



UNIVERSIDADE FEDERAL DO AMAZONAS
INSTITUTO DE COMPUTAÇÃO
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

ARTe - SPL

Framework de Apoio ao Teste de Regressão em Linhas de Produto de Software.

Kariny Marques de Oliveira

Manaus – AM
Setembro - 2014

Kariny Marques de Oliveira

Framework de Apoio ao Teste de Regressão em Linhas de Produto de Software

Monografia de Graduação apresentada ao
Instituto de Computação da Universidade
Federal do Amazonas como requisito parcial
para a obtenção do grau de bacharel em
Sistemas de Informação.

Orientador

Dr. Sc. Arilo Cláudio Dias Neto

Universidade Federal do Amazonas – UFAM

Instituto de Computação – IComp

Manaus-AM

Setembro 2014

Monografia de Graduação sob o título Framework de Apoio ao Teste de Regressão em Linhas de Produto de Software apresentada por Kariny Marques de Oliveira e aceita pelo Instituto de Computação da Universidade Federal do Amazonas, sendo aprovada por todos os membros da banca examinadora abaixo especificada:

Dr. Sc. Arilo Claudio Dias Neto.

Orientador

IComp

Universidade Federal do Amazonas (UFAM)

Dr. Sc. Bruno Freitas Gadelha

Convidado

IComp

Universidade Federal do Amazonas (UFAM)

Manaus-AM, 2 de Setembro de 2014.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

O48f Oliveira, Kariny Marques de
Framework de Apoio ao Teste de Regressão em Linhas de
Produto de Software / Kariny Marques de Oliveira. 2014
64 f.: il.; 29,7 cm.

Orientador: Arilo Cláudio Dias Neto
TCC de Graduação (Sistemas de Informação) - Universidade
Federal do Amazonas.

1. Teste de Regressão. 2. Linha de Produto de Software. 3.
Rastreabilidade. 4. Teste em Linha de Produto de Software. I. Dias
Neto, Arilo Cláudio II. Universidade Federal do Amazonas III. Título

UNIVERSIDADE FEDERAL DO AMAZONAS

Reitora: Profa. Márcia Perales Mendes Silva

Pró-Reitor de Ensino e Graduação: Prof. Lucídio Rocha dos Santos

Diretor do Instituto de Computação: Prof. Ruiter Braga Caldas.

Em homenagem à minha família.

Agradecimentos

À Deus por tornar tudo isso possível.

À minha família pelo constante apoio.

Aos amigos e integrantes do ExperTS (Grupo de Experimentação e Teste de Software) da UFAM, que auxiliaram e forneceram o suporte necessário para a execução desse trabalho, sejam nas revisões ou nos momentos de descontração. E também aos amigos e integrantes do Laboratório de Sistemas Embarcados, foram ótimos momentos de aprendizado que passei com vocês.

Aos amigos que me acompanharam durante a graduação e aos que se juntaram a nossa turma com o tempo, aprendi lições muito valiosas com vocês e também aos colegas do Computação Desplugada I e II, foram momentos muito divertidos.

Agradeço também à Prof^a Odette Passos, que sempre me ajudou no decorrer acadêmico, me ensinando princípios de planejamento e responsabilidade para com as tarefas determinadas e também a não ter medo de pedir ajuda quando necessário.

Aos professores do IComp em geral que ajudaram durante as disciplinas acadêmicas e muitas vezes fora delas com palavras de apoio.

Ao Prof.^o Arilo Claudio pelas orientações e por mostrar muitas vezes a luz que faltava na pesquisa. Ao Prof^o Bruno Gadelha pelo apoio e paciência durante os meus momentos de desespero. Ao Prof^o Raimundo Barreto pelo apoio inicial na vida acadêmica. Ao Prof^o César Melo por ajudar nas atividades necessárias junto a universidade. E à Prof.^a Tayana Conte por sempre ter a palavra amiga e dar o apoio para que isso fosse possível.

“O êxito da vida não se mede pelo caminho que você conquistou, mas sim pelas dificuldades que superou no caminho”.

Abraham Lincoln

“Descobrir consiste em olhar para o que todo mundo está vendo e pensar uma coisa diferente”.

Roger Von Oech

Framework de Apoio ao Teste de Regressão em Linhas de Produto de Software

Autor: Kariny Marques de Oliveira

Orientador: Dr. Sc. Arilo Cláudio Dias Neto

RESUMO

Teste de Software é um dos fatores primordiais para garantir a qualidade de um produto. Quando aplicado a Linhas de Produto de Software (LPS), tem como objetivo validar cada produto gerado pela LPS, levando em consideração seus aspectos comuns e variáveis. Uma LPS pode apresentar grandes mudanças em sua configuração, sendo necessária uma análise dos testes antes propostos para a primeira configuração da LPS, a fim de garantir a qualidade dos produtos gerados pela mesma. Para os testes de LPS, podem ser utilizados testes unitários para cada aspecto comum e aspecto variável da linha, teste de integração para garantir o perfeito funcionamento das características do produto em conjunto e teste de regressão para reaplicar os testes na nova configuração da linha. O Framework de Apoio ao Teste em Linhas de Produto de Software proposto neste trabalho tem como objetivo rastrear os casos de teste especificados para uma LPS e as características afetadas por cada caso de teste. Ao ser inserida uma nova configuração da LPS, são apresentados quais os casos de teste podem ser utilizados para testar os produtos, minimizando o esforço necessário, usando como base o número de novos testes necessários para garantir a qualidade dos produtos, através do reuso.

Palavras-chave: Teste de Regressão, Linha de Produto de Software, Rastreabilidade.

Assistance Framework for Regression Testing in Software Product Lines

Author: Kariny Marques de Oliveira

Advisor: Dr. Sc. Arilo Cláudio Dias Neto

ABSTRACT

Software testing is one of the main strategies to guarantee the quality of software products. When applied to Software Product Lines (SPL), it aims to validate each product generated by a SPL, considering their commonalities and variabilities. A SPL may present several modifications in its configuration among different versions. That makes necessary an analysis of tests proposed to a previous SPL configuration aiming at to assure the quality of its generated products. For testing in SPL, we can apply unit tests for each product line's commonalities and variabilities, integration testing to assure the correct behaviour of all features and regression testing to rerun the tests in the new SPL configuration. The Assistance Framework for Regression Tests in Software Product Lines aims at to track the test cases specified to a SPL and the features affected by test cases. Thus, when inserted a new SPL configuration, it lists which test cases should be used to test the products, minimizing the effort required to test the SPL, based on the number of new tests needed to ensure the quality of products by reusing test cases.

Keywords: Regression Test, Software Product Line, Traceability.

Lista de figuras

Figura 2.1 Gráfico de comparação entre o custo de desenvolvimento para Sistemas Únicos e para LPS (Pohl et al., 2005).	18
Figura 2.3 Ciclo de vida da Engenharia de Linha de Produto de Software (Pohl <i>et al.</i> , 2005).....	19
Figura 2.4 Exemplo de Modelo de features (Kang et al., 1990).....	25
Figura 2.5 Exemplo de Modelo de features com implicação (Kang et al., 1990).....	25
Figura 2.6 Exemplo de Modelo de Features com exclusão (Kang et al., 1990).....	26
Figura 2. 7 Propagação do erro (Avizienis et al., 2004).....	29
Figura 2.8 Níveis de Teste (Utting, M.; Legeard, 2006).....	35
Figura 2.9 Estratégia Produto a Produto (Colanzi TE, Assunção, 2011)	41
Figura 2.10 Estratégia Teste Incremental (Colanzi TE, Assunção, 2011)	42
Figura 2.11 Estratégia RAI (Colanzi TE, Assunção, 2011)	43
Figura 3. 1 Visão geral do framework conceitual	46
Figura 3.2 Diagrama de atividade do framework conceitual	47
Figura 3.3 Exemplo de um <i>feature model</i> no editor eclipse usando fmp.....	49
Figura 3.4 Trecho de arquivo XML gerado pelo fmp.....	49
Figura 3.5 Exemplo de tela de Cadastro de Casos de Teste	50
Figura 3.6 Exemplo de tela de mapeamento	51
Figura 3.7 Exemplo de configuração da LPS no editor eclipse usando <i>fmp</i>	52
Figura 3.8 Trecho de arquivo XML de configuração	53

Lista de tabelas

Tabela 2.1: Características do modelo de <i>features</i> proposto por Kang (1990).....	24
Tabela 3.1: <i>Template</i> utilizado para definição dos componentes de processo.....	49
Tabela 3.2: Atividade da Etapa 1.....	49
Tabela 3.3: Atividade da Etapa 2.....	52
Tabela 3.4: Atividades da Etapa 3.....	52
Tabela 3.5: Atividades da Etapa 4.....	53
Tabela 3.6: Atividades da Etapa 5.....	55

Lista de abreviaturas e siglas

FMP - Feature Modeling Plug-In for Eclipse.

FODA - Feature-Oriented Domain Analysis.

IComp – Instituto de Computação.

LPS – Linhas de Produto de Software.

SPL – Software Product Lines.

SUT – System Under Test.

UFAM – Universidade Federal do Amazonas.

XML – eXtensible Markup Language

SPLE – Engenharia de Linha de Produto de Software

SPLA - Arquitetura da Linha de Produto de Software

OMG - Object Management Group

UML – Unified Modeling Language

Sumário

1 Introdução	14
1.1 Contexto e Descrição do Problema.....	14
1.2 Motivação e Justificativa	15
1.3 Objetivo	15
1.4 Organização do Trabalho	15
2 Referencial Teórico	16
2.1 Linha de Produto de Software.....	16
2.1.1 Motivações para utilização de LPS	17
2.1.1.1 Redução do custo de desenvolvimento.....	17
2.1.1.2 Melhoria da qualidade	18
2.1.2 Engenharia de Linha de Produto de Software	19
2.1.2.1 Engenharia de Domínio.....	20
2.1.2.2 Engenharia de Aplicação.....	21
2.1.3 Modelo de <i>Features</i>	22
2.2 Teste de Software.....	26
2.2.1 Conceitos e terminologias de Teste de Software.....	28
2.2.2 Técnicas de Teste.....	30
2.2.2.1 Teste de Caixa Branca	30
2.2.2.2 Teste de Caixa Preta.....	32
2.2.3 Níveis de Teste.....	34
2.2.3.1 Teste de unidade.....	35
2.2.3.2 Teste de Integração.....	36
2.2.3.3 Teste de Sistema.....	37
2.2.3.4 Teste de aceitação	39

2.2.4 Teste de regressão.....	39
2.3 Testes em Linha de Produto de Software	40
2.3.1 Produto a produto (<i>Product by Product</i>).....	40
2.3.2 Incremental	41
2.3.3 <i>Reusable Asset Instantiation</i>	43
Capítulo 3 - Framework conceitual de Apoio ao Teste de Regressão em Linhas de Produto de Software.	45
3.1 Concepção do Framework Conceitual.....	45
3.2 Detalhamento das etapas do framework conceitual.....	47
3.2.1 Gerar Modelo de Features da LPS.....	48
3.2.2 Gerar Casos de Teste da LPS.....	49
3.2.3 Definir Relação de Casos de Teste com Features	51
3.2.4 Definir Nova Configuração da LPS.....	51
3.2.5 Diagnosticar Casos de Teste	53
4 Conclusões e Perspectivas Futuras	54
4.1 Considerações finais	54
4.2 Trabalhos Futuros.....	54
Referências	56
APÊNDICE A – Mapeamento CT x Features.....	59
APÊNDICE B – Relatório de Diagnóstico de Casos de Teste	60

1 Introdução

Neste capítulo serão apresentados o contexto e a descrição do problema, o que motivou essa pesquisa, os objetivos e a organização desse trabalho.

1.1 Contexto e Descrição do Problema

Linha de produto de software (LPS) é uma abordagem que tem como principal objetivo promover a geração de produtos similares, porém específicos, com base na reutilização de uma infraestrutura central, reduzindo o tempo de desenvolvimento e manutenção, e aumentando a produtividade (Denger e Kolb, 2006).

Assim, utilizando a abordagem de LPS pode-se modelar em uma única vez um conjunto de possíveis produtos a serem desenvolvidos. Uma razão essencial para a introdução de engenharia de LPS é a redução de custos (Pohl et al., 2005). No entanto, assim como ocorre em qualquer contexto da Engenharia de Software, estes produtos requerem qualidade. Para garantir a qualidade desse conjunto de produtos, uma das estratégias mais eficientes seria teste de software.

Teste de Software é um dos fatores primordiais para garantir a qualidade de um produto. Segundo Myers (2004), testar um software é um processo de executar um programa com o objetivo de encontrar defeitos. Teste de software é um processo, ou um grupo de processos, definidos para garantir que um código faz o que ele foi projetado para fazer, e não faz nada que não foi especificado para fazer.

Quando o teste é aplicado à LPS, ele tem como objetivo validar cada produto gerado pela linha, levando em consideração seus aspectos comuns e variáveis. Uma das grandes dificuldades no teste de LPS é a grande quantidade de casos de teste gerados. Uma das formas mais utilizadas para garantir a qualidade desses produtos é o teste unitário. Porém, dependendo das características selecionadas da LPS, nem todos os casos de teste precisam ser utilizados para testar uma série de produtos gerados pela

seleção, o que diminuiria o esforço necessário nos testes. Muitas vezes os produtos não são testados corretamente, o que reduz a qualidade e a confiabilidade nos produtos originados de LPS.

1.2 Motivação e Justificativa

A implementação de teste em LPS é algo complexo devido ao grande número de variabilidades entre os produtos. Podem ser gerados casos de teste que podem ser aplicados em uma ou mais funcionalidades, porém nem todos devem ser utilizados para testar uma série de produtos gerados por uma configuração. Assim, utilizando o conceito de reusabilidade, seria possível reduzir o esforço necessário para testar um conjunto de produtos.

1.3 Objetivo

O objetivo geral deste trabalho é: “Apresentar um Framework de Apoio ao Teste de Regressão em Linhas de Produto de Software”. Sendo assim, o objetivo geral decompõe-se nos seguintes objetivos específicos:

- a) Analisar as estratégias disponíveis na literatura técnica para testes em LPS.
- b) Prover um framework, que a partir da lista de *features* x casos de teste, sugira casos de teste a serem aplicados aos produtos gerados pela nova configuração da LPS.

