foi levado em consideração por ser um item importante para equipes de desenvolvimento em que o tempo de permanência dos desenvolvedores é baixo. A TaRGeT é desenvolvida por um time de pesquisa que tem como desenvolvedores alunos de graduação e pesquisadores, que permanecem no time enquanto estão desenvolvendo seus trabalhos. Técnicas com alto grau de complexidade, quando adotadas, podem elevar os custos de treinamento para desenvolvedores e, conseqüentemente, os custos de desenvolvimento e manutenção da linha.

6.1 Implementações de TaRGeT Output

Para esta *feature* foram analisadas as implementações com Extension Points do Eclipse, herança com compilação condicional, e herança combinada com Reflection e arquivo de propriedades.

As implementações serão melhor detalhadas nas próximas subseções, onde serão discutidos seus pontos fortes e fracos.

Os seguintes itens compõem cada uma das subseções de cada possibilidade de implementação:

- Composição da variação (*Configuration Knowledge*) - Lista dos itens que devem ser adicionados ao projeto para que a feature em questão seja adicionada ao produto;
- Código original afetado - Artefatos que foram modificados com a implementação da feature utilizando as técnicas analisadas;
- Código adicionado - Artefatos adicionados ao projeto em conseqüência do uso das técnicas analisadas para implementação da feature;
- Vantagens e desvantagens da aplicação das técnicas analisadas.

6.1.1 Extension points do Eclipse

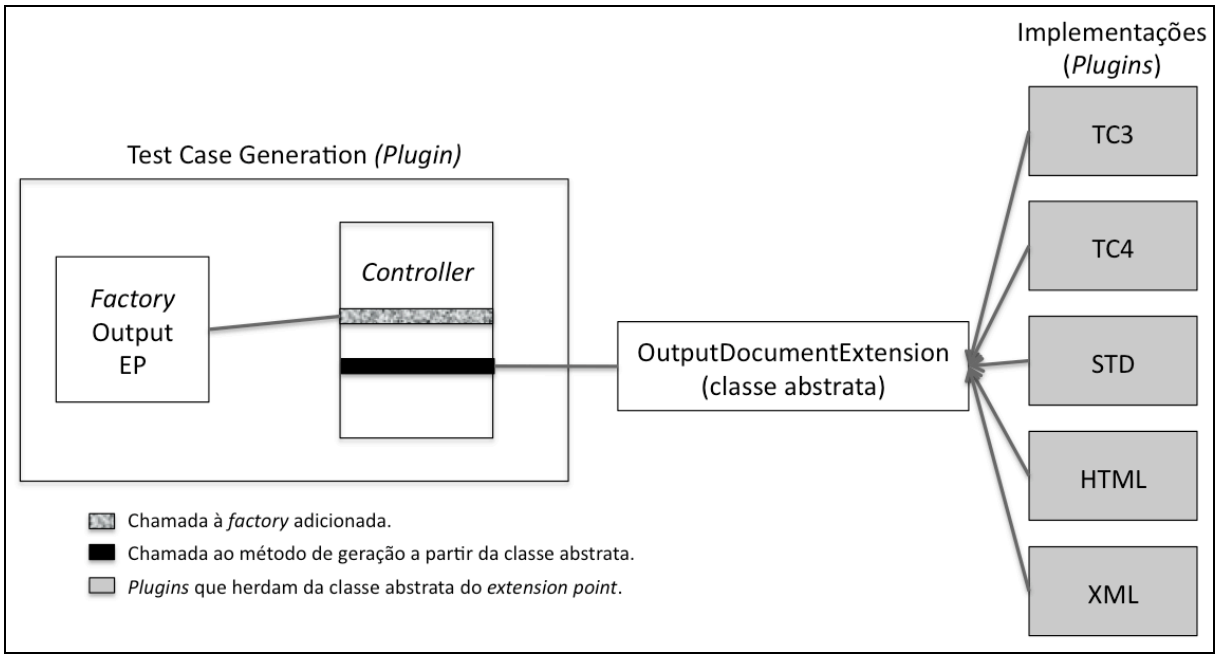


Figura 6.1 - Output com Extension Point do Eclipse.

Composição da variação (*Configuration knowledge*)

- target.product' – target.product original com adição da linha que referencia o plugin que implementa o *extension point* de *Output* (ou, para adicionar a funcionalidade ao produto gerado, o config.ini' – config.ini original com adição do ID do plugin que implementa o *Output*);
- *Plugin* que implementa o *extension point*.

Código original afetado

- Adição da variação de output escolhida para a configuração no arquivo de configuração target.product do *Core* da aplicação;
- Adição do *extension point* de Output ao *plugin* TaRGeT Test Case Generation, responsável pela geração das suítes de teste;
- A chamada ao método de *output* original foi precedida pela chamada à *factory* do *extension point*.

Código adicionado

- Um *plugin* novo para cada variação (para TC3, TC4, STD, XML e HTML - 5 *plugins* novos);
- *Factory* para reconhecer o ponto de extensão implementado e mapear seus atributos (extension point ID, *configuration element* e *implementation attribute*) em um objeto cuja classe implementa a interface OutputDocumentExtension;

Quantidade de linhas de código fonte (todas as variações – excluindo os arquivos de configuração XML): 1741.

Vantagens

A *factory* para Extension Points, ainda que exija um maior esforço de implementação por possuir maior complexidade e ter mais detalhes, como mapeamento de atributos e referência a IDs de *plugins*, não necessita de alterações para variações novas implementadas posteriormente.

Isso faz com que a linha de produtos tenha um baixo custo de manutenção para variações novas (não é necessário nenhuma alteração no *core* da aplicação), fornecendo um bom suporte à escalabilidade. Com isso, a chance de inserção de *bugs* no core da aplicação durante a implementação de variações novas é eliminada. Como nenhuma alteração no core é feita durante o desenvolvimento da variação, o esforço nos procedimentos para o release do projeto é reduzido, já que não são necessárias inspeções de código para aprovar modificações no *core*.

Por não ser necessário nenhum conhecimento a respeito do código fonte do *core* da aplicação, utilizando Extension Points, a implementação de variações por terceiros é bem vinda, sendo necessário apenas conhecer a interface do *extension point* de Output.

Além disso, o mecanismo de *plugins* do Eclipse faz com que sejam necessárias poucas configurações para alternar entre as variações disponíveis. Para que isso ocorra, é necessário apenas editar o arquivo de configuração .product da aplicação RCP ou editar o arquivo config.ini do produto gerado, deixando o *configuration knowledge* para o ponto de variação bastante simplificado.

