Glauco Silva Neves

UMA ABORDAGEM REATIVA DE CONSTRUÇÃO DE LINHAS DE PRODUTO DE SOFTWARE BASEADA EM TDD E REFATORAÇÃO

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciência da Computação.
Orientadora: Prof.ª Dra. Patrícia Vilain

Florianópolis
2014

Ficha de identificação da obra elaborada pelo autor
através do Programa de Geração Automática da Biblioteca Universitária
da UFSC.

A ficha de identificação é elaborada pelo próprio autor
Maiores informações em:
http://portalbu.ufsc.br/ficha

Glauco Silva Neves

UMA ABORDAGEM REATIVA DE CONSTRUÇÃO DE LINHAS DE PRODUTO DE SOFTWARE BASEADA EM TDD E REFATORAÇÃO

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre", e aprovada em sua forma final pelo Programa Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Florianópolis, 25 de setembro de 2014.

_____
Prof. Ronaldo dos Santos Mello, Dr.
Coordenador do Curso

Banca Examinadora:

_____
Prof.ª Patrícia Vilain, Dr.ª
Orientadora
Universidade Federal de Santa Catarina

_____
Prof. Edson Alves de Oliveira Junior, Dr.
Universidade Estadual de Maringá

_____
Prof. Raul Sidnei Wazlawick, Dr.
Universidade Federal de Santa Catarina

_____
Prof. Mario Antonio Ribeiro Dantas, Dr.
Universidade Federal de Santa Catarina

Finish what you started.

AGRADECIMENTOS

Aos meus pais, agora e sempre, por fornecerem todos os alicerces necessários para que eu chegasse até aqui.

À Ceci, minha companheira, que me acompanhou por toda essa experiência que foi o mestrado, e que sempre estará ao meu lado.

Agradeço também à professora Patrícia Vilain pelo envolvimento e orientação no trabalho.

No mais, agradeço todos os envolvidos para que essa dissertação fosse concluída: à UFSC, pela oportunidade e ambiente propício para o desenvolvimento da pesquisa acadêmica; aos meus amigos, que deram suporte e sempre acreditaram no meu potencial; e aos professores da banca, pela contribuições construtivas.

Most industry observers agree that improved software development productivity and product quality will bring an end to the software crisis. In such a world, reusable software would abound.

(Pressman, 1982)

RESUMO

Linhas de Produto de Software (LPS) trazem diversos benefícios, como a diminuição do tempo de entrada no mercado, a redução dos custos de desenvolvimento, o aumento da produtividade e a melhora da qualidade do produto final. Uma das práticas que auxilia na garantia de qualidade é a prática de testes. No entanto, ainda existem desafios e lacunas na utilização desta prática no desenvolvimento de LPSs. Nem todas as técnicas de testes utilizadas no desenvolvimento de um produto único podem ser aplicadas em LPSs e, portanto, novas adaptações e propostas são necessárias. Além disso, o desenvolvimento tradicional de LPSs também demanda um alto investimento inicial de longo prazo e oferece riscos para mercados dinâmicos, onde mudanças são difíceis de prever. Entretanto, já existem propostas para levar as vantagens de LPSs para mercados dinâmicos por meio da utilização de práticas de desenvolvimento ágil de software, essa união é denominada Engenharia de Linha de Produto Ágil (ELPA). Esta dissertação visa a elaboração de uma abordagem para a construção de LPSs, utilizando a ELPA. Nesta proposta as práticas ágeis de Desenvolvimento Dirigido por Testes (*Test-Driven Development - TDD*) e Refatoração conduzem a criação de uma LPS de abordagem reativa sem a tentativa de prever variações futuras. Para dar suporte à prática de testes unitários no desenvolvimento reativo de LPSs, foi desenvolvido um *framework* de testes com a proposta de adaptar padrões de testes unitários que facilitem a verificação da exatidão das aplicações geradas. Os padrões de teste Framework de Automação de Testes e Testes Dirigidos por Dados fornecem a reutilização da lógica de testes e a automação dos mecanismos de implementação, reduzindo o esforço necessário para testar as variações de cada aplicação. A proposta foi avaliada através de um exemplo que mostrou a aplicação da abordagem e do framework de testes em uma LPS que foi construída de forma reativa a partir de uma aplicação existente. Como resultado foi visto um alto grau de reuso de testes, com 89% de reuso na segunda aplicação, após a modificação de três *features*, 97% na terceira aplicação, após a adição de uma *feature* e modificação de outra, e 100% na quarta aplicação onde nenhuma *feature* foi adicionada ou modificada, e a aplicação foi construída com variantes existentes.

Palavras-chave: Linha de produto de software. Desenvolvimento ágil de software. Desenvolvimento dirigido por testes. Refatoração. Testes. Padrões de testes unitários.

ABSTRACT

Software Product Line (SPL) brings benefits such as lower time-to-market, less development costs, increased productivity, and improved quality. The quality assurance can be reached through the testing area, however this area still has challenges and gaps in the SPL development. Since not all testing techniques used in a single product development can be applied to SPL, thus some adaptations and new proposals are required. Traditional SPL also requires a high initial investment and offers long-term risks to dynamic markets where changes are difficult to predict. Currently, proposals bring the advantages of SPL for dynamic markets through the use of agile software development practices, which is called Agile Product Line Engineering (APLE). This work presents the development of the necessary steps for building SPL using the APLE. In this proposal the agile practices Test-Driven Development (TDD) and Refactoring lead a reactive development of SPL without attempting to predict future variations. It is also proposed to adapt unit test patterns in the context of SPL. The test patterns Test Automation Framework and Data-Driven Tests provide the reuse of test logic and automation of the implementation mechanisms, reducing the effort required to test variations of each application. The result of this adaptation is a testing framework to be used during application engineering to configure tests through parameterized tests and verify the correctness of generated applications. Thus, this work shows how the agile practices TDD and Refactoring may cause a SPL to evolve and acquire variation points on demand. The proposal was evaluated through an example of a SPL that was built with a reactive approach from an existing application. As a result, it was obtained a high degree of tests reuse, with 89% of reuse in the second application after modifying three features, 97% in the third application after adding one feature and modifying another one, and 100% of reuse in the fourth application where no feature was added or modified, and the application was built with existing variants.

Keywords: Software product line, Test-driven development, Refactoring, Variability realization. Software product line testing. Software testing. Unit test patterns.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
BDD	*Behavior Driven Development*
COTS	*Commercial Off-The-Shelf*
DDT	*Data-Driven Test*
DDTAF	*Data-Driven Test Automation Framework*
ELPA	Engenharia de Linha de Produto Ágil
JSON	*JavaScript Object Notation*
LPS	Linha de Produto de Software
SPL	*Software Product Line*
SUT	*System Under Test*
TAF	*Test Automation Framework*
TDD	*Test-Driven Development*
XP	*Extreme Programming*

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo são apresentados a contextualização do problema, a motivação, os objetivos alcançados, as contribuições, as limitações e a estrutura da dissertação.

1.1 CONTEXTUALIZAÇÃO DO PROBLEMA

Linha de Produto de Software (LPS) é um conjunto de aplicações que compartilham pontos em comum visando um determinado segmento de mercado (CLEMENTS; NORTHROP, 2001). Essa prática já comprovou as vantagens que possui em reduzir o tempo de entrada no mercado, diminuir os custos de desenvolvimento, aumentar a produtividade e melhorar a qualidade das aplicações (CLEMENTS; NORTHROP, 2001). Existem duas fases essenciais para o desenvolvimento de uma LPS, a engenharia de domínio e a engenharia de aplicação (POHL; BÖCKLE; LINDEN, 2005).

Na engenharia de domínio é feito um planejamento inicial visando identificar os pontos em comum dos produtos e os pontos onde os produtos podem variar entre si. Nessa fase é definida a arquitetura da LPS, e os artefatos de software são produzidos para serem utilizados pelos produtos. Em todos os artefatos de software podem ser introduzidas variabilidades (POHL; BÖCKLE; LINDEN, 2005). Esses artefatos irão constituir o repositório de *core assets*[1].

Na engenharia de aplicação, o foco é entender as necessidades de aplicações específicas dos clientes para poder reusar os recursos do repositório de *core assets*, como, por exemplo, a arquitetura, código e testes, explorando os pontos de variação e montando a aplicação.