1.4 Organização do Trabalho

Este trabalho está organizado em 5 capítulos, sendo o capítulo 1 a introdução, onde é apresentada uma visão geral do trabalho assim como seus objetivos. O capítulo 2 é o referencial teórico, neste é apresentado as teorias necessárias para o desenvolvimento do trabalho. O capítulo 3 descreve o Framework conceitual apresentado no trabalho. Por fim o capítulo 4 mostra os próximos passos a serem desenvolvidos através deste trabalho.

2 Referencial Teórico

Neste capítulo serão apresentadas as definições de Linha de Produto de Software e Teste de Software, que consistem nas duas principais áreas de pesquisa abordadas nesse trabalho. Além disso, são apresentadas as técnicas de teste em Linhas de Produto de Software.

2.1 Linha de Produto de Software

Uma linha de produto de software (LPS) é um sistema intensivo de software que compartilha um conjunto comum e gerenciado de funcionalidades que satisfazem à necessidade específica de um segmento particular do mercado ou uma missão, sendo desenvolvido a partir de um conjunto comum de artefatos de forma planejada (Clements e Northrop, 2001).

A abordagem de linha de produto de software possibilita às organizações explorar semelhanças entre seus produtos, aumentando, assim, a reutilização de artefatos. Como consequência, tem-se uma diminuição dos custos e tempo no desenvolvimento (Heymans e Trigaux, 2003). Em LPS cada característica de um produto é chamada de *feature* e uma seleção de algumas dessas *features* para geração de um produto específico é chamada de configuração.

Durante o processo de desenvolvimento de uma LPS, são identificados e descritos os produtos criados pela linha de produto através das diferenças pelas funcionalidades que proveem, os requisitos que atendem e, até mesmo, da sua infraestrutura de arquitetura (Clements e Northrop, 2001). Esta flexibilidade é chamada de variabilidade e constitui a base da engenharia da linha de produto de software, pois esta representa a diferença em cada produto gerado pela linha de produto de software.

Segundo Pohl et al. (2005), a engenharia de linha de produto é um paradigma de desenvolvimento de software (sistemas de software intensivos e produtos de software) que utiliza uma infraestrutura central e customização em massa. Para isso, a infraestrutura central é o plano estratégico para o desenvolvimento de artefatos reutilizáveis, juntamente

com a sua utilização. Neste contexto, a customização em massa significa a utilização de variabilidade gerenciável, no qual as diferenças e similaridades da aplicação devem ser modeladas de maneira única.

2.1.1 Motivações para utilização de LPS

Segundo Pohl et al. (2005), a principal razão para a engenharia de LPS é prover produtos customizados à um preço razoável se comparado ao desenvolvimento tradicional. As principais motivações, para o desenvolvimento de software utilizando a abordagem de linha de produto de software são descritas nas subseções a seguir.

2.1.1.1 Redução do custo de desenvolvimento

Uma razão essencial para a introdução da engenharia de LPS é a redução de custos. A partir da utilização de uma infraestrutura central em diferentes sistemas, temos a redução de custo em cada sistema.

A Figura 2.1 apresenta o custo acumulado de desenvolvimento de acordo com o número de produtos, comparando a abordagem de LPS com a abordagem de desenvolvimento tradicional. A linha sólida apresenta o custo de desenvolvimento de sistemas independentes, enquanto a linha tracejada apresenta o custo para uma linha de engenharia de produto. No caso de um pequeno número de sistemas, os custos de uma LPS são relativamente altos, porém são significativamente menores para uma grande quantidade de sistemas.

Neste, caso, é possível notar que inicialmente temos um custo maior para o desenvolvimento de uma LPS. No entanto, a partir de um certo momento, estes custos se igualam. No caso, avaliações experimentais sugerem que este ponto é de aproximadamente três sistemas (Pohl et al., 2005). Esta métrica, porém, depende das características da organização, a experiência da equipe, base de usuário, mercado e variedade e tipo de produtos.

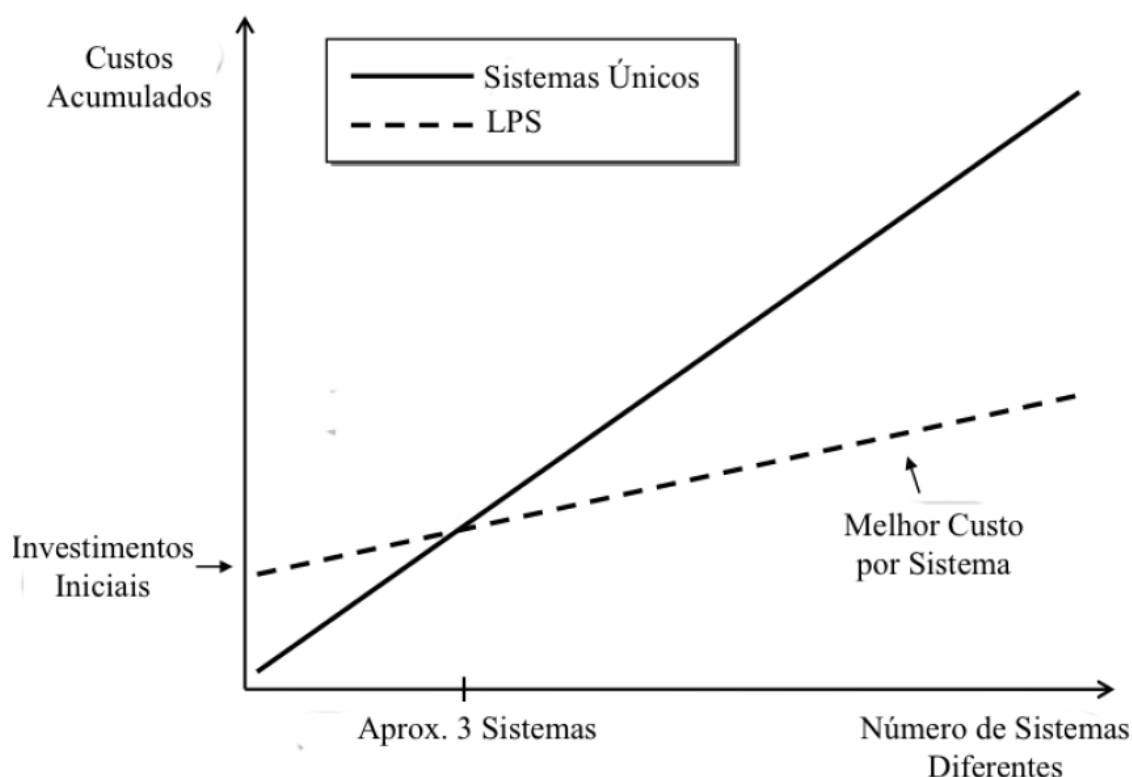


Figura 2.1 Gráfico de comparação entre o custo de desenvolvimento para Sistemas Únicos e para LPS (Pohl et al., 2005).

2.1.1.2 Melhoria da qualidade

Os artefatos da infraestrutura central da LPS são revisados e testados em uma série de produtos. Eles têm que apresentar um funcionamento correto em mais de um tipo de produto. Isto implica em uma maior chance de detectar falhas e corrigi-las e, dessa maneira, melhorar a qualidade de todos os produtos.

Técnicas usadas em linhas de produto de software contribuem para a diminuição dos erros, principalmente pela propriedade de reuso de código. Ao reusar código, a correção de um único defeito pode ter impacto em todos os produtos que utilizam esse componente de código, não sendo então necessário corrigir vários erros. Isso implica em uma menor necessidade de suporte aos clientes, ciclos de testes menores e menor atividade de correção de erros.

2.1.2 Engenharia de Linha de Produto de Software

A engenharia de software tradicional está focada em projeto (*design*) e desenvolvimento de apenas um sistema de cada vez e com algumas reutilizações oportunistas de trechos do código. Por outro lado, a Engenharia de Linha de Produtos de Software (SPLE – *Software Product Line Engineering*) é uma abordagem distinta, que é focada na reutilização de todos os ativos desenvolvido durante o ciclo de vida de desenvolvimento do software, tais como os requisitos, arquitetura, código fonte e artefatos de teste, para gerar um conjunto de sistemas relacionados. O conjunto de todos estes ativos reutilizáveis compõem a infraestrutura da LPS. Outra diferença fundamental é que a SPLE é baseada em dois ciclos de vida separados: Engenharia de Domínio e Engenharia de Produto/Aplicação (ver Figura 2.2) (Pohl et al., 2005).

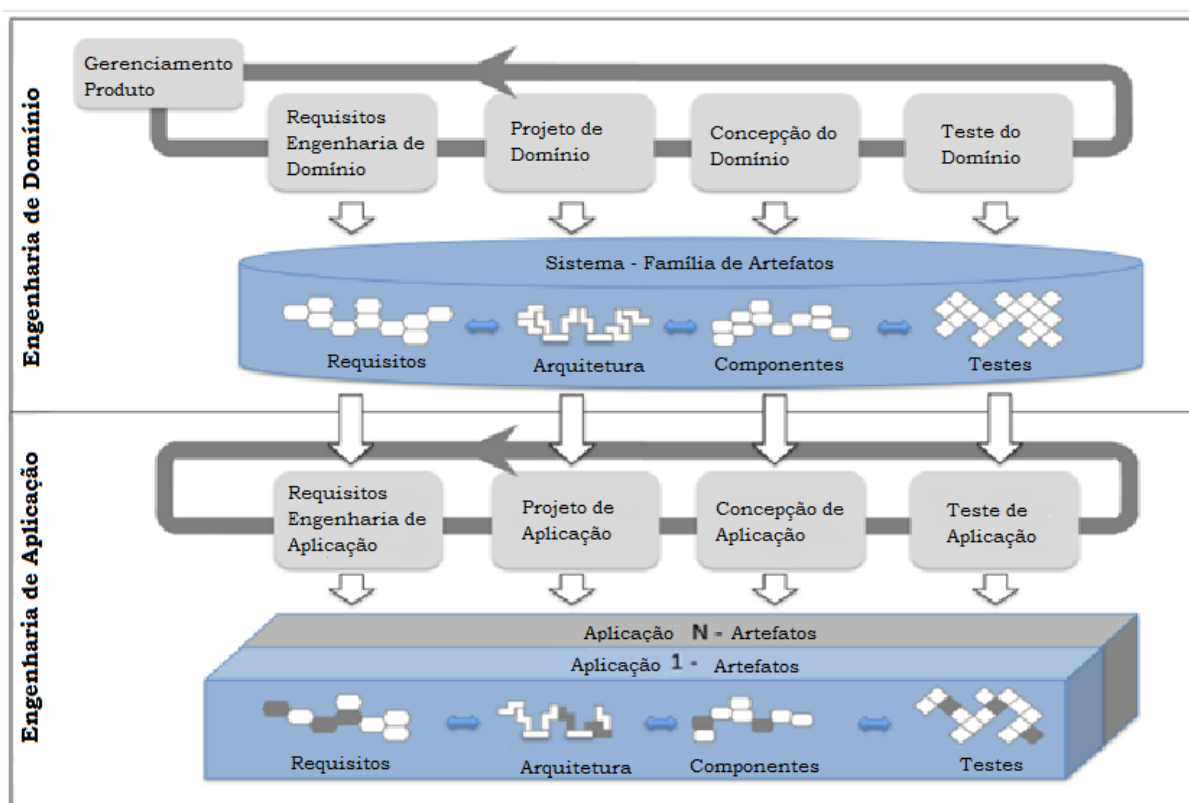


Figura 2.2 Ciclo de vida da Engenharia de Linha de Produto de Software (Pohl et al., 2005)

2.1.2.1 Engenharia de Domínio

O ciclo de vida da Engenharia de Domínio está focado em definir o escopo da LPS e gerar todos os bens comuns, como os requisitos de teste, que compõem a plataforma da LPS. Assim, baseado nesses bens comuns é definida a plataforma de variabilidade para apoiar a geração de diferentes sistemas. Em seguida, será abordada uma breve introdução dos cinco sub-processos que compõem o ciclo de vida da Engenharia de Domínio (Pohl et al., 2005) (representados pelas cinco maiores caixas cinzas na Figura 2.2):

- **Gerenciamento de Produtos:** os principais objetivos do sub-processo de gestão de produtos são a definição do âmbito de aplicação da LPS e a gestão do portfólio de produtos da organização. Os resultados desse processo são o roteiro de produtos, o cronograma para liberar cada produto, a definição de todos os produtos gerados e a lista de todos os artefatos reutilizáveis (Kang et al., 1990).
- **Requisitos de Domínio:** a engenharia de requisitos usa o roteiro de produtos e os requisitos de cada sistema desejado para definir e documentar os requisitos comuns e variáveis da LPS. Para atingir essas metas, cinco fases são executadas: *Elicitação*, que é a análise das necessidades dos usuários; *Documentação*, onde as necessidades dos usuários são formalizadas em documentos textuais ou modelos; *Negociação*, onde as exigências são discutidas e negociadas com as usuários/as partes interessadas; *Verificação e Validação*, nesta fase o objetivo é verificar se os requisitos estão corretos e compreensíveis; *Gerenciamento*, seu objetivo é manter e avaliar as mudanças nos requisitos durante o ciclo de vida da linha de produtos. Com o resultado dessas fases, tem-se o conjunto de variáveis comuns e requisitos bem definidos e o modelo de variabilidade da LPS.
- **Projeto do Domínio:** o foco principal deste sub-processo é usar os requisitos de domínio e modelo de variabilidade da LPS para definir a Arquitetura da Linha de Produto de Software (SPLA – *Software Product Line Architecture*).

- **Concepção do Domínio:** na concepção do domínio, a SPLA e os ativos de domínio desenvolvidos previamente são utilizados para projetar e implementar os componentes de software do domínio.
- **Teste de Domínio:** durante o teste de domínio, os componentes desenvolvidos na realização de domínio são testados em sua especificação (por exemplo, requisitos e SPLA). Para reduzir o tempo e esforço durante o teste de domínio, os artefatos de teste desenvolvidos para testar os componentes também são reutilizáveis e podem ser utilizado para testar um novo ou um componente modificado.

Como mostrado na Figura 2.2, a Engenharia de Domínio gera e gerencia um conjunto de bens comuns, que por sua vez são utilizados na engenharia de aplicação para gerar os produtos da LPS. Por outro lado, o sub-processo de engenharia de aplicação proporciona feedback que pode ser usado para guiar as alterações e/ou o desenvolvimento de novos ativos na Engenharia de Domínio (Pohl et al., 2005).