Desvantagens

Para que variações sejam implementadas utilizando *extension points* é necessário conhecer o *framework* para desenvolvimento de aplicações Eclipse RCP, que, como qualquer *framework*, exige certo tempo de aprendizado, o que pode aumentar os custos do projeto.

Além disso, comparando a solução a outras implementações, tais como herança com compilação condicional, há uma maior complexidade para implementação da *factory* quando o *extension point* é utilizado. Ela contém mais detalhes, como o ID do plugin, além de mapeamentos para os atributos do *extension point* e conseqüentemente mais linhas de código.

6.1.2 Herança e compilação condicional

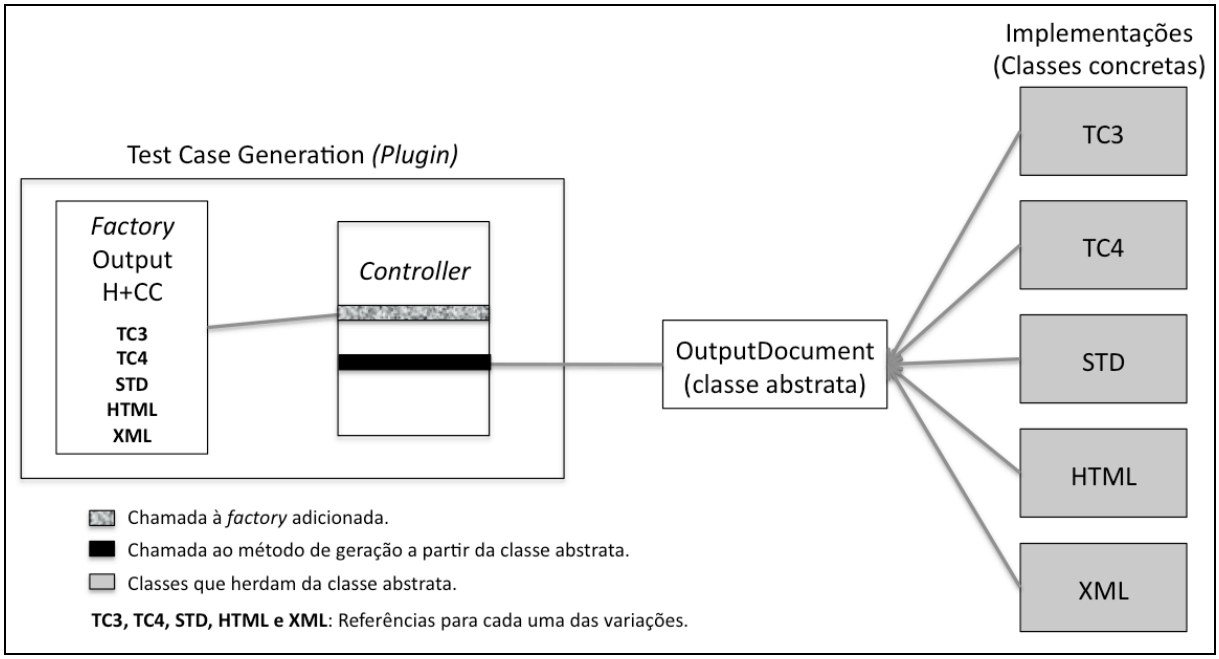


Figura 6.2 - Output com Herança e Compilação Condicional.

Composição da variação (*Configuration knowledge*)

- Classe que implementa a interface definida para a *feature* de Output;
- Parâmetro de informação para a compilação condicional para que a cláusula referente à variação escolhida seja satisfeita.

Código original afetado

- FactoryOuput.java' - FactoryOuput.java original com a adição de cláusulas (#ifdef) para cada variação nova;
- A chamada ao método de output original foi precedida pela chamada à *factory* de Output.

Código adicionado

- *Factory* de *Output* com um #ifdef para cada possibilidade de variação, cada uma contendo o código necessário para que a subclasse seja instanciada;
- A subclasse de cada variação.

Quantidade de linhas de código fonte (todas as variações): 1741.

Vantagens

Para a implementação com herança e compilação condicional não são necessárias alterações nos arquivos de configuração dos *plugins*, o que elimina as chances de inserção de *bugs* nas configurações da aplicação.

Além disso, a implementação de variações novas se dá de forma mais simplificada. Para isso, é necessário apenas estender a classe abstrata da *feature* ou implementar a interface da mesma.

Como a sintaxe da compilação condicional é bastante simples, seu tempo de entendimento torna-se irrelevante. Com isso o tempo de entendimento da técnica e o esforço para a implementação são inferiores se comparados a outras técnicas.

Desvantagens

Com o uso de herança com compilação condicional, a *factory*, que contém cláusulas #ifdef, precisa ser editada a cada variação nova. Isso faz com que o suporte a escalabilidade para feature Output seja bastante precário. Pois o código fonte da *factory* tende a crescer e a ficar confuso com a sintaxe da compilação condicional misturada ao código da preocupação.

Além disso, o uso da pré compilação pode facilmente introduzir erros no código fonte difíceis de detectar, já que erros no código contido nas cláusulas são identificados pelo compilador Java por permanecerem comentados até que o pré processamento ocorra.

Para suprir as deficiências da técnica, que já estão muito bem resolvidas com outras técnicas mais avançadas, é necessário o uso de ferramentas auxiliares no projeto, como visualizadores de código etc., que seriam totalmente desnecessários quando utilizando técnicas como Extension Points do Eclipse.