No entanto, é necessário um alto investimento antecipado e de longo prazo para projetar e desenvolver inicialmente o repositório de *core assets*. Isto dificulta a utilização da LPS em mercados dinâmicos, onde existe o risco de mudanças não previstas e de artefatos desenvolvidos não serem utilizados. Para suprir essa dificuldade, existem propostas de combinar o desenvolvimento ágil de software com LPS, resultando na Engenharia de Linha de Produto Ágil (ELPA) (DÍAZ et al., 2011).

Uma das práticas ágeis empregadas na ELPA é a de Desenvolvimento Dirigido por Testes (*Test-Driven Development* – TDD), que encoraja os desenvolvedores a escreverem os testes antes de

[1] Recursos reusáveis para construção de software

escreverem o código que vai ser testado. Aplicar TDD em LPS requer o entendimento do funcionamento da área de testes em LPS. Essa área é fundamental na prática de engenharia de software, porém ainda necessita de trabalhos de pesquisa no contexto de LPS (NETO et al., 2011). De acordo com Engström e Runeson (2011) o foco em testes de LPSs não foi significativo em pesquisas anteriores, embora os testes sejam essenciais para descobrir defeitos, como demonstrado no desenvolvimento de produtos únicos (NETO et al., 2011).

Boas práticas de testes podem reduzir o tempo de desenvolvimento, melhorar a qualidade do produto, aumentar a satisfação dos clientes e reduzir os custos de manutenção (NETO et al., 2011). No entanto, apesar de existirem várias propostas para testes em LPS, ainda existem muitos desafios, principalmente relacionados com os testes unitários, testes de regressão e automação de testes (NETO et al., 2011) (GANESAN et al., 2013) (LAMANCHA; USAOLA; VELTHIUS, 2009) (LEE; KANG; LEE, 2012).

Lamancha, Usaola e Velthius (2009) também destacam que existem apenas recomendações para a aplicação de testes unitários em LPS, porém técnicas específicas de testes ainda são escassas. Automação de testes e re-execução é outra área crítica dos testes em LPS (LAMANCHA; USAOLA; VELTHIUS, 2009). Muitas técnicas de testes não têm sido amplamente exploradas, e entre elas estão a seleção de testes durante os testes de aplicação e a reutilização de código de teste (LEE; KANG; LEE, 2012). Estas questões são abordadas nesta dissertação.

1.2 MOTIVAÇÃO

A motivação dessa dissertação é a evolução de aplicações que foram desenvolvidas sem visar o reuso para LPSs, reutilizando código e testes. Isto pode ser alcançado por meio de uma abordagem de ELPA para definir os passos para a construção reativa de LPSs, com foco no reuso dos testes unitários. Assim, a variabilidade da LPS pode ser especificada sob demanda usando as práticas de TDD e de Refatoração.

Por ser uma prática central do TDD, é importante que o foco seja nos testes unitários. Assim, para facilitar a reutilização, execução e automação de testes unitários é possível a utilização de padrões de testes unitários no contexto de LPS. Essas vantagens podem ser obtidas com três padrões catalogados por Meszaros (2007): (1) *Framework* de Automação de Testes (*Test Automation Framework* – TAF), responsável por facilitar a escrita e execução de testes; (2) Testes Dirigidos por

Dados (*Data-Driven Test* – DDT), que ajudam a preparar os testes automatizados; e (3) Testes Parametrizáveis, que reduzem a duplicação do código de teste.

1.3 OBJETIVOS

Com base na contextualização do problema e na motivação, são definidos o objetivo geral e os objetivos específicos desta dissertação.

1.3.1 Objetivo Geral

Definir uma abordagem para construir LPSs de abordagem reativa onde a variabilidade das LPSs é especificada sob demanda usando as práticas de desenvolvimento dirigido por testes e de refatoração, com suporte para o reuso e execução dos testes.

1.3.2 Objetivos Específicos

Três objetivos específicos foram alcançados e representam contribuições no escopo desta dissertação. Estes objetivos são os seguintes:

1. Definir uma abordagem e seus passos para a criação de LPSs de abordagem reativa que utilizam as práticas de TDD e refatoração.
2. Construir uma *framework* de testes baseado em padrões de testes unitários, que facilitam a escrita dos testes e o reuso deles, e que pode ser aplicado para facilitar a seleção e execução dos testes dos novos pontos de variação na LPS.
3. Validar a aplicação da abordagem e do *framework* de testes, por meio de um exemplo, construindo uma LPS através de uma abordagem reativa.

1.4 CONTRIBUIÇÕES

As principais contribuições desta dissertação vem através de: (1) uma abordagem que faz uso das práticas ágeis de TDD e a refatoração de código para a evolução de uma LPS de abordagem reativa, incorporando pontos de variação e variantes conforme as necessidades vão aparecendo, e (2) um *framework* de testes unitários que dá suporte à abordagem proposta, facilitando a reutilização, execução e automação de testes unitários.

As vantagens do *framework* de testes são alcançadas por meio do uso de padrões de testes unitários. Esse *framework* de testes também pode ser usado em outras abordagens de LPS, por exemplo, quando se utiliza uma estratégia proativa (NEVES; VILAIN, 2014b).

1.5 METODOLOGIA

Do ponto de vista da natureza do trabalho, essa é uma pesquisa aplicada, pois tem com objetivo "gerar conhecimentos para aplicação prática e dirigidos à solução de problemas específicos" (SILVA; MENEZES, 2005).

Do ponto de vista dos objetivos, é uma pesquisa exploratória, pois visa "proporcionar visão geral, de tipo aproximativo, acerca de determinado fato" (GIL, 1991).

Do ponto de vista dos procedimentos técnicos, essa pesquisa é do tipo pesquisa bibliográfica, pois será realizada "a partir de material já elaborado, constituído principalmente de livros e artigos científicos" (GIL, 1991).

1.6 ESCOPO

O escopo desta dissertação é para organizações que atuam em mercados dinâmicos, onde existe o risco de mudanças não previstas e de artefatos desenvolvidos não serem utilizados.

No que se refere às LPSs, essa dissertação foca em LPSs do tipo reativa, que não fazem um grande investimento inicial para o desenvolvimento do repositório de artefatos e dessa forma nem todas as práticas aqui sugeridas servem para as LPSs do tipo proativa.

1.7 ESTRUTURA DO TRABALHO

A presente dissertação está dividida em sete capítulos organizados da seguinte maneira: no capítulo dois é feita a fundamentação teórica da dissertação onde são apresentados os principais conceitos de LPS, da combinação de LPS com práticas ágeis, e da área de testes. No capítulo três é apresentada a revisão do estado da arte executada por meio de uma revisão sistemática da literatura com o objetivo de identificar como é usada a prática de Desenvolvimento Dirigido por Testes e como são feitos os testes unitários em LPS. No capítulo quatro é descrita a proposta de uma abordagem para a construção de uma LPS de abordagem reativa e o *framework* de testes

para dar suporte à abordagem apresentada. O capítulo cinco apresenta um exemplo descrevendo uma LPS construída com base na abordagem proposta. O capítulo seis mostra uma comparação desta proposta com os trabalhos relacionados. O capítulo sete conclui a dissertação, mostrando o que foi feito, quais foram as contribuições e sugestões de trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Nesse capítulo são apresentados conceitos que são relevantes no contexto da presente dissertação: linhas de produto de software, engenharia de linha de produto ágil, testes unitários e aplicação de testes em LPS.

2.1 LINHA DE PRODUTO DE SOFTWARE

Segundo Clements e Northrop (2001), uma Linha de Produto de Software (LPS) é:

> Uma família de produtos de software que compartilham um conjunto comum e gerenciado de funcionalidades que satisfazem as necessidades específicas de um segmento de mercado particular ou missão e que são desenvolvidos a partir de um repositório central de artefatos comuns de forma prescrita (CLEMENTS; NORTHROP, 2001).

A produção de novos produtos se torna mais econômica pela reutilização de componentes e pela utilização de mecanismos de variabilidade pré-planejados (CLEMENTS; NORTHROP, 2001). Um conceito importante em LPSs é o de variabilidade. A variabilidade indica a habilidade que a LPS tem em variar. Essa variabilidade é explorada na criação dos produtos da LPS de forma premeditada (POHL; BÖCKLE; LINDEN, 2005).