2.1.2.2 Engenharia de Aplicação

No ciclo de vida da engenharia de aplicação, os bens comuns e variáveis desenvolvido na Engenharia de Domínio são combinados com ativos específicos para criar um produto da LPS. A engenharia de aplicação é composta por quatro sub-processo, que recebem como entrada a sua contraparte na Engenharia Domínio para gerar artefatos de aplicativos (Pohl et al., 2005). O sub-processo da engenharia de aplicação é brevemente introduzido abaixo:

- **Requisitos de Aplicação:** o sub-processo de requisitos de aplicação usa os requisitos de domínio e o roteiro da linha de produto, além de requisitos específicos do produto de destino, para gerar os requisitos de um produto específico.
- **Projeto de Aplicação:** durante o projeto de aplicação, o SPLA é usado para instanciar a arquitetura de um produto específico. Portanto, as partes desejadas são selecionadas (os pontos de variação são

decididos) e algumas adaptações relacionadas ao produto específico são adicionadas.

- **Concepção da aplicação:** o foco principal da concepção da aplicação é usar a arquitetura do produto, os componentes de concepção de domínio e a concepção dos componentes do produto específico para gerar um produto executável.
- **Teste da aplicação:** como no processo de desenvolvimento de software tradicional, os produtos gerados a partir de uma LPS também devem ser testados. Embora os componentes individuais geralmente sejam testados no teste de domínio, o produto completo deve ser testado durante a fase de teste da aplicação, porque é quase impossível testar todas as combinações de componentes e alguns componentes específicos podem ter sido adicionados. Nos últimos anos, vários trabalhos que apresentam abordagens, técnicas e ferramentas relacionadas com testes de LPS podem ser encontrados na literatura técnica (Olimpiew e Gomaa, 2005)(Engström e Runeson, 2011)(Lee et al., 2012).

2.1.3 Modelo de *Features*

A abordagem F-O-D-A (*Feature Oriented Domain Analysis*) é um conceito orientado a funcionalidades, baseando-se na ênfase colocada no método de identificação das funcionalidades que um usuário comumente espera no domínio da aplicação (Kang et al., 1990). Esta análise do domínio é parte da análise de requisitos e do projeto de alto nível. Ela engloba uma família de produtos em um domínio, ao invés de um sistema único, produzindo um modelo de domínio parametrizável para identificar as diferenças e a arquitetura padrão para o desenvolvimento de componentes. A abordagem é dividida nas etapas de Análise de Contexto, Modelagem de Domínio e Modelagem de Arquitetura (Kang et al., 1990):

- A etapa de análise de contexto consiste na iteração com usuário e especialistas do domínio para definir o escopo a ser analisado.

- A modelagem do domínio consiste na utilização das informações do usuários e dos produtos da análise de contexto para a criação do domínio do modelo.
- A modelagem da arquitetura consiste na utilização do domínio do modelo para o desenvolvimento do modelo de arquitetura da aplicação.

A etapa de modelagem do domínio apresenta os seguintes produtos: modelo de entidade relacionamento, modelo de *features*, modelo funcional e dicionário de terminologia do domínio.

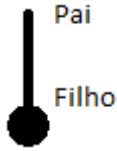
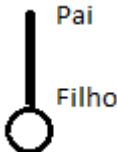


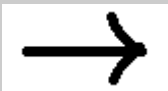

Como o interesse primário é identificar as semelhanças da aplicações de uma família de sistemas, será utilizado o modelo de *features* para mostrar as *features* em comum e diferentes da aplicação do domínio. Com isso, as *features* do modelo de *features* são utilizadas para generalizar e parametrizar os produtos.

O conceito de *features* utilizado para o modelo de *features* é de que esta é qualquer aspecto, qualidade ou característica de software perceptível visualmente pelo usuário (Kang et al., 1990). O propósito do modelo de *features* é representar os recursos para a aplicação no domínio.

Modelos de *features* são criados utilizando *features* obrigatórias, opcionais, alternativas e 'ou'. As *features* obrigatórias são aquelas que estão contidas em todos os produtos criados pela linha de produto de software. As *features* opcionais são aquelas que podem ou não estar contidas nos produtos da LPS. As *features* alternativas são aquelas em que duas ou mais *features*, apenas uma delas pode estar contidas nos produtos da LPS. E as *features* 'ou' são aquelas em que dadas duas ou mais *features*, pelo menos umas delas deve estar contida nos produtos de uma LPS. O modelo de *features* também contém elementos de restrições: implicações e exclusões. A notação de implicação representa que quando uma *feature* for selecionada, outra *feature* também deve ser selecionada. A restrição de exclusão indica que duas *features* não podem estar selecionadas em um mesmo produto já que as identifica como mutualmente exclusiva.

A Tabela 2.1 apresenta os relacionamentos, o tipo, a semântica e a notação do modelo de *features*.

Tabela 2.1: Características do modelo de *features* proposto por Kang (1990)

Relacionamento	Tipo	Semântica	Notação
Relacionamento de Domínio	Mandatório	Se a <i>feature</i> pai é selecionada, a <i>feature</i> filha também deve ser selecionada	
	Opcional	Se a <i>feature</i> pai é selecionada, a <i>feature</i> filha pode ser selecionada.	
	Alternativa	Se a <i>feature</i> pai é selecionada, exatamente uma <i>feature</i> filha deve ser selecionada.	
	Ou	Se a <i>feature</i> pai é selecionada, pelo menos uma das <i>features</i> filha deve ser selecionada.	
Dependência	Implicação	Se uma <i>feature</i> é selecionada, a <i>feature</i> implicada deve ser selecionada, ignorando a sua posição na árvore de <i>features</i> .	
	Exclusão	Indica que ambas as <i>features</i> não podem ser selecionadas na mesma configuração de produto e que são mutuamente exclusivas.	

A Figura 2.3 apresenta um exemplo de um modelo de *features* para um carro. Este descreve duas *features* mandatórias: Transmissão e Motor. A primeira é uma *feature* alternativa, e dessa forma, requer que seja selecionada exatamente um modelo de transmissão; neste caso, as *features* manual ou automático. A segunda é a *feature* de motor, a qual estará presente em todos os produtos gerados pela linha de produto de software. Além das *features* mandatórias, o exemplo tem a *feature* Ar Condicionado, que é opcional, de modo que, pode estar ou não nos produtos gerados pela LPS.

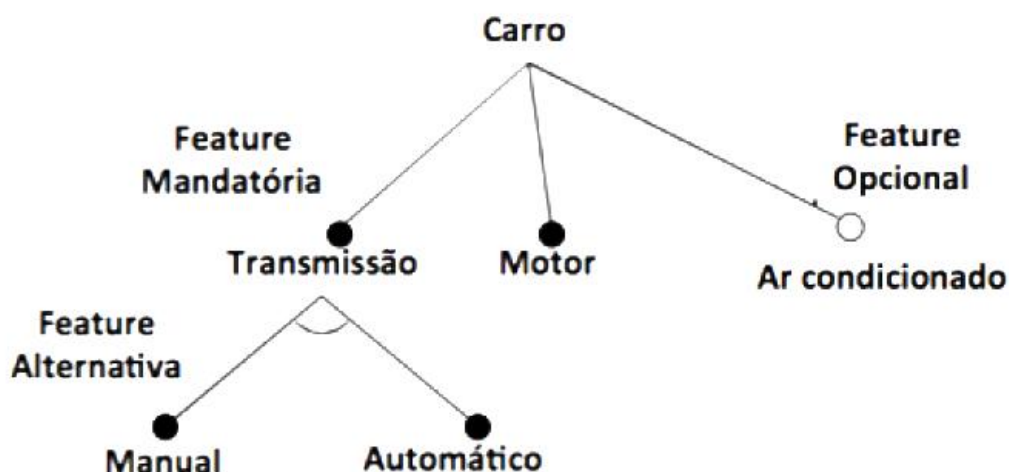


Figura 2.3 Exemplo de Modelo de features (Kang et al., 1990).

A Figura 2.4 apresenta o mesmo exemplo da Figura 2.3, com o acréscimo de um relacionamento de implicação, através do qual, independente da hierarquia das *features*, é possível especificar que toda vez que uma *feature* for selecionada, a outra também deve ser selecionada. Neste caso, é possível observar este comportamento no relacionamento entre a transmissão automática e o ar condicionado, a qual determina que toda vez que a *feature* de transmissão automática for selecionada, a *feature* ar condicionado também deve ser selecionada.

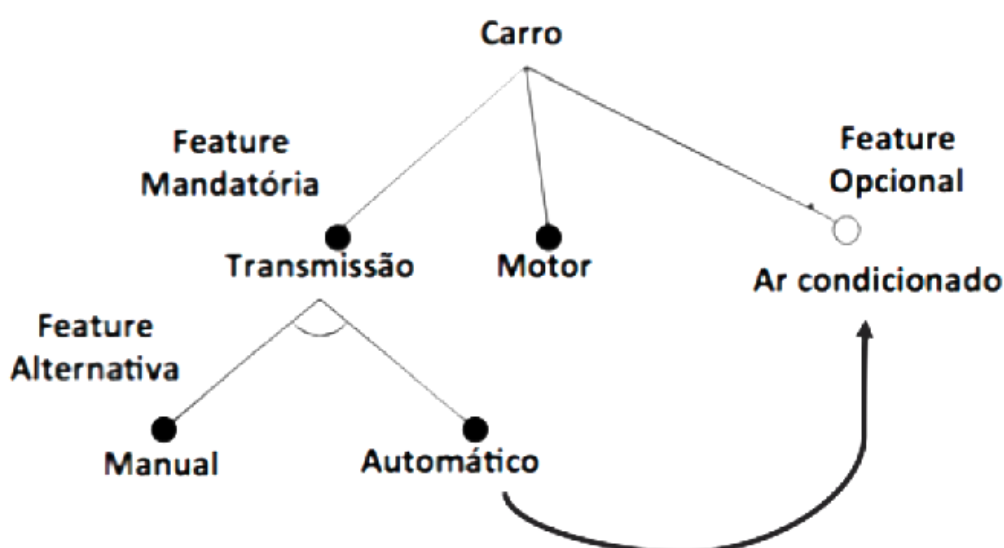


Figura 2.4 Exemplo de Modelo de features com implicação (Kang et al., 1990).

A Figura 2.5 apresenta o mesmo exemplo da Figura 2.3, com o acréscimo um relacionamento de exclusão, através do qual, independente da hierarquia das *features*, é possível especificar que toda vez que uma *feature* for selecionada, a outra não pode ser selecionada. Neste caso, é possível observar este comportamento no relacionamento entre a transmissão manual e o ar condicionado, a qual determina que toda vez que a *feature* de transmissão manual for selecionada, a *feature* ar condicionado não pode deve ser selecionada, ou viceversa.

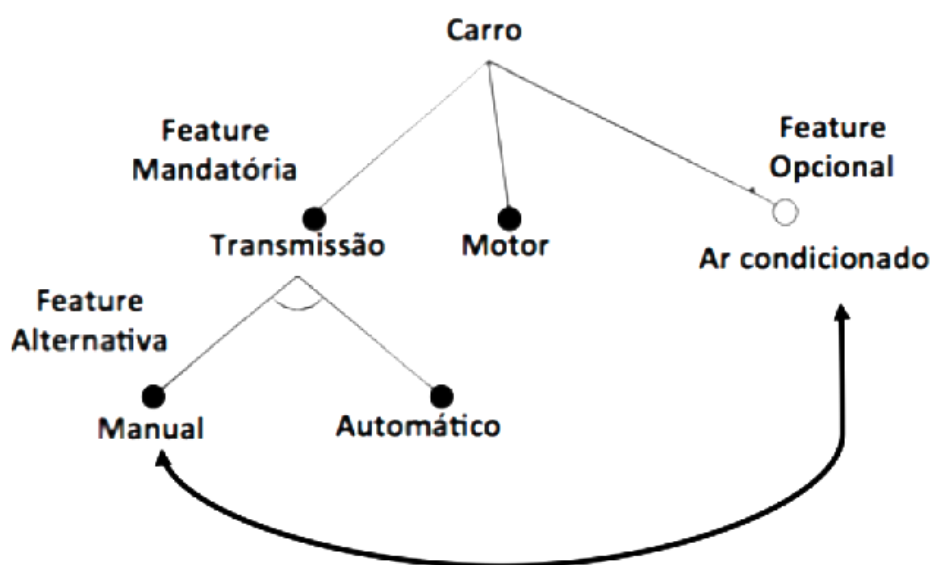


Figura 2.5 Exemplo de Modelo de Features com exclusão (Kang et al., 1990).

2.2 Teste de Software

Hoje em dia, quase todas as atividades humanas tiram proveito do uso de software, desde um sistema simples para se gerenciar uma empresa local a um sistema crítico complexo, tais como aqueles usados para controlar uma usina nuclear, operações financeiras ou de um avião moderno. Independente do sistema, trivial ou complexo, é necessário um certo nível de confiabilidade e segurança. Por esta razão, as empresas de desenvolvimento de software têm tentado chegar a um nível cada vez maior de confiabilidade em seus softwares. Assim, eles estão fazendo um esforço substancial e investimento financeiro para garantir que o software esteja funcionando corretamente. Consequentemente, software de alta qualidade é um ativo muito importante para qualquer organização de desenvolvimento de software.

No entanto, existem diversos atributos que estas organizações devem considerar para garantir que software é de alta qualidade. No trabalho de Avizienis *et al.* (2004), os autores afirmam que a alta qualidade software (confiável) deve abranger seis atributos: disponibilidade, confiabilidade, confidencialidade, segurança, integridade e sustentabilidade. Além disso, Avizienis *et al.* (2004) também classificaram as quatro técnicas principais que pode ser usadas para alcançar confiabilidade em um software:

- *Prevenção de falhas*: esta categoria esta focada em prevenir a ocorrência ou introdução de falhas;
- *Tolerância a falhas*: esta se concentra em evitar falhas de serviço, se uma falha estiver presente;
- *Previsão de falha*: seu foco principal é prever falhas prováveis, para que eles possam ser removidos ou seus efeitos sobre o sistema possam ser minimizados ou contornados;
- *Remoção de falha*: esta categoria se concentra em reduzir o número de falhas, através da detecção de falhas e, em seguida, removendo as mesmas, durante o desenvolvimento de software e utilização.

Apesar de todas estas técnicas serem utilizadas para alcançar a confiabilidade do software, uma das técnicas mais utilizadas no processo de desenvolvimento de software na indústria é a remoção de falhas. Em conformidade com (Avizienis *et al.*, 2004), esta categoria é subdividida em verificação e validação (V&V). A validação é o processo de avaliação do software para garantir o cumprimento do uso pretendido (requisitos) e verificação é o processo de avaliar se o software tem atendido aos requisitos especificados para todas as fases de desenvolvimento de software. A análise estática, teoremas, verificação de modelos, execução simbólica e teste de software são exemplos de técnicas de verificação e validação (Avizienis *et al.*, 2004).