6.1.3 Herança, Arquivos de propriedades e Reflection

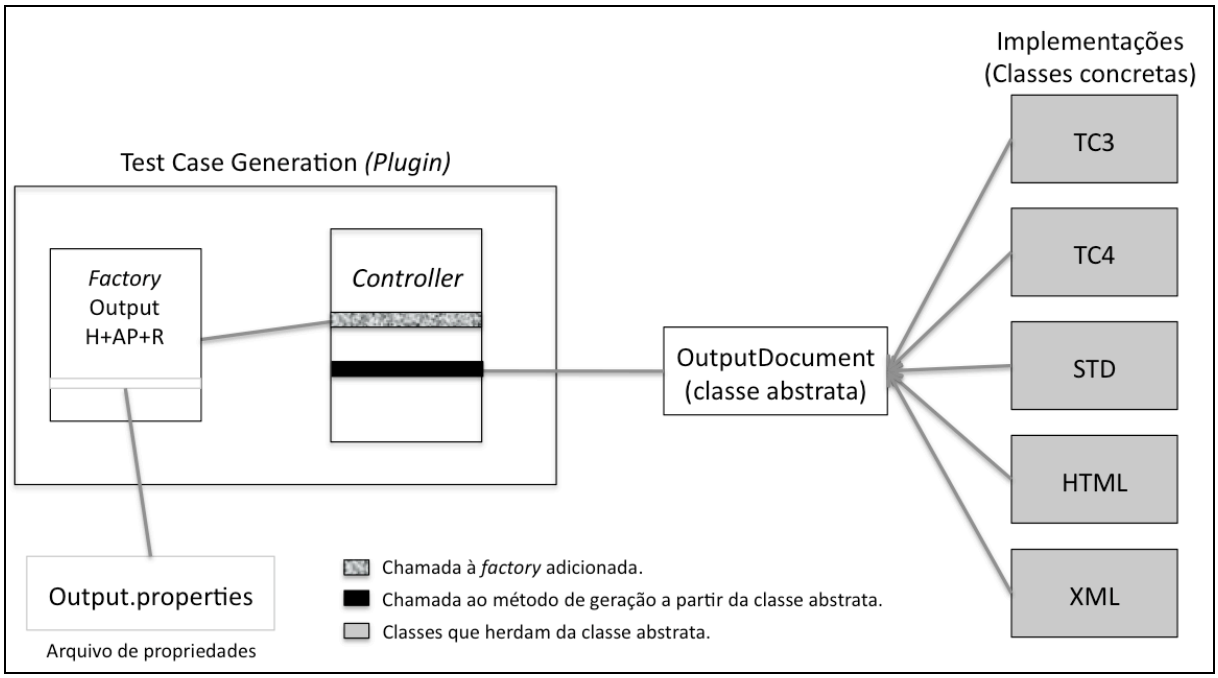


Figura 6.3 - Output com Herança, Arquivo de propriedades e Reflection.

Composição da variação (*Configuration knowledge*)

- Classe que implementa a classe abstrata definida para a *feature* de *Output*;
- Output.properties' – Arquivo de propriedades alterado com o nome da classe que implementa a classe abstrata definida para a *feature*.

Código original afetado

- A chamada ao método de Output original foi precedida pela chamada à *factory* de Output.

Código adicionado

- *Factory* que lê de um arquivo de propriedades qual subclasse deve ser instanciada para geração da saída;

- Arquivo de propriedades contendo uma entrada que indica qual variação será utilizada;
- Uma subclasse para cada variação.

Quantidade de linhas de código (todas as variações): 1540.

Vantagens

Por ter uma *factory* mais simplificada, a implementação do ponto de variação utilizando herança, arquivo de propriedades e Reflection é bastante simplificada. Além de conter menos detalhes na *factory*, a implementação de variações novas não requer nenhum conhecimento de Eclipse RCP, não é necessário nenhuma alteração nos arquivos de configuração nem dos *plugins*, nem do produto, sendo o conhecimento em Java suficiente.

Assim como na implementação com Extension Points do Eclipse, a *factory* não precisa ser editada a cada variação nova e conseqüentemente não tende a crescer.

Desvantagens

Como a classe da variação é lida de um arquivo de propriedades, erros (nome da classe ou caminho do pacote) na edição do arquivo para a troca da variação do produto só serão percebidos em tempo de execução.

6.1.4 Comparativo das implementações

Critério / Implementação	Herança + Compilação condicional	Herança + Arquivo de propriedades + Reflection	Extension point do Eclipse
Quantidade de código	1562 linhas	1540 linhas	1741 linhas
Capacidade de extensão por terceiros	É necessário alterar a factory para dar suporte às variações novas.	O produto não precisa ser recompilado para ter a variação adicionada. É necessário apenas conhecer a interface para implementação de variação nova.	O produto não precisa ser recompilado para ter a variação adicionada. É necessário apenas conhecer a interface para implementação de variação nova.
Modularidade da feature	Parte do código da feature permanece espalhado. A factory da feature permanece com rastros da variação mesmo na sua ausência.	A variação fica modularizada na subclasse da variação. Porém parte do código da feature permanece espalhado na aplicação.	A variação fica modularizada na subclasse da variação. Porém parte do código da feature permanece espalhado na aplicação (Chamada a factory do extension point).
Escalabilidade	A factory tende a crescer e o código fonte a ficar confuso.	Não é necessário alterar o código fonte original para variações novas.	Não é necessário alterar o código fonte original para variações novas. Nenhuma técnica adicional é necessária para alternar entre as variações.
Conhecimento necessário	Apenas conhecimento em Java.	Apenas conhecimento em Java.	Conhecimento em Java, Eclipse RCP e seu mecanismo de *plugins/extension points*.

Tabela 6.1 - Comparativo das implementações de Output.

6.1.5 Conclusões

A implementação com herança e compilação condicional não fornece ao projeto um bom suporte a escalabilidade e modularidade. O código da *factory* tende a crescer já que ganha uma cláusula nova para cada variação nova implementada. Além disso, o código

adicionado representa um espalhamento do código da variação, demonstrando perda de modularidade da mesma.

Soma-se a esses problemas, a perda de entendimento do código da *factory*, pois ele tende a ficar confuso com o código da compilação condicional misturado com o código Java, ambos com sintaxes distintas. Ainda sobre o código referente à compilação condicional, possíveis *bugs* em sua implementação só serão conhecidos em tempo de execução, o que significa maior chance de inserção de falhas no *core* da aplicação.

A implementação realizada com Reflection, herança e arquivo de propriedades apresenta-se como sendo uma opção viável tanto por manter sua *factory* imutável mesmo com a implementação de variações novas, o que permite facilmente a implementação por terceiros, como por fornecer um bom suporte a escalabilidade com baixo esforço e contando apenas com o conhecimento em Java.

Comparando a segunda e a terceira implementações, percebemos que ambas atendem satisfatoriamente aos quesitos capacidade de extensão por terceiros e escalabilidade. O conhecimento necessário para utilização da técnica diferencia-as por ser necessário, além do conhecimento em Java, conhecimento a respeito do *framework Eclipse RCP* para o uso de Extension Points, o que torna a técnica mais complexa quando comparada a herança, arquivo de propriedades e Reflection.

Porém, para desenvolvedores de aplicações Eclipse RCP, o tempo de aprendizado da técnica de Extension Points do Eclipse pode ser considerado irrelevante, considerando que o conhecimento previamente obtido para a implementação da aplicação servirá de suporte para o aprendizado das técnicas mais complexas do *framework*.

Além disso, apesar do conhecimento necessário sobre o *framework*, para aplicações implementadas com Eclipse RCP que priorizem a escalabilidade de suas features, Extension Points ainda apresenta-se como a melhor solução pelas facilidades que ele apresenta, principalmente no que se refere ao suporte para a troca de variações no produto.