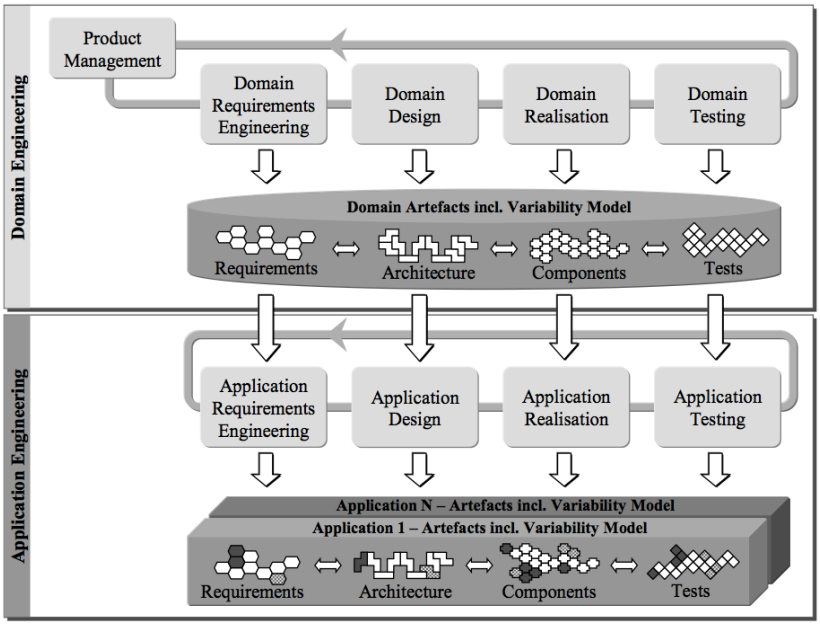
A utilização de LPS traz diversas vantagens que já foram observadas em várias empresas (CLEMENTS; NORTHROP, 2001), tais como: ganhos de produtividade em grande escala; diminuição do tempo de entrada no mercado; aumento da qualidade do produto; diminuição do risco do produto; maior agilidade no mercado; aumento da satisfação do cliente; utilização mais eficiente dos recursos humanos; capacidade de efetuar customização em massa; capacidade de manter presença no mercado; capacidade de sustentar o crescimento sem precedentes.

2.1.1 Atividades da Linha de Produto de Software

Desenvolver uma LPS envolve três atividades principais: (1) a engenharia de domínio e (2) a engenharia de aplicação, que são desenvolvidas sob (3) o gerenciamento técnico e organizacional. No *framework* de LPS proposto por Clements e Northrop (2001) essas

atividades são sinônimas à *core asset development*, *product development* e *management*, respectivamente. Essas atividades essenciais em uma LPS são mostradas na Figura 1 e serão descritas a seguir.

Figura 1 – Atividades essenciais de LPS



Fonte: Pohl, Böckle e Linden (2005).

2.1.1.1 Engenharia de Domínio

O objetivo principal dessa atividade é estabelecer a capacidade de produção de novos produtos, definindo as similaridades e particularidades entre eles (POHL; BÖCKLE; LINDEN, 2005). Durante esta atividade os componentes e demais artefatos reusáveis da LPS são construídos. Para que os componentes possam se adaptar aos diversos produtos que se pretende gerar, são utilizados mecanismos de variabilidade. Os mecanismos de variabilidade podem ser concretizados em diversos níveis. Pohl, Böckle e Linden (2005) citam alguns desses mecanismos como: geração de código, programação orientada a aspectos, abordagens dirigidas por modelos, macros de pré-compilação,

compilação condicional, arquivos *make*, arquivos de configuração e registro.

Clements e Northrop (2001) mostram como essa atividade é iterativa e influenciada pelos fatores contextuais descritos a seguir:

2.1.1.1.1 Restrições de Produto

As restrições de produto dizem respeito às similaridades e particularidades que existem entre os produtos que constituem a LPS. As funcionalidades, ou *features,* vão definir quais são os comportamentos esperados para o produto, e também devem estar alinhadas com o mercado. Requisitos não funcionais também precisam ser levados em conta, como por exemplo: limites de desempenho, integrações com sistemas externos, restrições físicas e requisitos de qualidades. Os artefatos do repositório de *core assets* devem possuir pontos em comum e acomodar variações previstas sem abrir mão da qualidade do produto. Essas restrições podem ser derivadas de produtos existentes, podem ser geradas novas restrições, ou podem ser uma combinação de restrições novas e derivadas (CLEMENTS; NORTHROP, 2001).

2.1.1.1.2 Restrições de Produção

As restrições de produção levantam questões como: em quanto tempo um novo produto deve chegar ao mercado; qual deve ser a capacidade de produção; quais padrões específicos precisam ser seguidos durante a produção do produto; quem irá construir os produtos; e qual ambiente será utilizado. Essas questões irão guiar decisões sobre: geração de código ou codificação manual; tipo de mecanismo de variabilidade fornecido nos *core assets*; processo de produção usado nos produtos; entre outros (CLEMENTS; NORTHROP, 2001).

2.1.1.1.3 Estratégia de produção

A estratégia de produção define a abordagem que será usada para construir os artefatos do repositório de *core assets* e os produtos individuais. Existem três principais estratégias para a construção de uma LPS: proativa, reativa e extrativa. A abordagem mais tradicional é proativa, onde o escopo, a arquitetura, os componentes e outros recursos, são definidos nas fases iniciais, antecipando semelhanças e particularidades.

Na abordagem reativa, a LPS cresce incrementalmente adquirindo variabilidade sob demanda. Pohl, Böckle e Linden (2005) mostram o estudo de caso da Salion Inc., uma empresa que aplicou com sucesso a abordagem reativa, desenvolvendo um primeiro produto para depois evoluí-lo para uma LPS. A Salion Inc. observou uma grande economia de custos ao aplicar essa abordagem, além de ter apresentado o caminho mais rápido para ter uma LPS operacional.

Na abordagem extrativa alguns aplicativos são desenvolvidos em primeiro lugar, e em seguida, os artefatos do repositório de *core assets* são extraídos a partir deles (KRUEGER, 2001).

Essas abordagens não são exclusivas, e portanto é possível seguir uma abordagem incremental, que combina, em momentos diferentes, as ideias das diferentes abordagens.

Também faz parte da estratégia de produção definir: os preços dos produtos, a compra de componentes, o investimento na construção de componentes e o gerenciamento da produção. É durante a estratégia de produção que será definida a arquitetura e seus componentes associados e o caminho para o seu crescimento. A estratégia também guia o processo pelo qual os produtos serão desenvolvidos a partir dos *core assets* (CLEMENTS; NORTHROP, 2001).

2.1.1.1.4 Artefatos pré-existentes

Os artefatos pré-existentes, como sistemas legados ou outros produtos existentes, influenciam diretamente no conhecimento que a organização possui sobre um determinado domínio e/ou define a sua presença no mercado. Possuir esse conhecimento e/ou presença pode ajudar na construção de uma LPS, principalmente porque a arquitetura da LPS pode se basear nesses projetos já comprovados.

Componentes da LPS também podem ser minerados a partir desses outros projetos, e são uma forma de representar a propriedade intelectual da organização em domínios relevantes. Esses componentes tornam-se então, parte do repositório de *core assets* da LPS.

Outros recursos também podem ser reaproveitados pela LPS como: bibliotecas, *frameworks*, algoritmos, ferramentas, componentes e serviços. Através de uma análise cuidadosa, uma organização determina o que é mais apropriado de usar. No entanto, artefatos pré-existentes não são limitados aos artefatos construídos pela organização. Softwares disponíveis externamente podem ser importados de fora da organização e utilizados para agilizar a construção da LPS (CLEMENTS; NORTHROP, 2001).

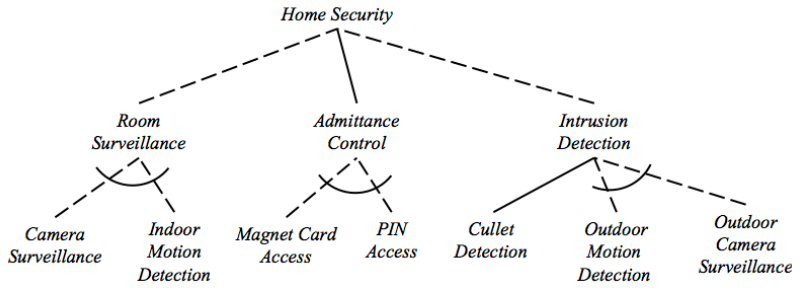
Esses fatores vão influenciar diretamente no resultado da engenharia de domínio. Ao final dessa fase tem-se o resultado necessário para a produção dos produtos da LPS. Clements e Northrop (2001) mostram que essa atividade produz três saídas, que são descritas a seguir:

2.1.1.1.5 Contexto da LPS

No escopo fica a descrição dos produtos possíveis de serem construídos pela LPS, podendo ser, por exemplo, uma lista de nomes de produtos. Outra forma de visualizar as capacidades da LPS, em termos de similaridades e particularidades, é por meio de um *feature model*.