Em adição a outras técnicas de verificação e validação, teste de software é uma das técnicas mais utilizadas, que constituem um elemento-chave para alcançar a confiabilidade em um sistema de software (Everett e Raymond, 2007). O teste de software pode ser definido como o processo de

avaliação sistemática de um sistema por uma execução e observação de teste controlada (Ammann e Offutt, 2008). O objetivo de teste de software pode ser definido como:

Os testes são realizados para avaliar e melhorar a qualidade do produto, identificando defeitos e problemas. O teste de software consiste na verificação dinâmica do comportamento de um programa com um conjunto finito de casos de teste, adequadamente seleccionados entre as execuções geralmente de domínio infinitos, contra o comportamento esperado (Bertolino, 2013).

2.2.1 Conceitos e terminologias de Teste de Software

Nesta seção, apresentamos alguns conceitos relevantes e terminologias que serão usados ao longo deste trabalho. A maior parte desta informação foi extraída do Glossário padrão IEEE de Engenharia de Software Terminologia (IEEE-610, 1990) e IEEE Standard para Documentação de Teste de Software e Sistema (IEEE-Std-829, 2008). No entanto, para manter a coerência com a taxonomia e os conceitos apresentados acima, optou-se por utilizar as definições de falha, erro e defeito apresentados em (Avizienis *et al.*, 2004). Além disso, acredita-se que os conceitos e taxonomia apresentados nesse trabalho são abrangente e estão estruturados de uma forma que torna mais fácil de ler e entender. Os principais conceitos e terminologias utilizados neste trabalho são:

- **Defeito:** este é um defeito estático inserido em um software, geralmente durante a sua fase de codificação. Muitas vezes, é causado por erros humanos, tais como, erros com digitação, falta de conhecimento, e má interpretação dos requisitos. No caso de um defeito estar presente no sistema, mas não ser executado, o defeito é inativo. Caso contrário, o defeito está ativa e pode provocar um erro (Figura 2.6).
- **Erro:** este pode ser definido como um estado interno incorreto de um sistema. No caso de este estado incorreto levar a um defeito, o erro foi propagado fora e pode resultar numa falha do serviço (Figura 2.6). Se um erro está presente, mas não está ativo e detectado, ele é classificado como um erro latente.

- **Falha:** falha é definida como um evento observável que ocorre quando o real comportamento do sistema desvia do comportamento esperado do sistema (Figura 2.6). Uma vez que o teste de software é composto da verificação dinâmica do comportamento de um programa, ele é focado em detecção de falhas para remover defeitos.



Figura 2. 6 Propagação do erro (Avizienis et al., 2004)

- **Especificação de teste:** é um documento que é usado pelo analista de teste para descrever o ambiente de teste, entradas de teste, condições de execução e comportamento esperado do sistema em teste (SUT). Este documento, muitas vezes é baseada em um acordo entre o cliente e o desenvolvimento / testadores.
- **Caso de teste:** é um conjunto de entradas de teste e condições de execução, com o objetivo de exercer um sistema particular (ou parte de um sistema) e, em seguida, monitorar o comportamento do sistema para verificar se ele esta de acordo com as suas necessidades.
- **Caso de teste abstrato:** ele descreve a execução e as condições de teste sem a preocupação com a linguagem de programação ou a ferramenta que irá ser usada para executar o teste. Normalmente, é um primeiro passo para criar um caso de teste concreto. Por exemplo, um caso de teste abstrato para testes de desempenho pode ser um documento textual (por exemplo, Word ou XML), que descreve as atividades que podem ser realizadas por usuários quando eles estão interagindo com o SUT.
- **Caso de teste de concreto:** é um caso de teste abstrato que tem todas as informações concretas necessárias para executar o teste. Normalmente, um caso de teste concreto é escrito em uma linguagem

de programação ou um formato de script. Por exemplo, um arquivo XML que descreve um caso de teste abstrato para teste de desempenho (atividades e os seus parâmetros) pode ser utilizado para criar um conjunto de scripts e cenários de desempenho para ferramenta específica de desempenho, tal como *LoadRunner* (HP, 2013), que por sua vez pode ser executado por uma ferramenta de teste de desempenho. Por exemplo, Costa *et al.* (2012) propõem uma abordagem para gerar scripts de desempenho e cenários de casos de teste abstratos.

- **Suíte de testes:** é um conjunto de casos de teste (normalmente casos de teste concretos).
- **Oráculo de teste:** é um artefato de teste que contém todas as informações sobre o SUT do comportamento esperado. Assim, durante ou após a realização do teste, esta informação é comparado com o comportamento observado do sistema para tentar detectar falhas.

2.2.2 Técnicas de Teste

Com base nos conceitos de teste e terminologia apresentada 2.2.1, nesta seção, serão apresentadas e discutidas as técnicas que podem ser utilizadas para tentar localizar e remover as falhas que podem estar presentes em um software. O teste de software é focado em revelar tantos defeitos quanto possível. Assim, para fazer isso, a maior parte da literatura reivindica o uso de duas técnicas complementares: Caixa-branca e Caixa-preta (Myers , 2004)(Everett e Raymond, 2007).

2.2.2.1 Teste de Caixa Branca

A técnica de teste de caixa-branca, também chamada de teste estrutural, é focada em garantir que os componentes internos de um sistema estejam funcionando corretamente pela análise da sua estrutura interna. Assim, os testes podem tirar proveito do acesso ao código-fonte e então usar o seu conhecimento sobre a estrutura interna do software para gerar casos de teste. Num cenário ideal, os casos de teste gerados cobririam todos os

caminhos do código, incluindo suas condições, loops e o fluxo de dados. No entanto, não é prático num ambiente industrial, uma vez que o tempo que é gasto para executar o teste é, na maior parte dos casos, maior do que o tempo de vida estimado do software. Assim, para ultrapassar esta limitação, o testador deve usar alguns critérios de cobertura (Everett e Raymond, 2007) para gerar casos de teste relevantes. Por exemplo, um critério de cobertura pode ser usado para definir qual percentagem do código fonte do sistema foi exercido durante o teste. Isto significa que o testador pode definir, por exemplo, que pelo menos 60% do código deve ser exercitado. Os seguintes critérios de cobertura pode ser usada para melhorar o processo de teste seleção ao aplicar o teste de caixa-branca (Myers , 2004)(Everett e Raymond, 2007):

- **Cobertura de instruções:** É um critério de cobertura simples, que é focada em determinar a percentagem do código fonte que foi executado durante o teste. O pressuposto é que quanto maior for a cobertura de instrução, menor é a possibilidade de erros presentes no software.
- **Cobertura de ramos:** Seu objetivo é determinar uma porcentagem dos ramos do código fonte (decisões) que foram executadas durante o teste, ou seja, uma decisão IF tem dois ramos (Verdadeiro e falso). Isto significa que em um caso de teste que quer atingir uma cobertura de 60% de um código fonte do sistema que tem 2.000 ramos, o teste deve exercer 1.200 pontos de ramificação. A suposição aqui é a mesma definida para cobertura de instruções: uma maior cobertura de ramos significa que existe menos possibilidade de falhas remanescentes presentes no software.
- **Cobertura de caminhos:** Este critério tem como objetivo determinar o percentual de cobertura de caminhos do código-fonte de um sistema que são executados durante o teste. Everett e Raymond (2007) afirmam que: "um caminho de código fonte é a sequência de instruções do programa a partir da primeira instrução executável através de uma série de aritmética, substituição de entrada/saída, ramificando, e declarações de looping retorno/parada/fim da instrução". O percentual de cobertura é medido a partir da estimativa

total de caminhos do código a ser testado. Assim, se o código tem 1.500 caminhos de código e o testador examina 300 caminhos de código, a cobertura de caminhos é de 20%. A suposição é de que quanto maior a cobertura percentual do trajeto, menor é a possibilidade de falhas ainda presentes no software.

- **Cobertura de loops:** Este critério tem como objetivo determinar o percentual de cobertura de *loops*, tais como DO, FOR, WHILE, presente no código a ser testado após a execução do teste. Assim, durante o teste do sistema, deve ser executado um *loop* zero (o verificador deve tentar evitar a execução de um *loop*), um, $n/2$, n e $n+1$ vezes (n representa o valor limite para parar a condição de *loop*). O objetivo do critério de cobertura de *loop* é encontrar condições de looping inesperadas e inadequadas, adotando uma abordagem semelhante a análise do valor limite para a técnica de Caixa-preta (ver próxima subseção). O percentual de cobertura alcançada é calculada sobre o código de *loops* estimados e supõe-se que uma maior cobertura dos *loops* significan que menos falhas podem estar presentes no código.

2.2.2.2 Teste de Caixa Preta

Teste de caixa preta, também chamado de teste funcional, é baseado em entradas de dados de teste/saídas de dados de teste, é uma técnica de teste voltada para derivar casos de teste a partir da especificação externa do software a ser testado, tais como especificações, requisitos e documentos de projeto. Seu objetivo é induzir o sistema a se desviar de seu comportamento esperado, através da apresentação de insumos externos (simulando atividades diárias de negócios). Diferentemente dos testes estruturais, testes funcionais não estão preocupados com a estrutura interna do software.

Apesar do fato de que a técnica de teste de caixa preta pode ser aplicada para encontrar todos os defeitos introduzidos no software, é impraticável, porque o testador teria que se submeter a cada entrada possível aceita pelo software. Além disso, em geral, é humanamente impossível

apresentar todas as entradas, pois pode ser muito grande ou mesmo infinito (Delamaro et al., 2007). Em seguida, é feita uma breve introdução de alguns critérios de cobertura de caixa preta que podem ser usadas pelo testador, para reduzir o tamanho do conjunto de entrada e para melhorar a possibilidade de casos de teste revelarem a existência de uma falha (Myers, 2004)(Everett e Raymond, 2007):

- **Particionamento em classes de equivalência:** Este critério se baseia na divisão do domínio de entrada do software a ser testado, identificado a partir da especificação, em classes de equivalência válidas e inválidas (subconjuntos de entradas do domínio) de casos de teste que pode ser gerados (Bertolino, 2013). No caso de um valor de entrada de uma classe de equivalência revelar uma falha, espera-se que todas as outras entradas a partir da classe de equivalência encontrem a mesma falha. Com base nisso, o testador pode supor que um valor de entrada de cada classe é equivalente a testar qualquer outro valor de entrada presente na classe. Assim, este pressuposto irá reduzir o tamanho das entradas de teste e, portanto, o esforço e tempo gasto para verificar o software.
- **Análise do valor limite:** Este critério é considerado como um complemento para o critério de particionamento em classes de equivalência, porque ao contrário do anterior que se baseia na escolha de uma entrada aleatória como valor da classe de equivalência, a análise do valor limite está focada em definir entradas de teste que margeiam ao longo dos limites de dados. Alguns autores, como (Everett e Raymond, 2007) e (Myers, 2004), afirmam que a análise de valor limite é importante para testadores porque uma grande quantidade de falhas funcionais ocorrem nesses limites ou próximo a ele. Com base nesta afirmação, testadores pode supor que os casos de teste que se aproveitam da análise de valor limite tem uma maior possibilidade de encontrar falhas do que os casos de teste que não o utilizam.
- **Grafo de causa-efeito:** o particionamento de equivalência e análise de valor limite são critérios bem conhecidos e tem sido usados durante

muitos anos para reduzir a dimensão das entradas de teste e o tempo e esforço para verificar um sistema. No entanto, estes critérios falham em não explorar a combinação de condições de entrada. Para superar essa limitação, o testador pode aplicar o critério de grafo de causa-efeito que estabelece os requisitos de teste com base nas possíveis combinações de entrada. De acordo com Myers (2004), para aplicar o critério do grafo de causa-efeito, o testador deve investigar as possíveis condições de entrada (causas) e possíveis ações (efeitos) do software. Depois disso, um gráfico é construído ligando as causas e os efeitos identificados e então é convertido em uma tabela de decisão a partir da qual os casos de teste são derivados. Uma das principais questões em relação ao grafo de causa-efeito (na verdade, pode ser um problema para todos os critérios funcionais) é que muitas vezes, as especificações do programa são feitas de maneira informal. Assim, os requisitos de teste derivados de tais especificações também são um pouco imprecisas.

- **Experiência (testes exploratórios):** É um critério *ad-hoc* focado em gerar as entradas de casos de teste com base na experiência do testador/desenvolvedor com um código semelhante, linguagem de programação e domínio de aplicação (Burnstein, 2003). Assim, caso os dados de teste de um código semelhante ou até mesmo uma versão anterior for mantida e há algum documento de identificação e mapeamento das falhas, o testador é capaz de, usando uma forma *ad-hoc*, "adivinhar" falhas que podem estar presentes no código. Embora este critério não proporcione as mesmas vantagens dos apresentados anteriormente, é bem conhecido que muitos testadores têm vindo a utilizar o critério da experiência em suas atividades diárias de teste, tal como discutido em (Burnstein, 2003)(Myers, 2004)(Everett e Raymond, 2007).

2.2.3 Níveis de Teste

Nesta seção, apresentam-se os diferentes níveis de teste e como o teste geralmente é realizado durante o processo de desenvolvimento de software,

desde os requisitos e especificações para o sistema completo e sua aceitação pelo cliente. Embora existam muitos modelos de desenvolvimento de software presente na literatura e também trabalhos que apresentam os níveis de teste e discutem sua relação com cada atividade do processo de desenvolvimento de software, tais como (Utting e Legeard, 2006) (Ammann e Offutt, 2008)(Sommerville, 2011)(Naik e Tripathy, 2011), a maioria destes trabalhos define quatro níveis ou fases principais: testes de unidade, testes de integração, testes de sistema e testes de aceitação. Dentre estes somente os testes de sistema e aceitação não podem ser utilizados com a técnica estrutural (ver Figura 2.7).