As duas primeiras implementações apresentam quantidades de linhas de código bastante próximas, enquanto que a com Extension points apresenta uma diferença de aproximadamente 200 linhas. A maior quantidade de código porém, não representa nenhum prejuízo para última técnica quando comparada às outras porque o código excedente encontra-se na *factory*, e não requer manutenção para variações novas.

No critério modularidade, tanto a implementação realizada com Herança/Arquivo de propriedades/Reflection como a com Extension Points do Eclipse fornecem uma boa modularidade para as variações. Porém, quando o espalhamento da *feature* é observado, nota-se um pequeno espalhamento nas chamadas à *factory* e aos métodos da classe abstrata ou interface da feature. Entretanto, como a *feature* de Output da TaRGeT é obrigatória, tais fragmentos nunca representarão código inutilizado, já que ele será necessário em toda configuração válida do produto.

```
...
OutputDocument outputDocument = FactoryOutputDocument.getOutputDocument();

        outputDocument.writeTestCaseDataInFile(TestCaseGeneratorController.getInstance()
                .generateTestSuiteFile(testSuiteDir));
...
```

Listagem 6.1 - Código de Output espalhado com Reflection.

Podemos então concluir que, para aplicações Eclipse RCP, nas situações em que a implementação com herança é uma boa solução, Extension Points mostra-se uma opção com maiores benefícios. Isso ocorre porque a técnica utiliza herança e torna desnecessário o uso de técnicas auxiliares para inicialização da classe correspondente à variação.

No caso da feature de Output da TaRGeT, por exemplo, caso a implementação com herança fosse escolhida, seria responsabilidade do programador gerenciar as subclasses ainda, usando Reflection com arquivos de propriedade ou compilação condicional, ou ainda outra técnica. Utilizando Extension Points do Eclipse, isso passa a ser responsabilidade do *framework*, que é quem reconhece as implementações da *feature*, inicializa-as e permite que sejam reconhecidas e utilizadas, caso elas estejam na seleção do produto.

6.2 Implementações de TaRGeT Import Template

Para a análise, a *feature* TaRGeT Import Template foi implementada utilizando as técnicas Extension Points do Eclipse com Aspectos e Extension Points do *Eclipse* isoladamente.

Os próximos tópicos detalharão as duas implementações, ressaltando as vantagens e desvantagens do uso de cada uma delas.

6.2.1 Extension points do Eclipse

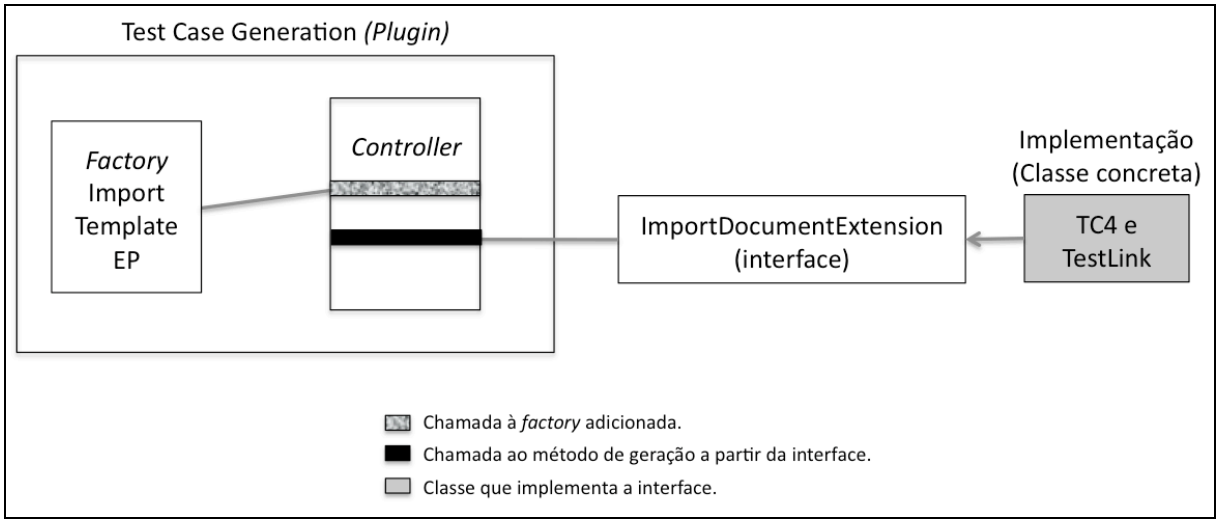


Figura 6.4 - Import Template com Extension Points.

Composição da variação (*Configuration knowledge*)

- Plugin que implemente o extension point;
- target.product' – target.product original com adição da linha que referencia o *plugin* que implementa o *extension point* do Import Template (ou, para adicionar a funcionalidade ao produto gerado, o config.ini' – config.ini original com adição do ID do *plugin* que implementa o *Output*);

Código original afetado

- Adição da variação de Import Template escolhida para a configuração no arquivo de configuração target.product do c*ore* da aplicação;

- Adição do extension point de Import Template ao plugin TaRGeT Test Case Generation, responsável pela geração das suítes de teste;
- Foi adicionada uma chamada à *factory* do extension point de Import Template e ao método de importação antes da geração do arquivo de saída da TaRGeT. A chamada só ocorre caso seja encontrada alguma implementação de Import Template no produto em execução.

Código adicionado

- Uma implementação para TaRGeT Import Template do padrão de saída Test Central 4;
- *Factory* para reconhecer o ponto de extensão implementado e mapear seus atributos (*extension point ID*, *configuration element* e *implementation attribute*) em um objeto cuja classe implementa a interface OutputDocumentExtension.

Vantagens

A implementação com Extension Points torna desnecessário o uso de mecanismos como arquivos de propriedades ou compilação condicional para seleção e inicialização da classe que implementa a classe abstrata ou interface da feature na *factory*. O próprio framework, utilizando as informações de IDs de *plugins* e do ponto de extensão, fornecidas na implementação da *factory*, se encarrega de reconhecer as implementações da classe abstrata presentes no produto e de inicializá-las.

As vantagens e desvantagens observadas na análise da feature *TaRGeT Output* implementada com *extension points* também são observadas nessa *feature*.

Desvantagens

A *feature* Import Template é uma feature opcional, logo o código fonte associado a ela só deve estar presente no produto caso ela seja selecionada. Porém, com o uso de Extension Points do Eclipse, o código fonte da aplicação permanece com fragmentos da *feature* Import Template mesmo quando ela não está mais presente no produto. Isso significa perda de modularidade e espalhamento de código da *feature* analisada.