Segundo Pohl, Böckle e Linden (2005), o *feature model* serve como documentação das *features* de uma LPS e facilita a discussão entre os interessados no projeto. Nesse modelo também é possível visualizar características de cada *feature*, por exemplo, se é uma *feature* obrigatória, opcional ou alternativa. A Figura 2 mostra o *feature model* de um sistema de automatização residencial onde é possível ver que a LPS oferece *features* como: vigilância de cômodos, controle de acesso e detecção de invasores.

Figura 2 - *Feature model* de um sistema de automatização residencial



Fonte: Pohl, Böckle e Linden (2005).

Para o sucesso da LPS, o escopo precisa ser bem definido. Segundo Clements e Northrop (2001), com um escopo muito grande, onde uma LPS admite muitas variações:

> Os artefatos do repositório de *core assets* serão estressados além da sua capacidade de acomodar

> variações, as economias da produção serão perdidas, e a LPS irá entrar em colapso se tornando um esforço de produzir um produto por vez (CLEMENTS; NORTHROP, 2001).

Enquanto com um escopo muito pequeno:

> Os artefatos do repositório de *core assets* podem não construir uma arquitetura genérica suficiente para acomodar um crescimento futuro, e a linha irá estagnar, economias no escopo não vão se concretizar e o potencial de retorno de investimento nunca vai se materializar (CLEMENTS; NORTHROP, 2001).

2.1.1.1.6 Repositório de Core Assets

Para Clements e Northrop (2001) o repositório de *core assets* é um base que inclui todos os artefatos que servem como a fundação para a produção dos produtos de uma LPS. Nem todos os artefatos são usados em todos os produtos da LPS, porém, todos eles devem ser usados o suficiente para justificar o seu desenvolvimento e sua manutenção.

Entre os artefatos mais comuns em uma LPS tem-se a arquitetura que os produtos irão compartilhar e os componentes que são desenvolvidos para serem reusados de forma sistemática por meio da LPS. Outros tipos de artefatos incluem: planos de teste, casos de teste, documentação de projeto, especificações de requisito, modelos do domínio, escopo da LPS, *web services*, componentes, *Commercial Off-The-Shelf* (COTS), cronogramas, orçamentos, planos, infraestrutura de produção, linguagens específicas de domínio, ferramentas, geradores e ambientes (CLEMENTS; NORTHROP, 2001).

2.1.1.1.7 Plano de Produção

O plano de produção descreve como os produtos são produzidos a partir dos *core assets*. Por exemplo, uma variabilidade pode ser alcançada selecionando componentes através de uma lista de componentes ou serviços para fornecer uma funcionalidade, adicionando ou removendo componentes, ou criando um ou mais componentes por herança ou parametrização, ou usando aspectos (CLEMENTS; NORTHROP, 2001). O plano de produção também descreve os detalhes

do projeto para permitir a execução e a gestão do processo e, portanto, inclui detalhes como o cronograma, lista de materiais e métricas (CLEMENTS; NORTHROP, 2001).

2.1.1.2 Engenharia de Aplicação

A engenharia de aplicação consiste no desenvolvimento dos produtos por meio do reuso sistemático dos artefatos do repositório de *core assets* derivando os pontos de variação da LPS. O ponto de variação indica um local da LPS que possui variabilidade, como por exemplo a cor de um botão, que pode ser de diversas cores. Durante a engenharia de aplicação, uma dessas opções é escolhida, por exemplo cor vermelha, que é então denominada variante (POHL; BÖCKLE; LINDEN, 2005).

Caso haja necessidade de alguma funcionalidade que não foi prevista na engenharia de domínio, é feita uma análise para verificar se a nova funcionalidade deve fazer parte do repositório de *core assets* ou se será desenvolvida somente para aquele produto (CLEMENTS; NORTHROP, 2001).

As entradas da engenharia de aplicação, definidas por Clements e Northrop (2001) são as seguintes:

- A descrição do produto, normalmente expressada como uma derivação da descrição de um produto genérico contida no escopo da LPS.
- O escopo da LPS, que indica se é viável incluir o produto na LPS.
- O repositório de *core assets* do qual o produto deverá ser construído.
- O plano de produção, que detalha como os artefatos devem ser usados para construir o produto.

Os desenvolvedores do produto irão usar os artefatos de acordo com o plano de produção para produzir o produto para que atinja os requisitos especificados. Eles também têm a obrigação de fornecer um *feedback* sobre qualquer problema ou deficiência encontrada com os artefatos do repositório de *core assets*, para que os artefatos continuem funcionando.

2.1.1.3 Gerenciamento

De acordo com Clements e Northrop (2001):

> O gerenciamento cumpre um papel crítico para a aplicação bem sucedida da LPS. As atividades devem ser dotadas de recursos e, em seguida, serem coordenadas e supervisionadas. Ambos níveis de gerenciamento técnico e organizacional precisam ser fortemente comprometidos com o esforço de linha de produto de software. Esse comprometimento manifesta por si em diversas formas que alimentam o esforço de linha de produto e a mantem funcional. (CLEMENTS; NORTHROP, 2001)

O gerenciamento organizacional determina a estratégia de produção. É por meio do gerenciamento organizacional que é criada a estrutura organizacional necessária para adotar uma abordagem de LPS. Essa estrutura está relacionada com a alocação de recursos, definição dos responsáveis, mitigação de riscos, relação com parceiros e fornecedores, entre outros. Uma das coisas mais importantes do gerenciamento organizacional é criação de um plano de adoção que descreve onde a organização pretende chegar e a estratégia para alcançar esse objetivo (CLEMENTS; NORTHROP, 2001).

O gerenciamento técnico supervisiona o desenvolvimento dos artefatos da base e dos produtos por meio da garantia de engajamento nas atividades daqueles que construirão os artefatos e os produtos, seguindo os processos definidos pela LPS, e coletar dados suficientes para rastrear o progresso. O gerenciamento técnico decide sobre o método de produção e fornece os elementos do gerenciamento de projeto do plano de produção (CLEMENTS; NORTHROP, 2001).

2.2 ENGENHARIA DE LINHA DE PRODUTO ÁGIL

Há várias razões para combinar práticas de LPS com as práticas de desenvolvimento ágil de software (DÍAZ et al., 2011). Díaz et al. (2011) afirma que essa combinação pode ser aplicada quando:

1. Não há muito conhecimento sobre o domínio para executar a engenharia de domínio;
2. Não é possível prever as mudanças que serão necessárias nos requisitos dos produtos;

3. Existe uma necessidade de diminuir o risco de desenvolvimento de produtos que podem não ser reutilizados, devido às modificações de mercado.

O uso de métodos ágeis em LPS não é visto como uma novidade. De acordo com Silva et al. (2011), o Scrum e *Extreme Programming* (XP) são os métodos ágeis mais utilizados no desenvolvimento de LPSs. Alguns estudos combinam ambas, Scrum e XP, devido ao fato de que a natureza desses métodos tem focos diferentes. Enquanto o Scrum se concentra na gestão do projeto, o XP se concentra em práticas de desenvolvimento. Uma das mais famosas práticas do XP é o desenvolvimento dirigido por testes (*Test-Driven Development* - TDD).

TDD é uma forma de programação, onde as tarefas de codificação são realizados em pequenos ciclos de acordo com o mantra *red/green/refactor* (BECK, 2002). Na fase *red*, um teste unitário falho é escrito. Na fase *green*, o código é alterado da maneira mais simples para que o teste apenas passe. E na última fase, *refactor*, o código é alterado, a fim de melhorar e manter o comportamento. É na fase *refactor* que as decisões de *design* são feitas, uma de cada vez.

O TDD também auxilia na prevenção de erros e pode servir como documentação do sistema. Segundo Beck (2002), o TDD pode auxiliar no desenvolvimento de *frameworks* porque foca no que é necessário, sem desenvolver código que no futuro pode não ser usado. À medida que surgem novas funcionalidades, o código é testado e reformulado, eliminando código duplicado. O código comum é colocado separado do código específico, facilitando a reutilização do código comum depois.