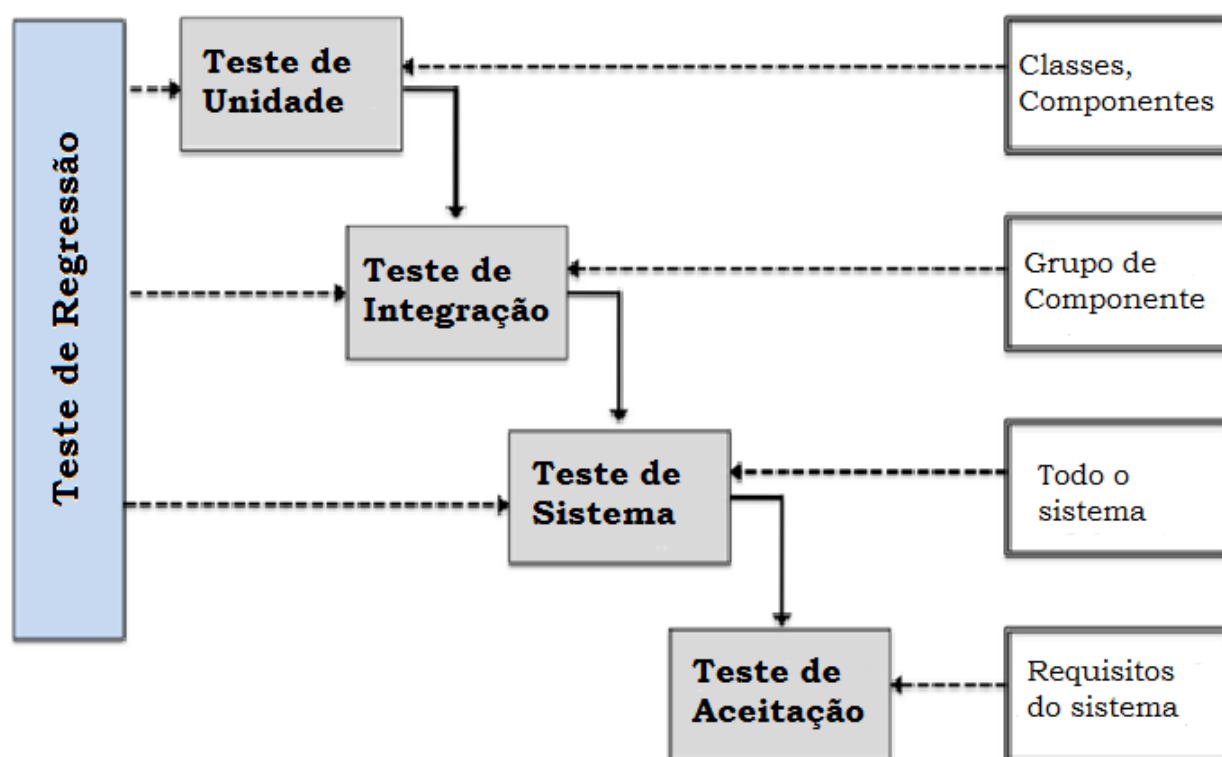


Figura 2.7 Níveis de Teste (Utting, M.; Legeard, 2006)

2.2.3.1 Teste de unidade

O teste de unidade é voltado a testar um pequeno pedaço testável de um sistema, tais como os métodos, classes e funções. Geralmente é realizado pelo desenvolvedor (parte superior da Figura 2.7). Neste nível de teste, o objetivo principal do testador é tentar detectar falhas funcionais e estruturais na unidade de código. Uma vez que a unidade de código sob

teste é normalmente pequena e o testador/desenvolvedor tem um alto conhecimento sobre o código, é bastante simples de desenhar e escrever testes relevantes e também para analisar os resultados. Além disso, um teste de unidade eficiente melhora a possibilidade de que a maioria das falhas mais relevantes serão removidos nos primeiros estágios do desenvolvimento de software, isso significa menor custo e esforço, porque é mais barato e mais fácil de remover falhas durante o teste de unidade do que em qualquer outro nível de teste (Utting, M.; Legeard, 2006).

2.2.3.2 Teste de Integração

O objetivo do teste de integração é verificar a incompatibilidade entre as interfaces e os componentes individuais que já foram testados e aprovados durante o teste de unidade. Apesar de que os componentes já serem testados individualmente durante o teste de unidade, não pode garantir que as interfaces foram implementadas corretamente e também que os componentes funcionarão adequadamente quando combinado com os outros componentes - em alguns casos eles são criados e codificada por um equipe diferente de desenvolvimento. Assim, o foco principal do teste de integração é a integração de um conjunto de componentes individuais em um componente de trabalho enorme ou módulo. Existem algumas abordagens para integrar e testar os componentes, tais como os apresentados em (Naik e Tripathy, 2011):

- **Incremental:** Nesta abordagem, a integração dos componentes é incremental por ciclo. Isto significa que alguns componentes são concluídos, integrados e testados. Em seguida, um novo conjunto de componentes são adicionados e testados. Assim, este ciclo continua até que todo o sistema esteja concluído, integrado e testado.
- **Top-down:** Essa abordagem de integração tem o foco em sistemas onde os componentes devem ter um relação hierárquica. Por exemplo, existe um componente de alto nível que pode ser decomposto em um conjunto de componentes que, por sua vez, pode ser decomposto em outros componentes, e assim por diante. Cada componente que não é

decomposto é um componente terminal. Por outro lado, os componentes que executam algumas operações e invocam os seus componentes decompostos são componentes não-terminais. Em uma abstração de alto nível, a principal idéia por trás dessa abordagem é adicionar um componente de cada vez para o componente de mais alto nível, testar a integração e, em seguida, adicionar outro componente não-terminal. O ciclo deve ser repetido até que o componente agregado seja um componente terminal.

- **Bottom-up:** Ao contrário da abordagem *Top-down*, a abordagem *bottom-up* concentra-se em iniciar a integração dos componentes que são componentes terminais. Para realizar a integração, é necessário desenvolver um *driver* para invocar os componentes que serão integrados (aos componentes terminais). Quando o testador decide que os componentes em teste estão prontos para serem integrados, o *driver* é removido, e outro *driver* é desenvolvido para testar os componentes que serão integrados com os demais já testados. Assim, o processo continua até que todos os módulos tenham sido testados e integrados.
- **Big Bang:** Nesta abordagem, todos os módulos são testados individualmente e são então integrados para construir o sistema inteiro. Depois disso, o sistema é testado como um sistema completo.

2.2.3.3 Teste de Sistema

Após os módulos do sistema serem integrados e testados, é necessário determinar se a implementação do sistema atende aos requisitos do cliente. Portanto, a equipe de testes testa todo o sistema com base em suas necessidades. Motivado pelo fato de que o teste do sistema requer uma quantidade elevada de recursos e é geralmente uma atividade que consome tempo, é normalmente realizada por uma equipe de testes dedicada, pelo menos em médias e grandes empresas. Como existem vários tipos de testes de sistema, as equipes de teste pode ter alguns profissionais altamente qualificados com foco em alguns tipos de teste do sistema. alguns desses tipos são os seguintes:

- **Teste funcional:** Seu objetivo é o de garantir que o comportamento do sistema desenvolvido satisfaça os requisitos. Para realizar o teste funcional, o testador deve submeter entradas válidas e inválidas para o sistema e, em seguida, analisar a sua saída (teste caixa-preta - ver Subseção 2.2.2). Além disso, o testador deve testar todas as funções/métodos do sistema e todos os estados e transições do sistema devem ser exercitados.
- **Teste de desempenho:** O foco principal do teste de desempenho é verificar se o sistema atende os requisitos de desempenho. Por exemplo, um requisito de desempenho pode citar que o o tempo de resposta do sistema deve ser inferior a dois segundos ou que o sistema tem de lidar com até duzentos usuários simultâneos. Assim, a equipe de desempenho pode simular os usuários e, em seguida, usar os resultados do teste para otimizar o desempenho do sistema. Um exemplo de otimização é o ajuste de recursos, tais como, a quantidade de memória disponível no sistema.
- **Teste de stress:** Seu principal objetivo é determinar a quantidade máxima de carga que o sistema pode manipular antes de quebrar, por exemplo, uma carga anormal de usuários. Assim, a equipe de teste utiliza esta informação para assegurar que o comportamento do sistema satisfaça os requisitos, mesmo sob os piores picos de carga suportada (Naik e Tripathy, 2011).
- **Teste de segurança:** É aplicado para verificar se o sistema atende aos requisitos de segurança. Um sistema atende aos requisitos de segurança quando o sistema e seus dados estão protegidos contra acesso não autorizado (confidencialidade), seus dados são protegidos contra modificações não autorizadas (integridade) e quando o sistema e seus dados estão disponíveis para usuários autorizados (disponibilidade).
- **Teste de recuperação:** É realizado a fim de verificar se o sistema se recupera adequadamente de um acidente, tais como falhas de hardware. Normalmente, o testador simula alguma falha de hardware (remoção de um processador), perda de conexão (retirar o cabo de rede)

ou abruptamente reiniciar o computador e, em seguida, analisa a forma como o sistema se recupera e quanto tempo levou para se recuperar.

2.2.3.4 Teste de aceitação

Após o teste do sistema, o sistema está pronto para ser entregue, mas antes ele deve ser aceito pelo cliente. O teste de aceitação é realizado por parte do cliente, e os seu principal objetivo é verificar o comportamento do sistema em relação aos requisitos especificados (Bertolino, 2013). Apesar do teste ser executado pelos clientes, geralmente as equipes de desenvolvimento e testes estão envolvidos na preparação dos mesmos, ajudando-os na execução de atividades e resultados de avaliação. Se o cliente identifica que alguns requisitos não foram satisfeitos, o sistema deve ser corrigido, ou os requisitos podem ser alterados. Caso contrário, o cliente aceita o software e, em seguida, é preparado para ser entregue.

2.2.4 Teste de regressão

O teste de regressão, foco do trabalho, não pode ser considerado como um nível de teste, uma vez que é realizado ao longo do processo de teste de software, como uma sub-fase dos níveis de Unidade, Integração e do sistema (veja a Figura 2.7). Ele é executado após as equipes de desenvolvimento/testes incluírem as melhorias funcionais ou reparos para o sistema. O propósito dos testes de regressão é de assegurar que a versão modificada do sistema mantenha as mesmas funcionalidades e para assegurar que nenhuma falha foi introduzida durante a modificação do sistema (Burnstein, 2003). Outra particularidade dos testes de regressão é que não é necessário projetar e escrever novos testes. Em vez disso, os testes são selecionados entre um conjunto de testes, que já foram projetados, desenvolvidos e executados. Além disso, como o teste de regressão representa um esforço substancial durante o processo de teste, geralmente apenas alguns testes são selecionados e executados. No entanto, o testador deve precisamente definir quantos testes devem ser escolhidos para melhorar a possibilidade de detecção de falhas (Jeffrey e Gupta, 2007)(Li *et al.*, 2007).

2.3 Testes em Linha de Produto de Software

Quando o teste é aplicado à Linhas de Produto de Software tem como objetivo validar cada produto gerado pela linha, levando em consideração seus aspectos comuns e variáveis. Uma das principais falhas em validações de produtos em Linhas de Produto é a grande quantidade de variabilidades da Linha inviabilizando as formas convencionais de teste de software, tendo em vista o crescimento exponencial de número de casos de teste para um conjunto de elementos da Linha de Produto de Software. Muitas vezes os produtos não são testados corretamente o que reduz a qualidade e a confiabilidade nos produtos originados de LPS.

Para o teste em LPS foram desenvolvidas diversas técnicas, as mais conhecidas são: Produto a produto (*Product by Product*), Incremental e *Reusable Asset Instantiation (RAI)*.

A partir do conhecimento da percentagem de comunalidades e variabilidades do conjunto de produtos é possível definir qual a melhor técnica a ser aplicada, podendo também serem aplicadas em conjunto.

2.3.1 Produto a produto (*Product by Product*)

Na estratégia de Produto a Produto, cada produto é testado individualmente, sem a reutilização de ativos de teste (ver Figura 2.8). Esta estratégia facilita a gestão (Jaaksi A, 2002) e garantia de qualidade, pois o teste é focado em apenas um produto por vez. Um primeiro produto, Pr1 é testado individualmente, com um conjunto de dados de teste T1, em T1 há alguns dados de teste que são comuns para os outros produtos Pr2, Pr3 e assim por diante, no entanto, eles não são usados para os produtos posteriores. Os outros produtos são testados individualmente com definem os testes T2, T3, ..., Tn.

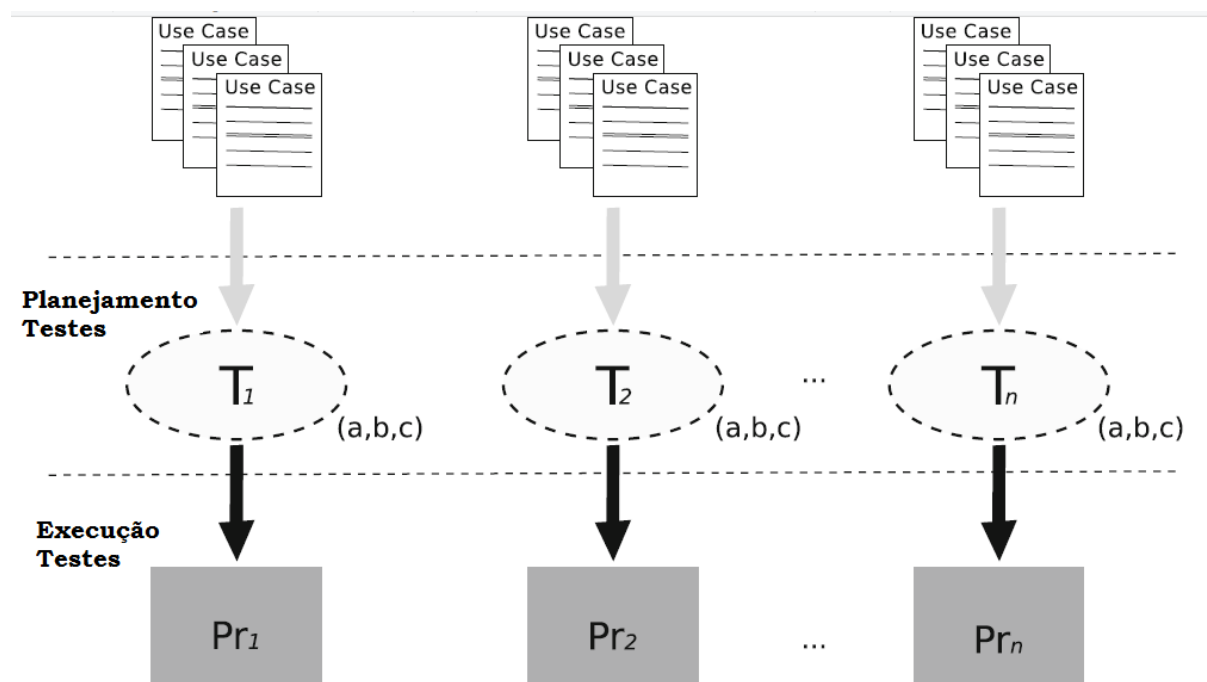


Figura 2.8 Estratégia Produto a Produto (Colanzi TE, Assunção, 2011)

2.3.2 Incremental

A estratégia incremental explora as semelhanças entre produtos através da reutilização sistemática de recursos de teste para testes de LPS. Neste contexto, o primeiro produto da LPS é individualmente testado e, quando são introduzidos novos produtos, eles serão testados usando as técnicas de regressão. Novos casos de teste serão obtidos somente para explorar as características específicas dos novos produtos, para reutilizar os casos de teste que são comuns aos produtos da LPS. Assim, quando a inserção de um novo produto para a LPS é necessário, também é importante verificar a reutilização de casos de teste de produtos anteriores, e para criar um conjunto específico de casos de teste para testar novas funcionalidades do produto em teste.