6.2.2 Extension Points do Eclipse e Aspectos

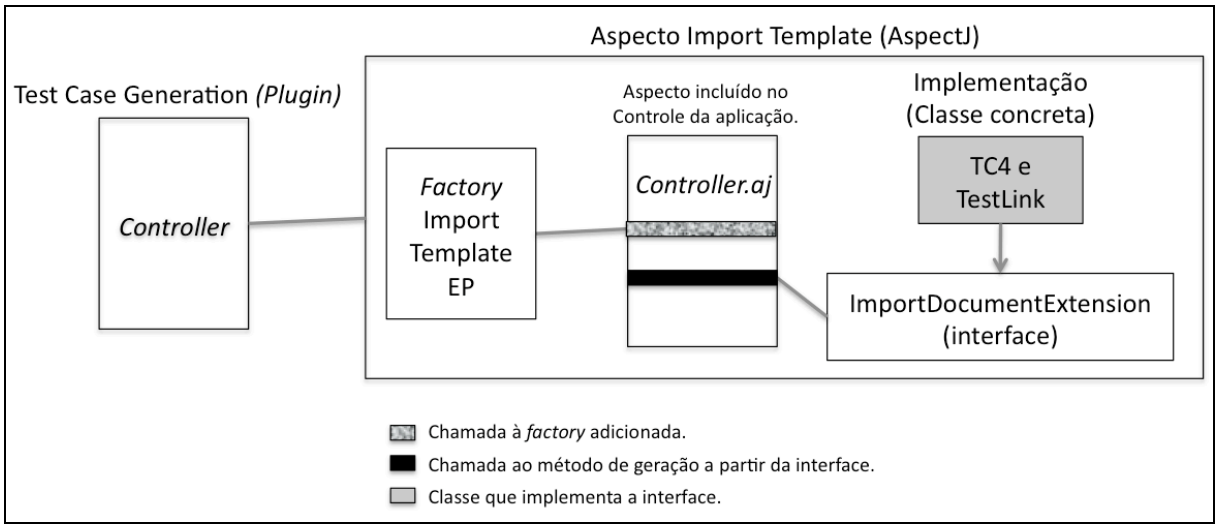


Figura 6.5 - Import Template com Extension Points e POA.

Composição da variação (*Configuration knowledge*)

- *Plugin* que implemente o *extension point*;
- target.product' – target.product original com adição da linha que referencia o plugin que implementa o *extension point* do Import Template (ou, para adicionar a funcionalidade ao produto gerado, o config.ini' – config.ini original com adição do ID do plugin que implementa o Output);
- .classpath' - .classpath original com adição da linha que adiciona o aspecto definido para a feature ao projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
        <classpathentry kind="src" path="src/java"/>
        <classpathentry kind="src" path="src/aspects/com/motorola/btc/research/target/tcg/inputTemplate"/>
        <classpathentry kind="con" path="org.eclipse.jdt.launching.JRE_CONTAINER"/>
        <classpathentry kind="con" path="org.eclipse.pde.core.requiredPlugins"/>
        <classpathentry kind="lib" path="lib/poi-3.2-FINAL.jar"/>
        <classpathentry kind="output" path="build/classes"/>
</classpath>
```

Listagem 6.2 - Adição do aspecto ao projeto.

Código original afetado

- Adição da variação de Import Template escolhida para a configuração no arquivo de configuração target.product do c*ore* da aplicação;
- Adição do *extension point* de Import Template ao plugin TaRGeT Test Case Generation, responsável pela geração das suítes de teste.

Código adicionado

- Aspecto com chamada à *factory* do extension point de Import Template e ao método de importação antes da geração do *output* da *TaRGeT*;
- Uma implementação para Import Template do padrão de saída *Test Central 4*;
- Factory para reconhecer o ponto de extensão implementado e mapear seus atributos (*extension point ID*, *configuration element* e *implementation attribute*) em um objeto cuja classe implementa a interface OutputDocumentExtension.

Vantagens

Todas as vantagens apresentadas pelo uso isolado de *extension points* também são observados nesta implementação.

O desenvolvimento das variações não requer conhecimento em aspectos, pois é necessário apenas implementar a interface do ponto de extensão.

Porém, a combinação entre *extension points* e aspectos proporciona uma melhor modularização da *feature*. Com o uso das duas técnicas combinadas, é possível remover

totalmente o código relacionado à preocupação quando a *feature* opcional está ausente da configuração do produto.

Desvantagens

Para que as chamadas possam ser interceptadas, no momento da criação da variação, podem ser necessários alguns *refactorings* no código original.

É necessário definir *design rules*, do contrário, alterações futuras no código serão arriscadas.

6.2.3 Comparativo das implementações

Como todas as vantagens observadas na implementação com Extension Points também são observadas com Extension Points e POA, foram obtidos resultados bem semelhantes nos critérios avaliados.

Apenas no quesito “Modularidade da *feature*” foram encontrados resultados divergentes, onde a segunda implementação se destacou por prover um melhor encapsulamento da *feature*, permitindo sua remoção total do core da aplicação com baixo esforço, evitando que fragmentos de *features* opcionais permaneçam no código quando elas não estiverem mais presentes.

Critério / Implementação	Extension point do Eclipse	Extension point do Eclipse + AOP
Quantidade de código	4851 linhas	4859 linhas
Capacidade de extensão por terceiros	O produto não precisa ser recompilado para ter a variação adicionada. É necessário apenas conhecer a interface para implementação de variação nova.	O produto não precisa ser recompilado para ter a variação adicionada. É necessário apenas conhecer a interface para implementação de variação nova.
Modularidade da feature	O core da aplicação permanece com rastros da feature mesmo na sua ausência.	Toda a feature é modularizada. Na ausência dela, nenhum código relacionado permanece no produto.
Escalabilidade	Não é necessário alterar o código fonte original.	Não é necessário alterar o código fonte original.
Conhecimento necessário	Conhecimento em Java, Eclipse RCP e seu mecanismo de plugins/extension points.	Conhecimento em Java, AspectJ e Eclipse RCP e seu mecanismo de plugins/extension points.

Tabela 6.2 - Comparativo Extension Points X Extension Points + POA.

6.2.4 Conclusões

A quantidade de código, capacidade de extensão por terceiros e escalabilidade das duas implementações apresentam as mesmas características. Por conta disso, esses critérios não podem ser considerados conclusivos na comparação entre as técnicas.

A segunda implementação, que faz uso de Aspectos, requer maior conhecimento dos desenvolvedores, já que é necessário dominar *pointcuts* e *advices* do AspectJ. Porém, as vantagens provenientes do uso de tal técnica, observadas na avaliação da modularidade da feature, compensam os esforços necessários para seu aprendizado.

O uso de Extension Points deixa rastros no código fonte mesmo quando a *feature* não está mais presente na seleção do produto. O código da chamada à *factory* e aos métodos da interface ou classe abstrata do *extension point* permanece no produto mesmo na ausência da feature.

Para as *features* obrigatórias, em que no mínimo uma das variações precisa estar selecionada para que se tenha um produto válido, o problema descrito acima não é observado, já que sempre haverá uma variação da feature presente no código e as chamadas à *factory* e aos métodos da interface do *extension point* não representarão código inutilizado. A *feature TaRGeT Output* é um bom exemplo disso.

Já para *features* opcionais, o problema é bem recorrente. Conforme analisado na implementação da feature TaRGeT Import Template com *extension points* e aspectos combinados, o código espalhado que fica inutilizado na ausência da feature pode ser facilmente resolvido movendo as chamadas à *factory* e aos métodos da interface para um aspecto que só será adicionado ao *build* do projeto caso a feature esteja presente na seleção do produto.