2.3 TESTES EM LINHAS DE PRODUTO DE SOFTWARE

É muito difícil escrever software livre de defeitos, uma vez que é impraticável ou impossível testar todas as entradas possíveis para exercitar um programa (NETO et al., 2011). De acordo com Clements e Northrop (2001), testes de software têm duas funções principais: (1) ajudar na identificação de problemas, de forma que possa ser reparado, e (2) determinar se o software funciona, conforme especificado pelos seus requisitos.

Além disso, Meszaros (2007) acredita que os testes devem ajudar a melhorar a qualidade, ajudar a entender o sistema sendo testado, reduzir riscos, e ao mesmo tempo, os testes devem ser fáceis de executar, ser fáceis de escrever e manter, e exigir o mínimo de manutenção, conforme o produto evolui.

A atividade de teste ocorre continuamente por meio de todo o processo de desenvolvimento de software. É uma atividade cara e pode custar até cinco vezes o esforço de desenvolvimento (CLEMENTS; NORTHROP, 2001). Assim, é uma área que se mostra atraente para ser trabalhada, com a finalidade de reduzir este alto custo (CLEMENTS; NORTHROP, 2001). Mas, apesar da importância desta área, Neto et al. (2011) afirmam que a atividade de teste de software não é tão avançada como as atividades de desenvolvimento de software e o mesmo acontece com a área de testes dentro da LPS.

Na LPS, a atividade de teste tem que examinar os *core assets*, o código específico de um produto, as interações entre eles, e os produtos individuais (CLEMENTS; NORTHROP, 2001). Pode tornar-se uma atividade mais complexa e dispendiosa conforme a variedade de produtos derivados da LPS aumenta (ENGSTRÖM; RUNESON, 2011). Isso faz com que os testes na LPS exijam maior custo do que os testes em sistemas individuais (NETO et al., 2011). Por isso, alguns autores consideram a área da atividade de testes um gargalo no desenvolvimento de LPSs (NETO et al., 2011).

Alguns estudos dividem as atividades de teste de acordo com as atividades de desenvolvimento da LPS: testes de domínio na engenharia de domínio e testes de aplicação na engenharia de aplicação (LEE; KANG; LEE, 2012). Durante a fase de testes de domínio, os testes são criados para testar os artefatos do repositório de *core assets*, mas também são produzidos com o objetivo de serem reutilizados pelos aplicativos durante os testes de aplicação. Durante a fase de testes de aplicação, artefatos específicos usados no produto são testados e os artefatos no repositório utilizados no produto são testados novamente. Isto é necessário porque quando os artefatos da base são integrados a uma aplicação específica, as interações com recursos específicos do aplicativo podem causar falhas.

Além de ser mais caro do que os testes de sistema único, o processo de teste em LPSs difere claramente do processo de testes de um sistema único. O processo de teste na LPS é mais complexo porque a LPS também inclui a dimensão do que tem que ser testado em conjunto na engenharia de domínio e o que tem que ser testado separadamente na engenharia de aplicação para cada aplicação (ENGSTRÖM; RUNESON, 2011). Contudo, a maior diferença é devido à variabilidade dos artefatos do domínio. Lidar com a variabilidade é um grande desafio e influencia diretamente a atividade de testes na LPS (LEE; KANG; LEE, 2012).

Na verdade, artefatos de teste de domínio devem ser tão variáveis quanto os artefatos de domínio (CLEMENTS; NORTHROP, 2001). Entre os artefatos de teste do domínio podemos citar: suítes de teste, casos de teste e planos de teste. Eles têm que ser planejados considerando a variabilidade para que sejam eficientemente reutilizados nos testes de aplicação. Clements e Northrop (2001) dizem que:

> Pesquisas apontam o uso do mesmo mecanismo de variabilidade usado na implementação de produtos para implementar variações no software de teste (CLEMENTS; NORTHROP, 2001).

Durante o desenvolvimento de uma aplicação de uma LPS, existem duas utilizações essenciais para os testes: (1) verificar que o que foi produzido em uma iteração é correto e adequado para ser usado na próxima iteração e (2) validar se o produto funciona como especificado (CLEMENTS; NORTHROP, 2001).

2.4 TESTES UNITÁRIOS

Existem diferentes níveis de teste que podem ser aplicados em software. McGregor (2001) descreve três níveis de testes: testes unitários, testes de integração e testes de sistema. Para Pohl, Böckle e Linden (2005) o nível de teste é “definido pela granularidade dos itens a serem testados”.

No processo de software, os desenvolvedores também são responsáveis por realizar testes no seu código para cada unidade. As "unidades" podem ser componentes de maior granularidade ou podem ser as menores unidades testáveis, como classes individuais, métodos ou funções (MESZAROS, 2007). Para cada unidade construída, um teste unitário deve garantir que ela (1) faz tudo o que sua especificação descreve e (2) não faz qualquer coisa que não deve fazer (CLEMENTS; NORTHROP, 2001). O teste unitário é geralmente o primeiro nível de teste de software (GANESAN et al., 2013).

A motivação para utilizar testes unitários é o fato de que o custo de encontrar e corrigir um problema durante a fase de testes unitários é muito menor do que encontrar e corrigir problemas encontrados durante os testes de integração, testes de sistema ou quando o produto já está em produção (GANESAN et al., 2013). Além disso, testes unitários fazem com que os testes de regressão, descritos na seção 2.4.1, sejam mais fáceis, pois sempre que forem feitas alterações no software, testes

unitários podem ser executados para permitir que os desenvolvedores verifiquem que não quebraram os recursos existentes (GANESAN et al., 2013).

2.4.1 Testes de Regressão

Clements e Northrop (2001) definem que:

> Os testes de regressão são usados para determinar se o software que está sendo testado e que exibiu o comportamento esperado antes de uma mudança, continua a exibir esse comportamento após a mudança (CLEMENTS; NORTHROP, 2001).

Esses testes devem ser executados periodicamente, ou desencadeados por alterações, para determinar se o software continua correto e consistente ao longo do tempo (CLEMENTS; NORTHROP, 2001).

Apesar do elevado benefício dos testes de regressão, Neto et al. (2011) observaram que não há nenhuma evidência de práticas de testes de regressão no contexto de LPS. Neto et al. (2011) também afirmam que os testes de regressão não pertencem a nenhuma atividade específica no processo de desenvolvimento de software e, por isso, não há clareza sobre como os testes de regressão devem ser realizados.

O maior valor dos testes de regressão é o *feedback* rápido que é dado ao modificar aplicações existentes, porque ajuda os desenvolvedores a capturar defeitos que poderiam ter sido introduzidos no código (MESZAROS, 2007). McGregor (2001) aconselha que as suítes de testes de regressão sejam tão automatizadas quanto possível para obter um *feedback* mais rápido com menos esforço.

2.4.2 Automação de testes unitários

De acordo com Meszaros (2007), "a verificação automatizada de comportamento de software é um dos maiores avanços nos métodos de desenvolvimento nas últimas décadas". Assim como nos testes de regressão, os testes unitários para serem eficazes devem ser totalmente automatizados.

Os objetivos da automação de testes incluem a redução de custos, a melhoria da qualidade e a melhoria da compreensão do código. Neto et

al. (2011) destacam que as ferramentas de automação de testes, que dão suporte às atividades de teste, são uma forma de alcançar a redução do esforço em testes em LPS.

Existem vários *frameworks* para automação de testes unitários, entre eles JUnit[2], XCTest, OCUnit[3], PHPUnit[4]. Os objetivos destes *frameworks* são:

- Facilitar que os desenvolvedores escrevam testes sem aprender uma nova linguagem de programação. Hoje em dia os *frameworks* de automação de testes para testes unitários estão disponíveis para a maioria das linguagens de programação;
- Facilitar os testes unitários individuais sem que todos os componentes estejam disponíveis. Este tipo de *framework* de testes é projetado para permitir testar o software de dentro para fora, isto é, das classes mais internas até a interface;
- Executar um teste ou uma suíte de testes com uma única ação, mantendo-o simples;
- Diminuir o custo de execução dos testes para que os desenvolvedores mantenham a coragem para executar os testes com frequência.