A estratégia incremental é ilustrado na Figura 2.9, onde:

- Pr_i : i produto da LPS;
- T_i : caso de teste definido para o produto i da LPS gerado usando o método de Heumann (Heumann, J., 2010);

- Tr_j : conjunto de casos de teste reutilizados a partir de j produto da LPS no teste do produto i , para $j = 1, \dots, i - 1$;
- Tsi : conjunto de casos de teste específicos para o produto i da LPS.

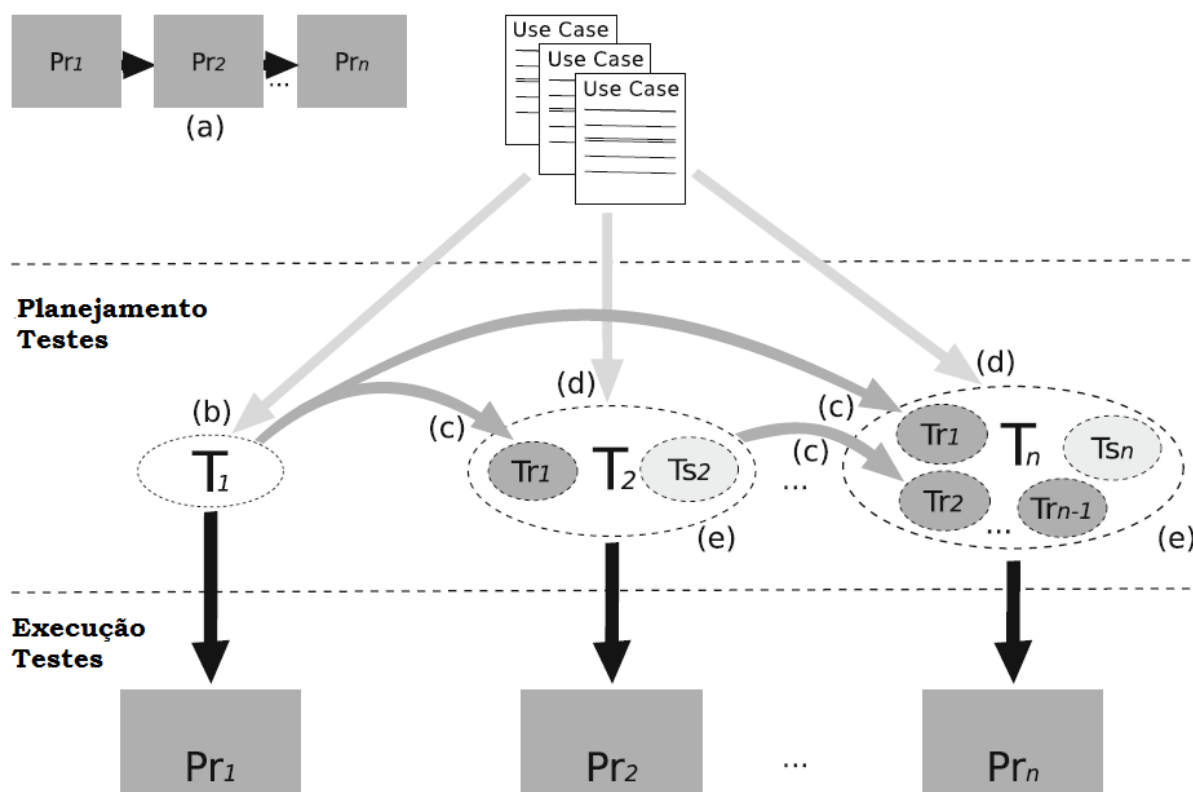


Figura 2.9 Estratégia Teste Incremental (Colanzi TE, Assunção, 2011)

Um conjunto de teste T_1 é especificamente gerado ao testar o primeiro produto. Quando um novo produto Pr_2 é introduzido na LPS, um conjunto de testes Tr_1 é selecionado, onde $Tr_1 \subset T_1$, ou seja, todos os Tr_j exploram as semelhanças entre o produto Pri e os produtos anteriores. Depois disso, um novo conjunto de testes Ts_2 que explora as características específicas de Pr_2 é criado. No final, os conjuntos de teste formam o caso de teste definido T_2 . Assim, o desenvolvimento de um novo produto Pri da LPS leva à geração de um novo conjunto de teste Tsi só para explorar recursos específicos de Pri , reutilizando o caso de teste Tr_j de outros produtos da LPS, reduzindo o esforço de teste.

2.3.3 Reusable Asset Instantiation – RAI

Na estratégia RAI, ativos de teste são criados na Engenharia de Domínio, considerando todas as variabilidades, quando artefatos e casos de teste abstratos são gerados. Na Engenharia de Aplicação esses ativos de teste abstratos refinado para testar as características específicas de cada produto. Assim, um aspecto importante da RAI é a reutilização de ativos de teste de *features* obrigatórias, uma vez que esses ativos são instanciados e reutilizados em todos os produtos da LPS (Figura 2.10).

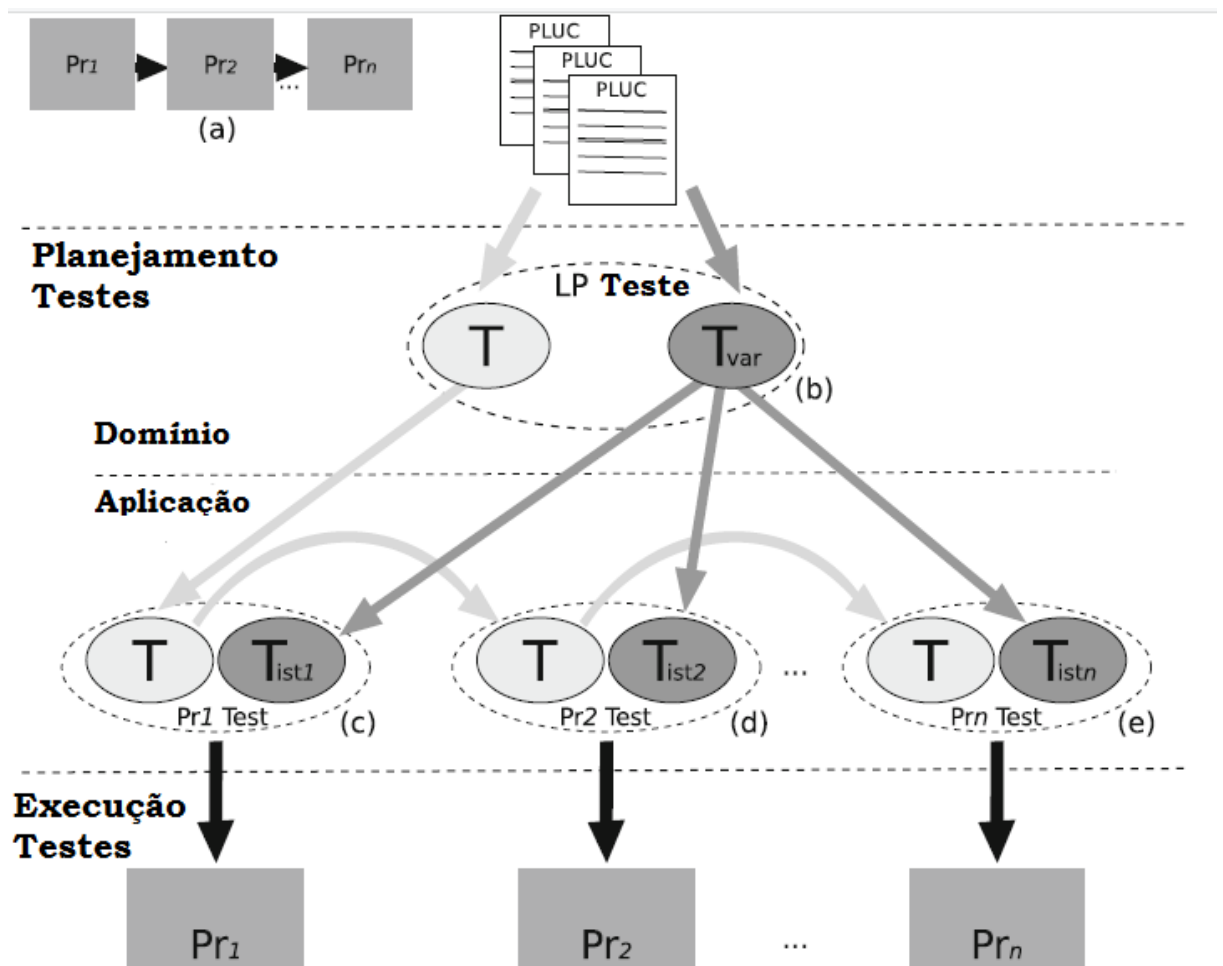


Figura 2.10 Estratégia RAI (Colanzi TE, Assunção, 2011)

2.3.3.1. Product Lines Use Case Test Optimization – PLUTO

PLUTO é aplicado na geração de dados de teste na estratégia RAI, porque RAI é orientada LPS e inclui teste no Nível de Engenharia de Domínio, e a metodologia de PLUTO é o que este nível considera.

Na Engenharia de Domínio, PLUTO gera casos de teste de cenários baseados em casos de uso, chamados PLUCs (Casos de Uso da Linha de Produto), que são refinados para testar os produtos. A notação utilizada para descrever os casos de uso é uma proposta por Cockburn (Cockburn A, 2001), mas estendido com anotações específicas para lidar com os aspectos variáveis da LPS.

Depois de analisar os PLUCs, um documento chamado Especificação de Teste é criada para cada PLUC, relatando as categorias, escolhas, restrições e cenários. Ele é a base para a geração de casos de teste. Para cada cenário, os casos de teste são criados pelas combinações possíveis de todas as escolhas de suas categorias. Quando um PLUC se refere a outro PLUC os casos de teste devem ser instanciados através da combinação de todas as categorias e escolhas dos dois PLUCs, considerando os cenários presentes em todos os casos de uso. A tarefa de instanciação dos casos de teste é iniciada apenas após a geração do documento de especificação de teste de todos os PLUCs.

Conforme definido pela RAI, no nível de domínio, PLUTO permite aos testadores descrever uma base genérica de recursos de teste referentes a todo o domínio da LPS, que corresponde aos requisitos obrigatórios da LPS, representado por T , e uma parte variável referente a ativos de teste para cada produto específico, representado por $Tvar$ (Bertolino A, Gnesi, 2004). Assim, o conjunto de teste para este nível, PL Teste, é dado por $Teste\ PL = T + Tvar$. No nível de aplicação, a metodologia permite aos testadores instanciar diretamente, para cada produto, a base dos ativos de teste genérico t , comum a todos os produtos, e também os seus bens de teste específicos, representado por $Tist$. Assim, o conjunto de teste a este nível, Pr Teste, é dada por $Pr\ Teste = T + Tist$. A técnica RAI está ilustrada na Figura 2.10.

Capítulo 3 - Framework conceitual de Apoio ao Teste de Regressão em Linhas de Produto de Software.

Este capítulo tem por objetivo apresentar a concepção do framework conceitual proposto neste trabalho para apoiar o teste de regressão em linhas de produto de software.

O processo de concepção e avaliação do *framework* proposto é apresentado ao longo deste capítulo.

3.1 Concepção do Framework Conceitual

A proposta desta pesquisa é a concepção de um *framework* conceitual que é constituído de conceitos e relacionamentos fundamentados em um conjunto de proposições adquiridas com o corpo de conhecimento. O cenário de aplicação do *framework* proposto são organizações em geral que desejam saber quais casos de teste aplicar em produtos gerados por uma configuração específica da LPS. Seu principal objetivo é listar os casos de teste que devem ser utilizados para garantir a qualidade da série de produtos gerados por aquela configuração inserida.

Na Figura 3.1 é apresentada uma visão geral do *framework* conceitual, onde primeiramente é realizada uma rastreabilidade entre os casos de teste e as *features* da LPS, ambos identificados como parâmetros inicial para utilização do framework. Isso visa identificar o relacionamento entre casos de teste e as *features*. O resultado dessa rastreabilidade é armazenado no *framework*.

Em um segundo momento, uma nova configuração de *features* para a LPS é fornecida ao *framework*, e então este realiza uma análise diagnosticando quais casos de teste devem ser utilizados para testar os produtos gerados pela nova configuração.