```
...
if (this.inputTemplateExtension == null)
	{
		Collection<InputDocumentTemplateExtension> inputExtensionsList =
				InputDocumentTemplateExtensionFactory.inputDocumentTemplatetExtensions();

		for (InputDocumentTemplateExtension inputTemplateExtension : inputExtensionsList)
		{
			this.inputTemplateExtension = inputTemplateExtension;
		}
	}
...
```

Listagem 6.3 - Fragmento que permanece no *core* da aplicação.

Dessa forma, concluímos que a combinação de Extension Points com Aspectos permite uma maior modularidade para *features* opcionais por remover o código espalhado presente no *core* da aplicação.

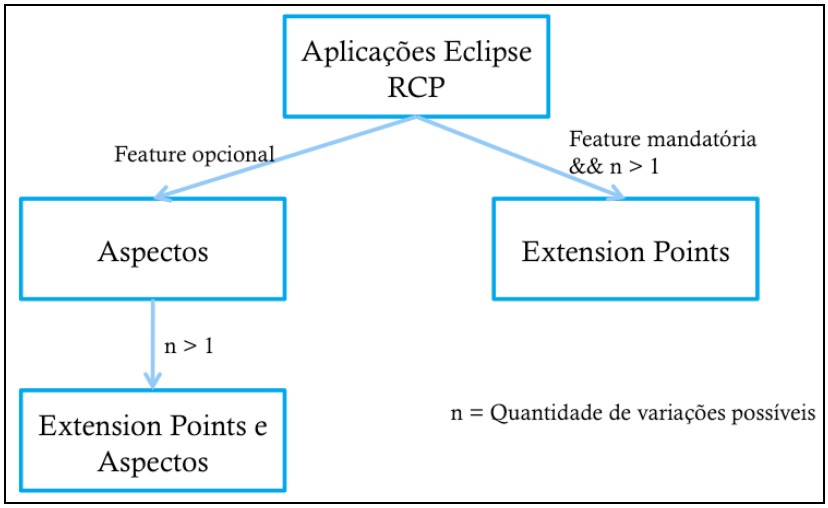


Figura 6.6 - Conclusão da análise EP X EP + POA.

6.3 Implementações de Consistency Management

Para esta feature foram analisadas as implementações com Extension Points e Aspectos.

As implementações serão melhor detalhadas nos próximos tópicos, onde suas vantagens e desvantagens são discutidas.

6.3.1 Implementação com Extension Points do Eclipse

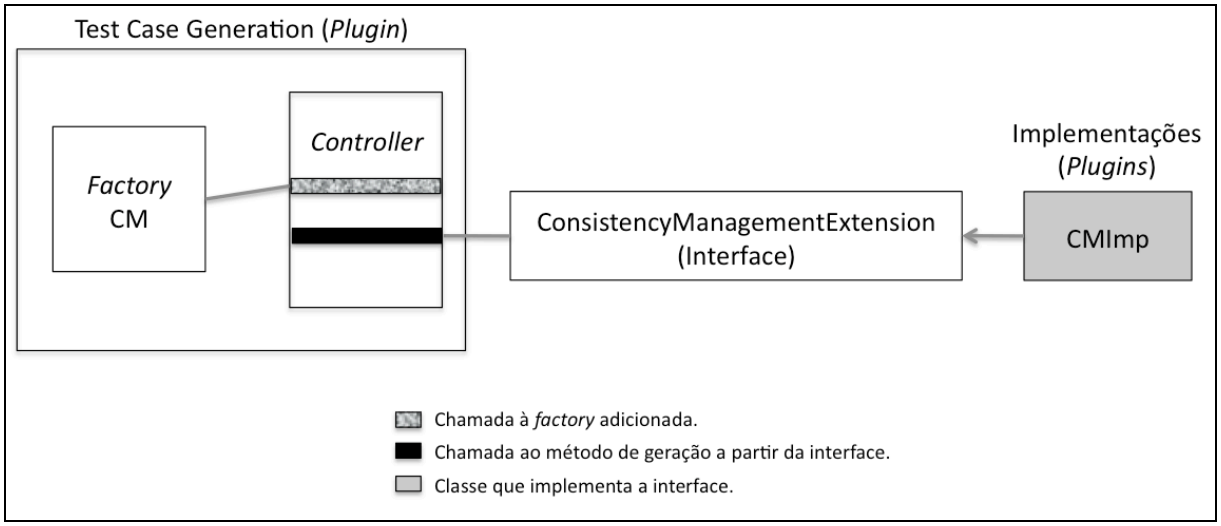


Figura 6.7 - Consistency Management com Extension Points do Eclipse.

Composição da variação (*Configuration knowledge*)

- target.product' – target.product original com adição da linha que referencia o plugin que implementa o extension point de *Consistency Management* (ou, para adicionar a funcionalidade ao produto gerado, o config.ini' – config.ini original com adição do ID do plugin que implementa o *Consistency Management*);
- *Plugin* que implementa o *extension point*.

Código original afetado

- Adição da variação no arquivo de configuração target.product do *core* da aplicação;
- Adição do extension point para Consistency Management ao plugin *TaRGeT Test Case Generation*, responsável pela geração das suítes de teste;
- Um *checkbox* para ativação de Consistency Management foi adicionado ao editor On The Fly. O componente só é inicializado caso o plugin esteja presente no produto.

Código adicionado

- Um plugin novo com a implementação de Consistency Management;

- *Factory* para reconhecer o ponto de extensão implementado e mapear seus atributos (*extension point ID*, *configuration element* e *implementation attribute*) em um objeto cuja classe implementa a interface ConsistencyManagerExtension.

Quantidade de linhas de código (todas as variações): 2110

Vantagens

As vantagens observadas para as implementações com *extension point* do *Eclipse* analisadas anteriormente também se aplicam a essa implementação.

Desvantagens

Com o uso de Extension Points do Eclipse, parte do código fonte da feature Consistency Management permanece no core do produto mesmo quando a feature não está mais selecionada. Com isso há uma perda significativa na modularidade da *feature*.

As desvantagens identificadas nas análises anteriores de implementações com Extension Points do Eclipse também se aplicam a essa implementação.