2.5 PADRÕES DE TESTE UNITÁRIO

Um padrão é descrito como uma solução para um problema recorrente num dado contexto. Padrões de teste unitário descrevem soluções relacionadas com a escrita, execução e manutenção de testes no contexto de testes unitários (MESZAROS, 2007). No entanto, alguns problemas são maiores do que os outros e, portanto, não é possível resolver o problema com apenas um único padrão de teste.

Meszaros (2007) catalogou 68 padrões de teste unitário que abordam diferentes tipos de problemas recorrentes. Esses padrões são divididos em três níveis diferentes: (1) padrões de nível de estratégia, que definem informações importantes sobre a forma de executar os testes; (2) padrões de nível de *design*, que definem a forma de organizar a lógica de teste; e (3) idiomas de codificação, que descrevem diferentes formas de codificar um teste específico.

---

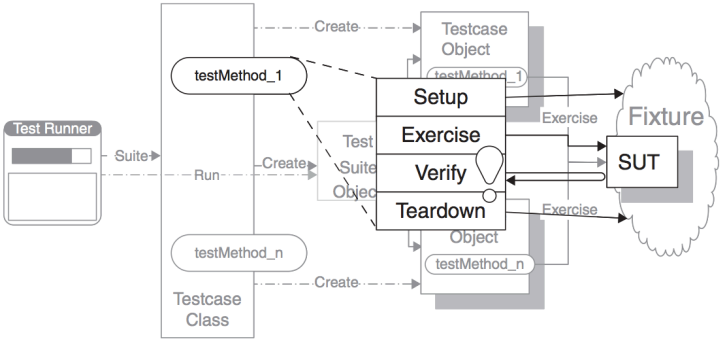
[2] http://junit.org
[3] http://cocoadev.com/OCUnit
[4] https://phpunit.de

Na presente dissertação é proposta uma nova estratégia de teste unitário, direcionada para testes unitários automatizados em LPS, que une dois padrões de nível estratégico definidos por Meszaros (2007): o *Framework* de Automação de Teste (*Test Automation Framework* – TAF) e o Teste Guiado por Dados (*Data-Driven Test* – DDT). No entanto, antes de descrever estes padrões de teste, é importante detalhar o padrão básico de teste unitário Teste de Quatro Fases (*Four-Phase Tests*), necessário para estruturar os testes unitários.

2.5.1 Teste de Quatro Fases (*Four-Phase Tests*)

Meszaros (2007) afirma que cada teste unitário deve ser estruturado em quatro etapas distintas. Em primeiro lugar, os dados do teste são configurados para que o sistema em teste possa estar preparado para apresentar o comportamento esperado (*setup)*. Em segundo lugar, o Sistema Sob Teste (*System Under Test* – SUT) é exercitado (*exercise*). Em terceiro lugar, é verificado se o resultado foi o esperado (*verify*). E, finalmente, no quarto passo, os dados são limpos para que o sistema volte para o mesmo estado que estava antes da execução do teste (*teardown*). Essa estrutura torna mais fácil para o leitor identificar qual o comportamento está sendo verificado. Na Figura 3 é possível ver à esquerda o *Test Runner* que é o responsável pela execução das suítes de testes. As suítes de testes vão ser geradas a partir de classes de casos de teste. Na Figura 3 é ilustrada uma classe de casos de teste mostrada pelo retângulo descrito como *Testcase Class*. Cada uma dessas classes pode conter diversos métodos de teste. A Figura 3 destaca, então, esse padrão para um teste unitário chamado *testMethod_1* que passa pelas quatro fases: *setup*, *exercise*, *verify* e *teardown*.

Figura 3 – Exemplificação do Teste de Quatro Fases



Fonte: Meszaros (2007).

2.5.2 *Framework* de Automação de Teste (*Test Automation Framework*)

Um dos problemas de escrita e execução de testes é a necessidade de repetir os mesmos passos. Assim é possível que os testes se tornem muito tediosos, consumam muito tempo, estejam mais sujeitos a erros e se tornem mais caros. O padrão *Framework* de Automação de Testes é uma forma de minimizar o esforço de escrever testes totalmente automatizados que podem ser realizados sem a intervenção manual.

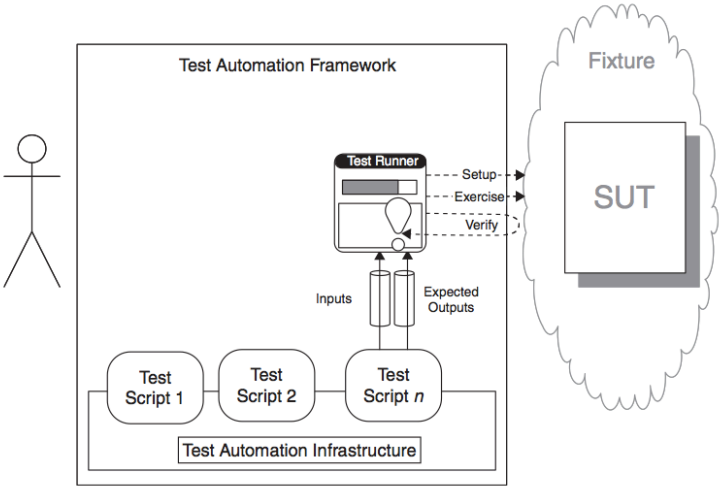
O *Framework* de Automação de Testes precisa implementar todos os mecanismos necessários para executar suítes de testes e registrar os resultados. Esses mecanismos estão relacionados com a capacidade de encontrar testes individuais, reuní-los em um conjunto de testes, executar um teste por vez (seguindo o padrão de Teste de Quatro Fases - seção 2.5.1), verificar a saída esperada, coletar e relatar quaisquer falhas ou erros no teste e limpar as coisas quando ocorrem falhas ou erros.

Ter um *framework* reduz o custo dos testes por meio do fornecimento dos recursos mais comuns. O único custo é o de aprender a usar o *framework*, mas, como Neto et al. (2011) mencionam "os recursos necessários para automatizar são amortizados durante o maior número de produtos".

O padrão *Framework* de Automação de Testes separa a implementação da lógica necessária para executar os testes, da lógica

dos testes. Isso ajuda o código de teste a se tornar mais claro, facilitando sua manutenção. Ele também ajuda a reduzir a duplicação de código de teste e garante que o teste escrito em um mesmo documento por diferentes desenvolvedores possa ser executado facilmente, gerando um único relatório com os resultados do teste. A Figura 4 mostra as partes importantes do *framework*, como a infraestrutura e os *n scripts* que possuem as entradas necessárias para configurar os testes. Ainda na Figura 4 é possível ver o fluxo das entradas em direção ao *test runner* que irá executar os testes de acordo com o padrão de Teste de Quatro Fases (seção 2.5.1) e irá verificar a saída real do sistema sob teste (*System Under Test* – SUT) com a saída esperada que também está no *script*.

Figura 4 – Partes importantes do *Framework* de Automação de Testes



Fonte: Meszaros (2007).

Meszaros (2007) afirma que o padrão Teste Guiado por Dados é uma categoria de *Framework* de Automação de Testes.

2.5.3 Teste Guiado por Dados (*Data-Driven Test*)

Com o padrão Teste Guiado por Dados, novos testes podem ser reduzidos a uma linha no arquivo em vez de exigir uma série de etapas

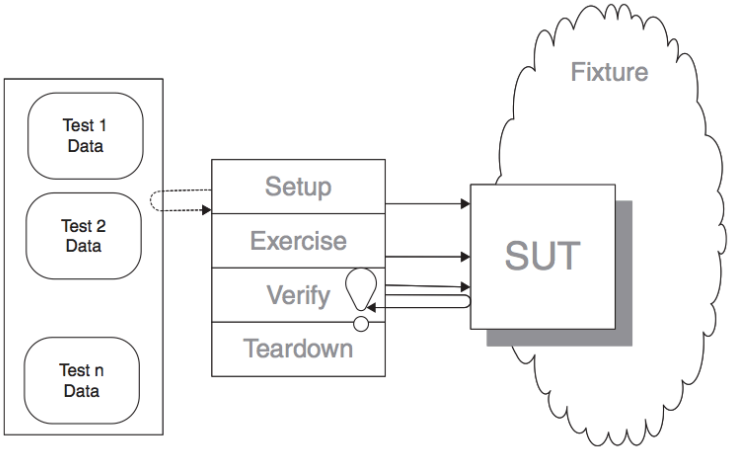
complexas e repetitivas. Por isso, os testes são organizados em duas partes: o interpretador e os arquivos de dados.