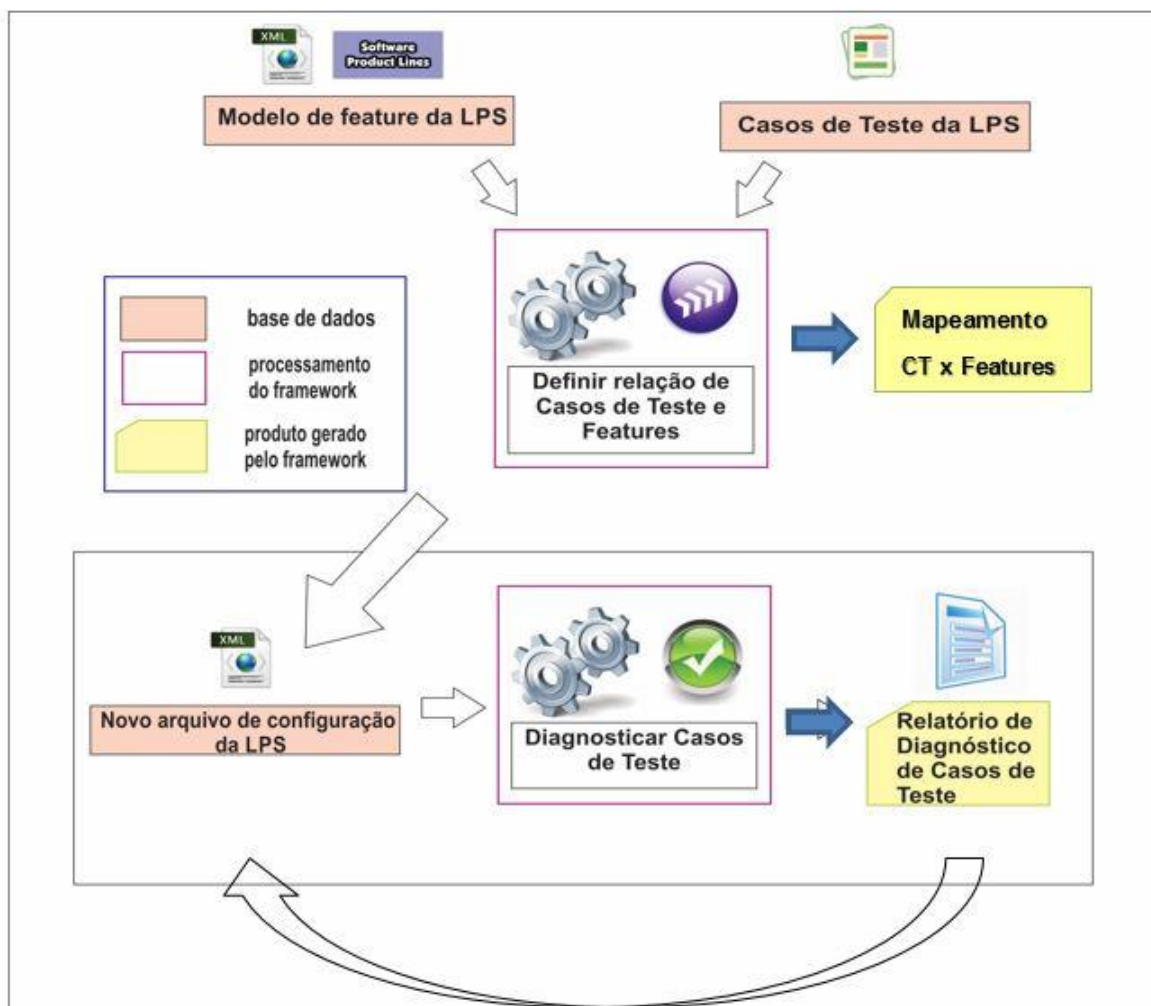


Figura 3. 1 Visão geral do framework conceitual

A Figura 3.2 apresenta o diagrama de atividades do *framework*, desde a inserção do modelo de *features* (Base de dados Modelo de *Features* da Figura 3.1) até a geração do produto final do *framework*, que é o Relatório de Diagnóstico de Casos de Teste para o teste dos produtos gerados pela configuração (Produto Relatório de Diagnóstico de Casos de Teste da Figura 3.1). Para representar esse *framework*, foi utilizado o diagrama de atividades da UML (*Unified Modeling Language*), proposto pela *Object Management Group* (OMG, 1997), que expressa a sequência de atividades de um software.

Observe na Figura 3.2 que a atividades 1, 2 e 4 são executadas pelo usuário, enquanto que as demais atividades (3 e 5) são executadas de forma automática pelo *framework* proposto neste trabalho. O Relatório de Diagnóstico de Casos de Teste (documento produzido na Etapa 5) é

composto pelos artefatos produzidos nas Etapas 3 e 4 do *framework*, conforme mostrado na Figura 3.2.

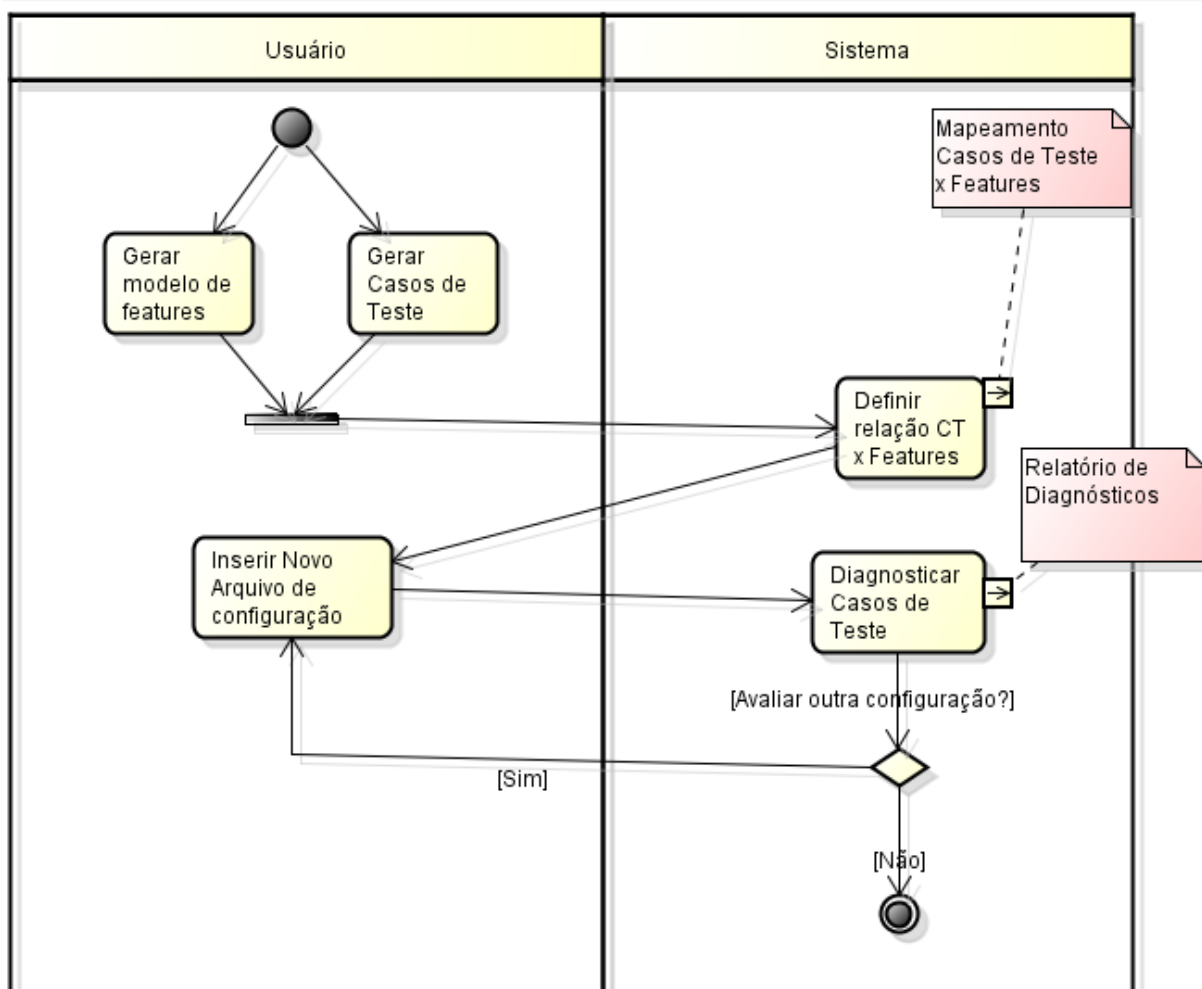


Figura 3.2 Diagrama de atividade do framework conceitual

3.2 Detalhamento das etapas do framework conceitual

Para facilitar a definição das atividades e dos artefatos produzidos, utilizou-se o *template* definido na Tabela 3.1. Dessa maneira, todas as etapas discutidas na Figura 3.1 estão apresentadas nas Tabelas 3.x, conforme os campos definidos no *template*.

Tabela 3.1: *Template* utilizado para definição dos componentes de processo

Identificador	<Nome do identificador>
Atividade:	<Nome da atividade>
Descrição:	<Descrição da atividade>
Entrada:	<Descrição dos dados de entrada>
Saída:	<Descrição dos dados de saída>
Responsável:	<Papel responsável pela execução da atividade>

3.2.1 Gerar Modelo de Features da LPS

Na etapa 1 deve ser gerado o modelo de features da LPS, pode ser utilizado o *feature model plugin* que trabalha integrado a ferramenta Eclipse. É necessário ter conhecimento de todas as *features* que compõem a LPS. Após a definição e montagem da árvore de produtos, é necessário exportar o modelo para o formato XML.

Como exemplo usamos a LPS de um EShop, a figura 3.3 demonstra sua árvore de *features* utilizando o plugin *fmp* e a ferramenta Eclipse. Na figura 3.4 vimos o arquivo XML originário da exportação da LPS, no arquivo pode ser observado o uso de algumas tags.

- Feature: identifica a *feature*, com seu número máximo e mínimo de seleções, o nome atribuído a ela, seu tipo e o id único.
- Feature Group: refere-se a um conjunto de *features*, contendo o número máximo e mínimo de seleções dentro do grupo e seu Id.

A definição de valores máximos e mínimos auxilia na identificação se a *feature* é obrigatória (min = max), opcional (min = 0) ou alternativa (quando min = max, dentro de um grupo).

Tabela 3.2: Atividade da Etapa 1

Identificador	Etapa 1
Atividade:	Gerar modelo de features da LPS
Descrição:	Nessa etapa deve ser gerado o modelo de features da LPS usando o plugin FMP (Antkiewicz e Czarnecki, 2004) integrado ao Eclipse, contendo as features e as dependências entre elas, gerando um arquivo XML a ser inserido no <i>framework</i> .
Entrada:	Descrição da LPS
Saída:	Modelo de <i>features</i> da LPS em formato XML
Responsável:	Usuário

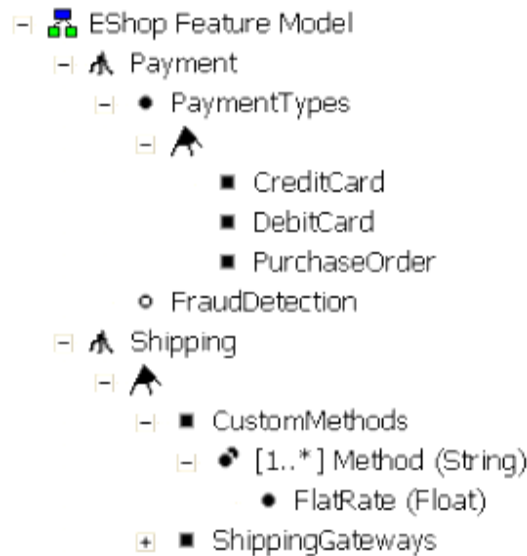


Figura 3.3 Exemplo de um *feature model* no editor eclipse usando fmp

```

<feature min="1" max="1" name="My Feature Model" type="NONE" id="My Feature Model">
  <feature min="1" max="1" name="Eshop" type="NONE" id="catalogo">
    <feature min="1" max="1" name="Catalogo" type="NONE" id="catalogo0">
    </feature>
    <feature min="1" max="1" name="Pagamento" type="NONE" id="pagamentp">
      <featureGroup min="1" max="1" id="group">
        <feature min="0" max="1" name="Cartao credito" type="NONE" id="cartaocredito">
        </feature>
        <feature min="0" max="1" name="Transferencia Bancaria" type="NONE" id="transferenc">
        </feature>
      </featureGroup>
    </feature>
    <feature min="1" max="1" name="Seguranca" type="NONE" id="seguranca">
      <featureGroup min="1" max="1" id="group0">

```

Figura 3.4 Trecho de arquivo XML gerado pelo fmp

3.2.2 Gerar Casos de Teste da LPS

Nessa etapa o usuário é responsável por gerar os casos de teste para o teste dos produtos da LPS e submetê-los ao *framework* através da tela de cadastro de casos de teste (ver Fig. 3.5). Existem diversas técnicas para geração de testes em LPS, algumas dessas técnicas foram apresentadas no referencial teórico, mas podem existir outras, o *framework* não faz distinção de qual técnica foi utilizada.

Alguns campos a serem preenchidos são:

- **ID:** Número de identificação do caso de teste, pelo qual será referenciado no sistema;
- **Nome:** Nome do caso de teste;

- **Autor:** Nome do autor do caso de teste;
- **Entradas:** Entradas que devem ser inseridas para executar o referido caso de teste;
- **Resultados Esperados:** Resultados esperados que o sistema retorne para o conjunto de entradas especificado;
- **Descrição:** Texto descrevendo o objetivo do caso de teste;
- **Dependência com CTs:** Devem ser marcados os casos de teste que afetam o referido caso de teste.

Figura 3.5 Exemplo de tela de Cadastro de Casos de Teste

Tabela 3.3: Atividade da Etapa 2

Identificador	Etapa2
Atividade:	Gerar casos de teste da LPS
Descrição:	Nessa etapa devem ser gerados os casos de teste da LPS usando o padrão IEEE-829 contendo as entradas e as dependências entre elas, a ser inserido no <i>framework</i> .
Entrada:	Descrição da LPS
Saída:	Casos de teste da LPS.
Responsável:	Usuário

3.2.3 Definir Relação de Casos de Teste com Features

Nessa etapa o usuário é responsável por relacionar os casos de teste, cadastrados na Etapa2, e as features inseridas na Etapa1, e submetê-los ao *framework* através da tela de mapeamento (ver Fig. 3.6). Os casos de teste são identificados pelo seu ID e as features apresentam seu nome. O objetivo dessa etapa é identificar quais casos de teste testam determinadas features, depois desse processor é gerado um documento mapeando as features e casos de testes disponível no Apêndice A.