6.3.2 Implementação com Aspectos

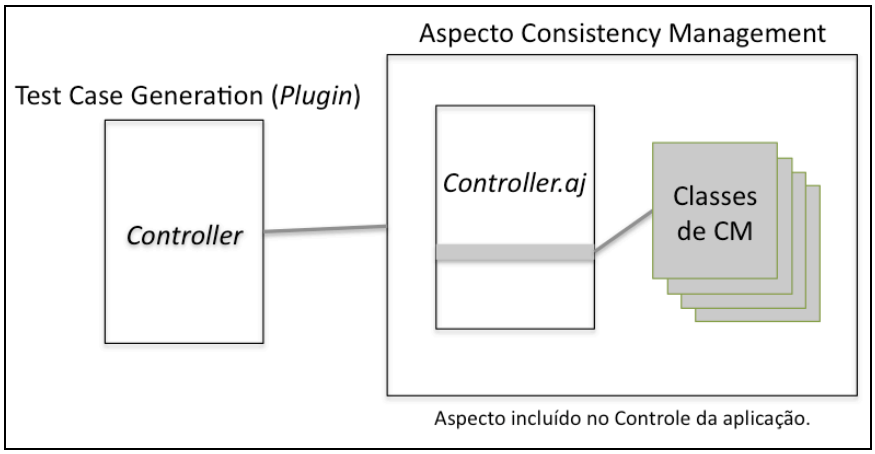


Figura 6.8 - Consistency Management com POA.

Composição da variação (*Configuration knowledge*)

- Adição dos aspectos e classes que encapsulam a preocupação da *feature* Consistency Management.

Código original afetado

- Arquivo de configuração plugin.xml' (TaRGeT Test Case Generation) - Adição de *tags* para o editor de Consistency Management;
- *Refactoring* no editor da geração On The Fly para capturar a chamada da geração da suíte mais facilmente.

Código adicionado

- Aspectos e classes de Consistency Management.

Quantidade de linhas de código (todas as variações): 2125.

Vantagens

Nenhum fragmento de Consistency Management é adicionado ao core da aplicação, pois todo o código da *feature* permanece apenas no aspecto, evitando o espalhamento e entrelaçamento de código.

Desvantagens

É necessário ainda o uso de alguma técnica auxiliar como AspectXML ou compilação condicional para adicionar o editor ao arquivo de configuração plugin.xml.

6.3.3 Comparativo das implementações

Critério / Implementação	Aspectos	Extension point do Eclipse
Quantidade de código	2125 linhas	2110 linhas
Capacidade de extensão por terceiros	É necessário conhecer o código fonte da aplicação detalhadamente.	É necessário apenas conhecer a interface do extension point.
Modularidade da feature	Toda a feature é modularizada. Na ausência dela, nenhum código relacionado permanece no produto.	O core da aplicação permanece com rastros da feature mesmo na sua ausência.
Escalabilidade	Não é necessário alterar o código fonte original.	Não é necessário alterar o código fonte original.
Conhecimento necessário	Conhecimento em Java e AspectJ.	Conhecimento em Java, Eclipse RCP e seu mecanismo de plugins/extension points.

Tabela 6.3 - Comparativo POA X Extension Points do Eclipse.

6.3.4 Conclusões

A quantidade de código e o suporte à escalabilidade de ambas as implementações apresentam as mesmas características. A respeito do conhecimento necessário para as implementações, ambas requerem conhecimento de técnicas específicas. Sendo AspectJ para a primeira e *Eclipse RCP* com seu mecanismo de Extension points para a última. Com isso, esses critérios não são conclusivos na análise das implementações em questão.

A *feature* Consistency Management é uma *feature* opcional e não tende a ter novas variações, o que torna irrelevante a perda da capacidade de extensão por terceiros. Como o ponto de extensão não será implementado por *plugins* novos e não será reusado, o uso de herança (*extension points* para aplicações *eclipse RCP*) não é bem justificado.

Por ser uma *feature* opcional, sua ausência da seleção do produto deveria significar a remoção de seu código fonte do core da aplicação, o que não ocorre com o uso de

extension points do Eclipse, que ainda deixa fragmentos no código mesmo com a *feature* removida.