Os testes podem ser muito repetitivos porque precisam ser executados várias vezes, mas eles também podem ser repetitivos porque apresentam uma lógica em comum. Por vezes, a única diferença entre a lógica de diferentes testes é a entrada e saída de dados do sistema. Assim, a lógica de teste pode consistir de exatamente os mesmos passos (MESZAROS, 2007).

Ter vários testes é uma boa maneira de garantir uma cobertura melhor de funcionalidade, mas pode dificultar a manutenção dos testes, uma vez que uma mudança na lógica de testes pode resultar em mudanças que precisam ser propagadas para todos os testes similares. O padrão de Testes Guiados por Dados é uma forma de manter a cobertura de testes, enquanto diminui a quantidade de código de teste necessário para escrever e manter (MESZAROS, 2007).

Para implementar o padrão de Testes Guiados por Dados, é necessário um interpretador (GAMMA et al., 1994), que contém toda a lógica responsável por ler os parâmetros e chamar os testes que têm a lógica comum. Os dados que variam de teste para teste vão para um arquivo que é lido pelo interpretador para executar os testes. Cada entrada neste arquivo dispara a execução de um teste usando o padrão de Teste de Quatro Fases (Seção 2.5.1). Em primeiro lugar, o interpretador lê os dados de teste a partir do arquivo e configura o teste utilizando os dados lidos. Em segundo lugar, ele exercita o sistema sob teste com os argumentos especificados no arquivo. Em terceiro lugar, ele compara os resultados reais produzidos do sistema em teste com os resultados esperados descritos no arquivo. Se os resultados não são os esperados, o teste é marcado como falho. Em quarto lugar, o interpretador faz qualquer limpeza necessária e, em seguida, move-se para a próxima entrada no arquivo (MESZAROS, 2007). A Figura 5 mostra um retângulo branco mais a esquerda com 3 *Test Data's*. Esses dados irão alimentar um *framework* de testes que irá disparar métodos de teste que seguem o padrão de Teste de Quatro Fases.

Figura 5 – Dados utilizados para disparar testes de quatro fases



Fonte: Meszaros (2007).

Meszaros (2007) destaca que:

> Também é importante executar esses testes como parte do processo de integração contínua para confirmar que os testes que passaram uma vez não comecem a falhar repentinamente (MESZAROS, 2007).

Assim, esses testes podem ser utilizados como testes de regressão.

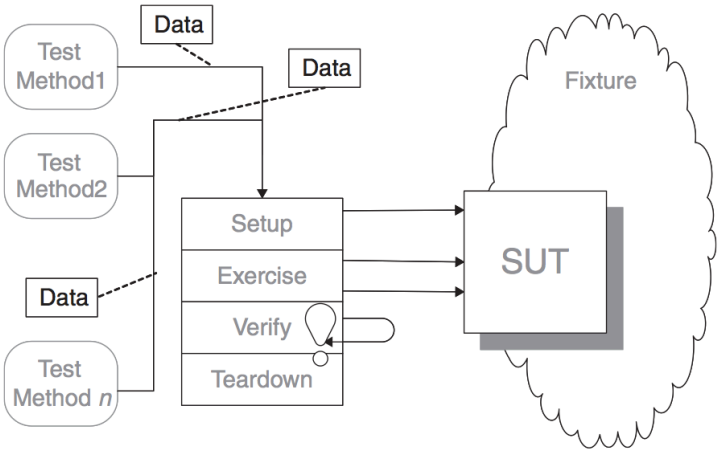
O padrão de Testes Guiados por Dados usa outros dois padrões, uma vez que executa a lógica que recebe os parâmetros provenientes do arquivo. Esses padrões são chamados de Testes Parametrizados e Método Utilitário de Teste e são descritos a seguir.

2.5.4 Teste Parametrizado (*Parameterized Test*)

O padrão Teste Parametrizado sugere a reutilização da mesma lógica de teste em uma quantidade grande de testes. A lógica comum é extraída em um método utilitário que usa somente as informações que diferem de um teste para outro, como seus argumentos. Os parâmetros

de teste são as informações que variam de teste para teste, e que são necessários para os testes com parâmetros a serem executados. Este padrão oferece uma boa cobertura e também reduz o código a ser mantido, tornando fácil de adicionar mais testes conforme necessário. A Figura 6 mostra os testes recebendo dados (*data*) através de parâmetros, antes de disparar a execução do teste.

Figura 6 – Testes que recebem dados por parâmetros

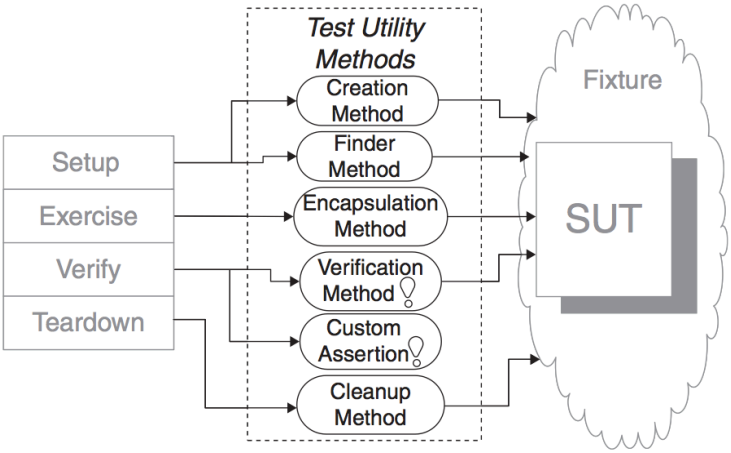


Fonte: Meszaros (2007).

2.5.5 Método Utilitário de Teste (*Test Utility Method*)

O padrão Método Utilitário de Teste serve para encapsular a lógica de teste que será reutilizada. Esse padrão ajuda a evitar a duplicação de código de teste. O Método Utilitário de Teste vai facilitar a extração da lógica comum do teste para ser executado pelo Teste Dirigido por Dados. A Figura 7 mostra diversos exemplos de métodos utilitários que podem ser usados nos testes unitários, como: métodos de criação, métodos de busca, métodos de encapsulação, métodos de verificação, asserções customizadas, e métodos de limpeza.

Figura 7 – Exemplos de métodos utilitários de teste



Fonte: Meszaros (2007).

3 ESTADO DA ARTE

Para o levantamento do estado da arte foi feita uma revisão sistemática. Segundo Kitchenham e Charters (2007):

> A revisão sistemática da literatura tem como objetivo avaliar e interpretar todas as pesquisas relevantes disponíveis sobre uma pergunta, área ou fenômeno, realizando, dessa forma, uma avaliação justa por meio de uma metodologia confiável, rigorosa e auditável (KITCHENHAM; CHARTERS, 2007).

A revisão feita nesta dissertação segue as diretrizes propostas por Kitchenham e Charters (2007) e é executada em três fases: planejamento, condução e relatório da revisão.

A razão para a execução desta revisão é mostrar como a prática de desenvolvimento dirigido por testes e como os testes unitários estão sendo usados em LPS, e identificar as lacunas existentes na utilização dessas práticas.

3.1 PLANEJAMENTO DA REVISÃO

Para o planejamento da revisão foi necessário definir as questões de pesquisa que a revisão irá responder e produzir o protocolo de revisão para definir qual é o procedimento de revisão dos trabalhos (KITCHENHAM; CHARTERS, 2007). Kitchenham e Charters (2007) dizem que o protocolo de revisão precisa especificar os métodos utilizados para conduzir a revisão, a fim de reduzir algum tipo de tendenciosidade. O protocolo é composto pelos seguintes elementos, que serão descritos nas seções seguintes:

- Questões de pesquisa;
- Estratégia de busca;
- Critério de seleção;
- Avaliação de qualidade;
- Extração dos dados.

3.1.1 Questões de Pesquisa

A definição das perguntas é o passo mais importante, pois guia toda a metodologia da revisão sistemática (KITCHENHAM;

CHARTERS, 2007). Para esta dissertação, as seguintes questões foram definidas:

Q1. De quais formas tem-se aplicado a prática de TDD em LPS?
Q2. Quais são os principais desafios e lacunas na aplicação das práticas de TDD em LPS?
Q3. Quais estratégias e práticas são utilizadas na aplicação de testes unitários em LPS?