Tabela 3.4: Atividades da Etapa 3

Identificador	Etapa3
Atividade:	Definir relação de Casos de Teste com features
Descrição:	Nessa etapa deve ser gerado um mapeamento contendo as features que exercitam determinados casos de teste.
Entrada:	Casos de Teste e modelo de features.
Saída:	Mapeamento de casos de teste e features [Apêndice A].
Responsável:	Sistema

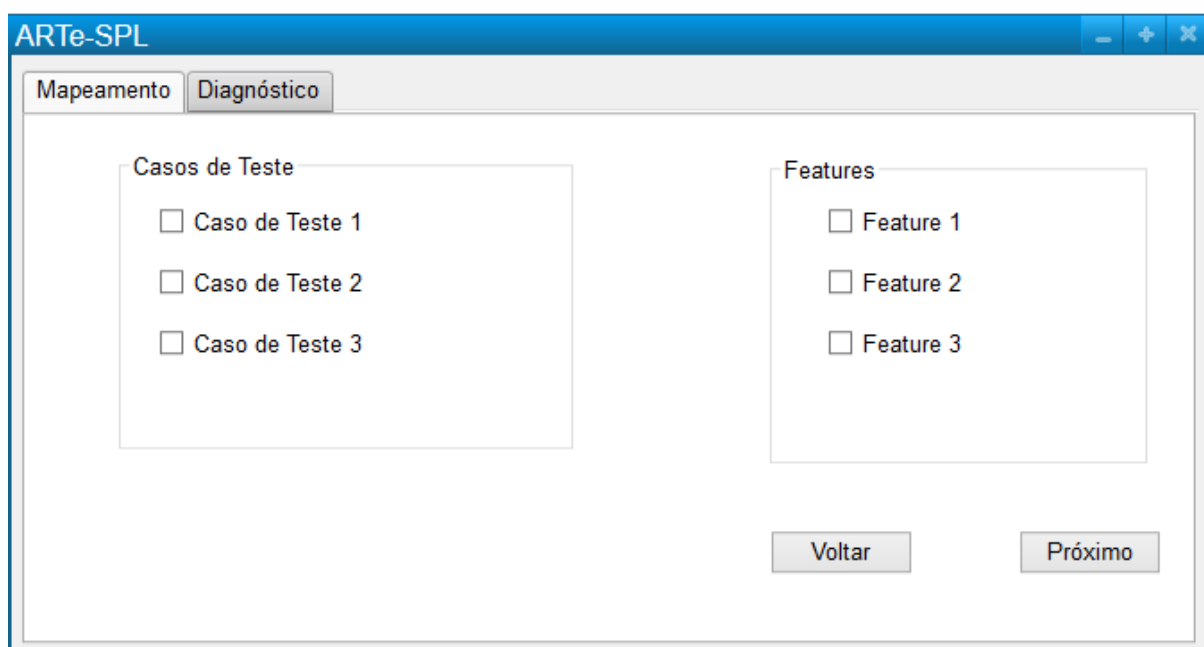


Figura 3.6 Exemplo de tela de mapeamento

3.2.4 Definir Nova Configuração da LPS

Na Etapa 4 deve ser gerada a nova configuração da LPS, pode ser utilizado o *feature model plugin* que trabalha integrado a ferramenta Eclipse

e é necessário ter executado a Etapa 1. Deve ser selecionada a configuração desejada e exportar o modelo para o formato XML.

Como exemplo usamos a LPS de um EShop, a figura 3.7 demonstra sua árvore de *features* utilizando o plugin *fmp* e sua devida configuração selecionada. Na figura 3.8 vimos o arquivo XML originário da exportação da LPS.

Tabela 3.5: Atividades da Etapa 4

Identificador	Etapa4
Atividade:	Definir configuração da LPS
Descrição:	Nessa etapa deve ser gerado uma configuração para a LPS.
Entrada:	Modelo de <i>features</i> .
Saída:	Arquivo de configuração da LPS em formato XML.
Responsável:	Usuário

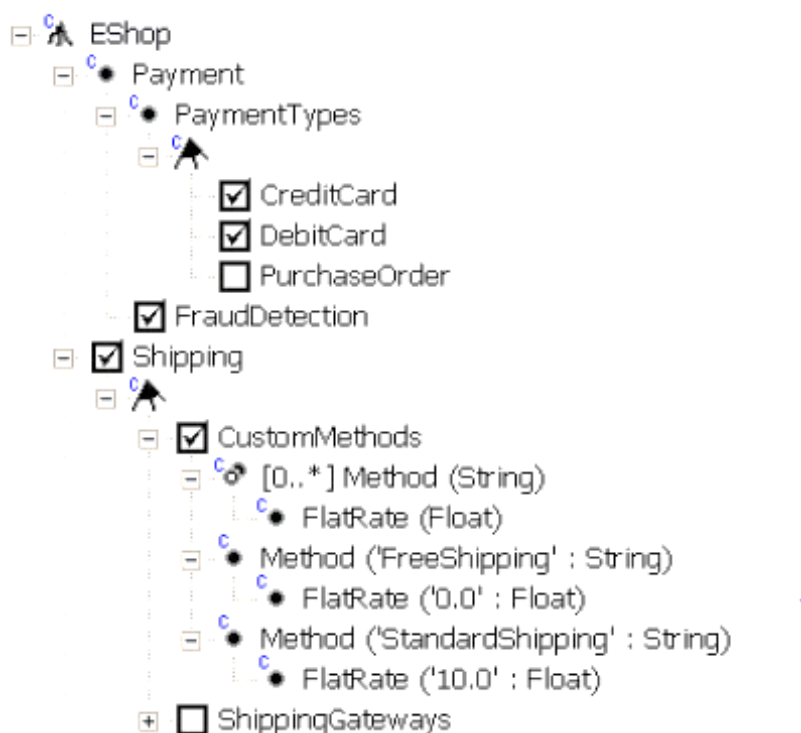


Figura 3.7 Exemplo de configuração da LPS no editor eclipse usando *fmp*

```

<feature name="Eshop" type="NONE" id="eshop">
  <feature name="Payment" type="NONE" id="payment">
    <feature name="PaymentTypes" type="NONE" id="paymentTypes">
    </feature>
    <feature name="FraudDetection" type="NONE" id="fraude">
    </feature>
  </feature>
  <feature name="Shipping" type="NONE" id="shipping">
    <feature name="ShippingGateways" type="NONE" id="shippingGateways">
    </feature>
  </feature>
</feature>

```

Figura 3.8 Trecho de arquivo XML de configuração

3.2.5 Diagnosticar Casos de Teste

Nessa etapa o sistema é responsável por identificar quais casos de teste serão utilizados para testar a nova configuração da LPS, utilizando para isso o mapeamento CT x *features* e a nova configuração inserida. Os casos de teste são identificados pelo seu ID enome, e o arquivo de configuração da LPS é mostrado no relatório. O objetivo dessa etapa é identificar quais casos de teste testam a configuração, depois desse processor é gerado um documento de diagnóstico de casos de teste disponível no Apêndice B.

Tabela 3.6: Atividades da Etapa 5

Identificador	Etapa5
Atividade:	Diagnosticar Casos de Teste
Descrição:	Nessa etapa deve ser gerado um relatório de diagnóstico de casos de teste.
Entrada:	Mapeamento CT x <i>features</i> e arquivo de configuração da LPS em formato XML.
Saída:	Relatório de diagnóstico de Casos de Teste. [Apêndice B]
Responsável:	Sistema

4 Conclusões e Perspectivas Futuras

Neste capítulo serão apresentadas as conclusões a respeito do trabalho desenvolvido e perspectivas futuras para continuidade da pesquisa.

4.1 Considerações finais

Uma das formas mais adotadas para testar os produtos é o teste individual de cada produto, porém ao definir uma configuração (diminuindo assim o número de produtos gerados) não se torna mais necessário executar todos os casos de teste antes propostos.

Diante disso, este trabalho apresenta um *framework* conceitual que tem como objetivo selecionar os casos de teste a serem executados para garantir a qualidade dos produtos gerados por uma configuração específica.

Considerando que o objetivo deste trabalho era apresentar o *framework* que a partir de um mapa de relacionamento de casos de teste e *features* e uma configuração determinada pelo usuário, apresentasse um relatório de casos de teste a serem utilizados para testar aquela configuração, pode-se afirmar que este foi cumprido, uma vez que todos os objetivos específicos foram atingidos.

4.2 Trabalhos Futuros

Com o propósito de estender e aprimorar os resultados obtidos, algumas das perspectivas de trabalhos futuros são apresentadas a seguir:

- **Extensão da Pesquisa:** Alguns aspectos relacionados à investigação conduzidas neste trabalho podem ser mais explorados em novas investigações:
 - Verificar, em estudos e trabalhos relacionados, como testes de regressão em LPS estão aplicados em pesquisas científicas e em organizações de software;
- **Implementação e Avaliação do Framework Conceitual:** Algumas pesquisas podem ser realizadas para implementar e avaliar o *framework* conceitual:

- Desenvolver uma ferramental computacional para dar apoio semi-automatizado ao modelo proposto de forma a ficar acessível a qualquer organização de software que queira utilizá-lo;
- Realizar avaliações, com organizações de software, do *framework* conceitual com o propósito de verificar sua aplicabilidade em relação ao tempo e custo de implantação.

Referências

- Ammann, P.; Offutt, J. "Introduction to Software Testing". Cambridge University Press, 2008, 344p.
- Antkiewicz, M.; Czarnecki, K. "FeaturePlugin: Feature Modeling Plug-In for Eclipse". University of Waterloo, Canada.
- Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Transaction on Dependable Secure Computing, vol. 1, Jan-Mar 2004, pp. 11-33.
- Bertolino A, Gnesi S (2004) Pluto: a test methodology for product families. Lect Notes Comput Sci 3014:181-197
- Bertolino, A. "Knowledge Area Description of Software Testing SWEBOK". Available in: <http://www.computer.org/portal/web/swebok>, April 2013.
- Burnstein, I. "Practical Software Testing: A Process-Oriented Approach". Springer, 2003, 709p.
- Clements, P.C., Northrop, L. Software Product Line: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, August 2001.
- Cockburn, A (2001) Writing effective use cases. Addison Wesley, Reading.
- Colanzi TE, Assunção WKG, Trindade DFG, Zorzo CA, Vergilio SR (2011) Evaluating Different Strategies for Testing Software. J Electron Test 29:9-24.
- Committee, I. C. S. S. E. T. "IEEE 610.12 Standard Glossary of Software Engineering Terminology". Institute of Electrical and Electronics Engineers, 1-84p, 1990.
- Costa, L. T.; Oliveira, F. M.; Rodrigues, E. M.; Silveira, Maicon Bernardino; Zorzo, A. F. "Uma Abordagem para Geração de Casos de Teste Estrutural Baseada em Modelos". In: Proceedings of Workshop de Teste e Tolerância a Falhas, 2012, pp. 87-100.

- Delamaro, M. E.; Maldonado, J. C.; Jino, M. "Introdução ao Teste de Software". Campus, 2007, 408p.
- Denger, C; Kolb, R. Testing and inspecting reusable product line componentes: first empirical results. In Proceedings of the 2006 ACM/IEEE. International symposium on Empirical software engineering (ISESE'06). ACM, New York, NY, USA, 184-193.
- Engström, E.; Runeson, P. "Software Product Line Testing - A Systematic Mapping Study", Information and Software Technology, vol. 53, Jan 2011, pp. 2-13.
- Everett, G.; Raymond, J. "Software Testing: Testing Across the Entire Software Development Life Cycle". John Wiley & Sons, 2007, 280p.
- Heumann J (2001) Generating test cases from use cases. The rational edge. Technical report. Available at: <http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jun01/GeneratingTestCasesFromUseCasesJune01.pdf>. Accessed June 2010
- Hewlett Packard - HP. "Software HP LoadRunner". Available in: <https://h10078.www1.hp.com/cda/hpms/>, April 2013.
- Heymans, P.; Trigaux, J. C.: "Software product line: state of the art". Technical report for PLENTY project, Institut d.'Informatique FUNDP, Namur, 2003.
- IEEE. "IEEE Std 829 Standard for Software and System Test Documentation". Available in: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4578383>, July 2013.
- Jaaksi A. "Developing mobile browsers in a product line". In: IEEE Softw 19:73-80, 2002
- Jeffrey, D.; Gupta, N. "Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction", IEEE Transactions on Software Engineering, vol. 33, Feb 2007, pp. 108-123.

- Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, Carnegie-Mellon University Software Engineering Institute, 1990, 148p.
- Lee, J.; Kang, S.; Lee, D. "A survey on Software Product Line Testing". In: Proceedings of the 16th International Software Product Line Conference - Volume 1, 2012, pp. 31–40.
- Li, Z.; Harman, M.; Hierons, R. M. "Search Algorithms for Regression Test Case Prioritization", IEEE Transactions on Software Engineering, vol. 33, Apr 2007, pp. 225–237.
- Linden, F. J.; Schmid, K.; Rommes, E. "Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering". Springer-Verlag New York, Inc., 2007, 333p.
- Myers, G. J.; Sandler, C. "The Art of Software Testing". John Wiley & Sons, 2004, 256p.
- Naik, S.; Tripathy, P. "Software Testing and Quality Assurance: Theory and Practice". Wiley, 2011, 648p.
- Olimpiew, E. M.; Gomaa, H. "Model-based Testing for Applications Derived from Software Product Lines". In: Proceedings of the 1st international workshop on Advances in model-based testing, 2005, pp. 1–7.
- Pohl, K.; Böckle, G.; Linden, F. J. v. d. "Software Product Line Engineering: Foundations, Principles and Techniques". Springer-Verlag New York, Inc., 2005, 494p.
- Sommerville, I. "Software Engineering". Pearson/Addison-Wesley, 2011, 792p.
- Utting, M.; Legeard, B. "Practical Model-Based Testing: A Tools Approach". Morgan Kaufmann, 2006, 456p.
- Weiss, D.M; Lai, C. T. R. Software Product-Line Engineering; A Family-Based Software Development Process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

APÊNDICE A – Mapeamento CT x Features



Universidade Federal do Amazonas (UFAM)
Pró-reitoria de Ensino de Graduação (PROEG)
Instituto de Computação (IComp)



Mapeamento Casos de Teste x *features*

Casos de Teste	<i>Features</i>
CT01, CT02	FT03
CT04	FT05, FT07
CT03	FT01

APÊNDICE B – Relatório de Diagnóstico de Casos de Teste



Universidade Federal do Amazonas (UFAM)
Pró-reitoria de Ensino de Graduação (PROEG)
Instituto de Computação (IComp)



Relatório de Diagnóstico de Casos de Teste

- Configuração Inserida

```
<feature name="Eshop" type="NONE" id="eshop">
  <feature name="Payment" type="NONE" id="payment">
    <feature name="PaymentTypes" type="NONE" id="paymentTypes">
    </feature>
    <feature name="FraudDetection" type="NONE" id="fraude">
    </feature>
  </feature>
  <feature name="Shipping" type="NONE" id="shipping">
    <feature name="ShippingGateways" type="NONE" id="shippingGateways">
    </feature>
  </feature>
</feature>
```

- Casos de Teste a serem utilizados

Identificador	Título
CT01	Pagamento inválido.
CT03	Pagamento válido.
CT05	Deteção fraude.
CT02	Quantidade válida.
CT07	Quantidade inválida.