```
...
ConsistencyManagerExtension cm = ConsistencyManagerExtensionFactor.newConsistencyManagerExtension();

if (cm != null)
{
        this.consistencyManagementcheckButton = new Button(client, SWT.CHECK);
        this.consistencyManagementcheckButton.setText(Properties.getProperty(
                        "start_consistency_management_before_saving"));
}
...
```

Listagem 6.4 - Parte dos fragmentos que permanecem no código.

Com Aspectos, é possível remover do core da aplicação todo o código espalhado e entrelaçado da feature, modularizando-a e permitindo uma melhor manutenção da mesma.

As alterações necessárias nos arquivos de configuração do core para que o editor de Consistency Management seja adicionado a TaRGeT são insignificantes, uma vez que podem ser automatizadas com o uso de compilação condicional ou AspectXML.

Com isso conclui-se que para manter uma melhor modularidade da *feature* e evitar seu espalhamento e seu entrelaçamento de código com o de outras *features*, a implementação com Aspectos é a mais viável.

7. CONCLUSÕES

Neste trabalho foram apresentadas a implementação e a análise da linha de produto da TaRGeT. O desenvolvimento foi iniciado com identificação de requisitos particulares de clientes da ferramenta.

Após uma breve contextualização envolvendo as técnicas atualmente utilizadas na abordagem de LPS e a apresentação da implementação da TaRGeT, foram mostrados os pontos de variação identificados e as variações implementadas a partir de cada um deles.

Por fim, foi apresentada uma análise de implementações alternativas que envolveu não apenas o uso de técnicas isoladas, mas também de combinações de técnicas a fim de propor soluções mais abrangentes e adequadas para a implementação da linha.

7.1 Trabalhos relacionados

Técnicas utilizadas no desenvolvimento de linhas de produto de software são analisadas em diversos trabalhos que buscam a identificação das técnicas mais apropriadas para o desenvolvimento de LPS e a concepção de modelos de decisão que auxiliem os desenvolvedores na implementação de variações.

Em [7], são definidos critérios, parâmetros e diretrizes para a avaliação de implementações de LPS. A partir disso, são feitas comparações entre técnicas utilizadas na indústria de software. Isso ocorre através da avaliação da performance de cada padrão de implementação de acordo com diferentes critérios.

O trabalho compõe um catálogo de tipos de variações, com um conjunto de padrões de implementação de variabilidade (usando técnicas diferentes) para resolver cada tipo de variação, com o objetivo de auxiliar desenvolvedores de software na decisão de qual modelo é o mais adequado para cada caso específico.

Uma abordagem semelhante, porém voltada para a reestruturação de variações em casos de testes em LPS é encontrada em [4]. O trabalho define um modelo de decisão para reestruturação de variações com o objetivo de melhorar a modularidade dos artefatos da

LPS. Para a construção do modelo, variações encontradas em casos de teste automatizados reais desenvolvidos pela Motorola foram analisados.

No trabalho, foi desenvolvida ainda uma ferramenta que sugere os mecanismos a serem utilizados na reestruturação de acordo com o modelo de decisão proposto.

7.2 Trabalhos futuros

Uma análise do aumento ou decréscimo da complexidade da implementação do projeto em função do uso de técnicas de variabilidade combinadas é um possível trabalho futuro. Dessa forma seria possível avaliar a influência do uso de técnicas combinadas para implementação de LPS nos custos de desenvolvimento do projeto.

Uma análise das conseqüências decorrentes do acoplamento ocasionado pelo uso de POA na implementação da LPS da TaRGeT também seria interessante. Com ela, seria possível avaliar mais precisamente como a modularidade e o conseqüente acoplamento decorrentes do uso de POA influenciam os custos de manutenção do projeto.

Por fim, uma análise das variações de Output da TaRGeT que levem em consideração o formato e conteúdo dos casos de teste da suíte de forma isolada, avaliando as vantagens dessa abordagem para a implementação das variações, seria mais um possível trabalho futuro.

REFERÊNCIAS BIBLIOGRÁFICAS

[1]. CLEMENTS, Paul; NORTHROP, Linda. **Software Product Lines: Practices and Patterns.** Boston: Addison-Wesley, 2002, 563 p.

[2]. COHEN, Sholom. **Product Line State of the Practice Report.** Disponível em http://www.sei.cmu.edu/publications/documents/02.reports/02tn017.html, 08/07/04.

[3]. ALVES, Vander; Gheyi, Rohit; Massoni, Tiago; Kulesza, Uirá; BORBA, Paulo; Lucena, Carlos. **Refactoring Product Lines.** Disponível em: http://toritama.cin.ufpe.br/twiki/pub/SPG/AspectProductLine/gpce40-alves.pdf, 18/10/2009.

[4]. RIBEIRO, Márcio. **Restructuring Test Variabilities In Software Product Lines.** Dissertação de mestrado, Universidade Federal de Pernambuco, Recife, Brasil, 2008.

[5]. **Rich Client Platform.** Disponível em: http://wiki.eclipse.org/index.php/ Rich_Client_ Platform, 28/10/2009.

[6]. BELL, Peter. **What's a Software Product Line - and Why Should I Care?** Disponível em: http://www.pbell.com/index.cfm/2009/6/24/Whats-a-Software-Product-Line--and-Why-Should-I-Care, 18/10/2009.

[7]. MATOS, Pedro. **Analisys of Techniques for Implementing Software Product Lines Variabilities.** Dissertação de mestrado, Universidade Federal de Pernambuco, Recife, Brasil, 2008.

[8]. **TaRGeT - Test and Requirement Generation Tool.** Disponível em: http://twiki. cin.ufpe.br/twiki/bin/view/CInBTCResearchProject/ToolTarget, 10.11.2009.

[9]. **BTC Test Research Project.** Disponível em: http://twiki.cin.ufpe.br/twiki/bin/ view/CInBTCResearchProject/WebHome, 10.11.2009.

[10]. **Qualiti Software Processes - Soluções Para o Processo de Construção de Software.** Disponível em: http://www.qualiti.com.br/home.html, 10.11.2009.

[11]. **Instituto Nacional de Ciência e Tecnologia para Engenharia de Software.** Disponível em: http://www.ines.org.br, 10.11.2009.

[12]. **Testlink - Avoid Bugs By Testing.** Disponível em: http://blog.testlink.org/, 10.11.2009.

[13]. Research Project Team - CIn BTC. **TaRGeT Help Contents.**

[14]. VOGEL, Lars. **Eclipse RCP - Tutorial.** Disponível em: http://www.vogella.de/articles/ RichClientPlatform/article.html, 13/11/2009.

[15]. **RCP FAQ.** Disponível em: http://wiki.eclipse.org/RCP_FAQ, 13/11/2009.

[16]. MCAFFER, Jeff; LEMIEUX, Jean-Michael. **Eclipse Rich Client Platform: Designing, coding, and pachaging Java applications** Boston: Addison-Wesley, 2006, 504 p.

[17]. CASTELIANO, André. **Introdução ao Eclipse RCP.** Disponível em: http://www.devmedia.com.br/articles/viewcomp.asp?comp=4352, 13/11/2009.

[18]. GACEK, Cristina; ANASTASOPOULOS, Michalis. **Implementing product line variabilities**. Proceedings of the 2001 symposium on Software reusability: putting software reuse in context, p.109-117, Maio/2001, Toronto, Ontário, Canadá.

[19]. BRAZ, Christian. **Acoplamento fraco X Herança.** Disponível em: http://www.devmedia.com.br/articles/viewcomp_forprint.asp?comp=3714, 13/11/2009.

[20]. **AspectJ - Crosscutting objects for better modularity.** Disponível em: http://www.eclipse.org/aspectj, 13/11/2009.

[21]. VOGEL, Lars. **Eclipse Extension Points and Extensions - Tutorial.** Disponível em: http://www.vogella.de/articles/EclipseExtensionPoint/article.html, 14/11/2009.

[22]. NEHRER, Peter. **Developing Eclipse Plug-ins.** Disponível em: http://www.developer.com/lang/article.php/3552481/Developing-Eclipse-Plug-ins, 14/11/2009.

[23]. **Trail: The Reflection API (The Java™ Tutorials).** Disponível em: http://java.sun.com/docs/books/tutorial/reflect, 15/11/2009.

[24]. Research Project Team - CIn BTC. **TaRGeT 4.0 - A brief tutorial.** Fevereiro/2009.

[25]. **Notes on the Eclipse Plug-in Architecture.** Disponível em: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, 25/11/2009.