3.1.2 Estratégia de busca

Nesta seção será descrita a estratégia usada para buscar os estudos primários, incluindo os termos de busca e as bases de dados. Foi feita uma busca manual por trabalhos em bibliotecas digitais. As bases de dados estão expostas no quadro a seguir (Quadro 1):

Quadro 1 - Bases de dados e seus endereços eletrônicos

Nome da base	**Link da base**
IEEE Xplore	http://ieeexplore.ieee.org
ACM Digital Library	http://portal.acm.org
Scopus	http://www.scopus.com
CiteSeerX	http://citeseerx.ist.psu.edu

Os termos de busca estão no Quadro 2 a seguir. Os termos foram separados para facilitar as buscas dos trabalhos para responder as perguntas elaboradas. O termo 1 visa encontrar trabalhos para responder as perguntas Q1 e Q2, enquanto o termo 2, visa encontrar trabalhos para responder a pergunta Q3. Para cada uma das bases foram feitos ajustes. Os termos utilizados em cada base encontram-se no Apêndice A. Os termos são buscados apenas no título, resumo e palavras-chave.

Quadro 2 - Termos utilizados na busca

Termos de busca
1. ("test-driven" OR "test driven" OR "test first" OR tdd) AND (SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families")))
2. ("unit test" OR "unit testing") AND (SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families")))

3.1.3 Critérios de Seleção

Nesta etapa, foram definidos os critérios que determinam quais estudos serão incluídos na revisão e quais serão excluídos. O critério de inclusão é qualquer material científico escrito em inglês que está dentro dos termos de busca definidos acima, e que foi publicado entre janeiro de 2004 e julho de 2014. Além disso, serão incluídos somente os trabalhos que apresentem alguma proposta ou abordagem de uso da prática de TDD ou do uso de testes unitários. Os critérios de inclusão foram aplicados somente nos trabalhos primários. O critério de exclusão é qualquer estudo que não se enquadre no critério de inclusão.

3.1.4 Extração dos Dados

Após a busca, é realizado um fichamento de cada trabalho encontrado na pesquisa colocando as seguintes informações em forma de tabela:

- Título
- Fonte
- Ano
- Autores
- Sumário

3.2 CONDUÇÃO DA REVISÃO

Essa fase inclui a execução dos seguintes passos: buscar estudos primários, selecionar estudos de acordo com o critério de inclusão e exclusão, avaliar a qualidade do trabalho e extrair os dados desses trabalhos.

3.2.1 Busca por Estudos

Seguindo os passos descritos anteriormente foi feita uma busca pelos trabalhos primários. Houve um retorno de 34 trabalhos (Apêndice B), porém haviam 6 trabalhos repetidos em bases diferentes, resultando em 28 trabalhos. O Quadro 3 mostra quantos trabalhos retornaram, incluindo os redundantes. Após a remoção das redundâncias, foram selecionados os trabalhos para serem avaliados na etapa seguinte.

Quadro 3 - Resultado das buscas por trabalhos primários

Base de dados	**Resultados**	**Incluídos**
Termo 1		
IEEE Xplore	4	2
ACM Digital Library	0	0
Scopus	8	2
CiteSeerX	2	1
Termo 2		
IEEE Xplore	1	1
ACM Digital Library	1	0
Scopus	15	4
CiteSeerX	3	0

3.2.2 Seleção dos Estudos

Após a seleção dos trabalhos não duplicados para as perguntas de pesquisa, foram aplicados os critérios de inclusão e exclusão. Avaliando o título e o resumo foram excluídos 19 trabalhos. Após a leitura dos documentos, outros 2 trabalhos foram excluídos, totalizando 21 trabalhos rejeitados. Após esse processo, um total de 7 trabalhos foram incluídos, e seus fichamentos podem ser vistos no Apêndice C. Os trabalhos selecionados e as respostas relacionadas são mostrados no quadro a seguir.

Quadro 4 – Trabalhos selecionados e relação com as perguntas de pesquisa

Título	**Questões**
A product line based aspect-oriented generative unit testing approach to building quality components (FENG; LIU; KERRIDGE, 2007)	Q3
Toward variability-aware testing (KÄSTNER et al., 2012)	Q3
Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach (GHANAM; MAURER, 2010)	Q1 e Q2
Architecture-Based Unit Testing of the Flight Software Product Line (GANESAN et al., 2010)	Q3
Extreme Product Line Engineering: Managing Variability & Traceability via Executable Specifications (GHANAM; MAURER, 2009)	Q1 e Q2
Product Line Variability with Elastic Components and Test-Driven Development (KAKARONTZAS; STAMELOS; KATSAROS, 2008)	Q1 e Q2
An analysis of unit tests of a flight software product line (GANESAN et al., 2013)	Q3

3.3 RELATÓRIO DA REVISÃO

Como mostrado anteriormente, a revisão sistemática retornou 7 trabalhos científicos. As respostas às perguntas de pesquisas são apresentadas nas subseções a seguir.

3.3.1 Q1. De quais formas tem-se aplicado a prática de TDD em LPS?

Ghanam e Maurer (2009, 2010) propõem a utilização de uma abordagem *test-driven* para introduzir variabilidade, refatorando o código existente. Dessa forma os testes guiam a inserção de novos comportamentos e novas formas de variabilidade sob demanda. Mas apesar de dizer que usa testes unitários e de aceitação para introduzir pontos de variação em uma LPS, Ghanam e Maurer (2009, 2010) não

explicam de qual forma os testes de aceitação são utilizados nesse processo. É nesse trabalho que é proposto a utilização do padrão de projeto *abstract factory* (GAMMA et al., 1994) no processo de refatoração para que novos pontos de variação sejam gerados. Também é citado que existem outras técnicas para inserir variabilidade, Ghanam e Maurer (2009, 2010) não detalham estas técnicas neste artigo.

Kakarontzas, Stamelos e Katsaros (2008) propõem uma abordagem sistemática para a criação de novos componentes de software variáveis por meio da customização dos *core assets* existentes de uma LPS. São consideradas ambas as variações de qualidade e variações funcionais. O trabalho de Kakarontzas, Stamelos e Katsaros (2008) utiliza o TDD para dar assistência na evolução dos componentes de software de uma LPS. Essa abordagem proposta por Kakarontzas, Stamelos e Katsaros (2008) sugere que seja criado um novo componente, que é uma extensão de um componente puro, quando novas funcionalidades forem adicionadas. Assim, quando não for necessária essa nova funcionalidade, basta utilizar o componente puro, ou de uma hierarquia superior. O problema desta abordagem é que quando existe uma longa hierarquia de componentes fica difícil escolher as funcionalidades individuais, já que elas estão atreladas aos componentes.

3.3.2 Q2. Quais são os principais desafios e lacunas na aplicação das práticas de TDD em LPS?

Ghanam e Maurer (2009, 2010) apontam que existem outros padrões e técnicas para a geração de variabilidade em LPS, e que sua proposta deve funcionar bem para qualquer implementação de variabilidade, porém, não apresenta dados que comprovem essa afirmação. Uma lacuna, então, seria a verificação de quais são os cenários onde apenas essa técnica para inserção de variabilidade não seria suficiente e apontar quais outras soluções poderiam resolver o problema.

Kakarontzas, Stamelos e Katsaros (2008) apresentam uma proposta que havendo a necessidade de adicionar novos casos de testes para que um componente satisfaça um novo requisito, um novo componente deve ser criado a partir do componente original. Essa proposta adiciona um novo ponto de complexidade por necessitar o gerenciamento de uma nova estrutura, que é essa hierarquia de componentes. Com uma hierarquia grande, pode ser difícil selecionar alternativas e opções de requisitos para compor um novo produto. Ainda

assim, o TDD cumpre um papel importante, pois é por meio dele que se desenvolve o mínimo necessário, e com os testes gerados nessa etapa é feita a verificação de que o componente cumpre com suas funcionalidade previstas. O desafio identificado seria continuar com as vantagens e simplicidade do TDD sem a adição do ponto de complexidade de adicionar novos componentes.

3.3.3 Q3. Quais estratégias e práticas são utilizadas na aplicação de testes unitários em LPS?

Sobre a utilização dos testes unitários dentro de uma LPS, quatro trabalhos endereçam esta questão e expõem estratégias e práticas para a aplicação destas práticas.

Kastner et al. (2012) investigaram como executar testes unitários para todos os produtos de uma LPS sem ter que gerar testes separados para cada produto possível pela LPS. Eles projetaram e implementaram um interpretador que possui conhecimento sobre a variabilidade. Esse interpretador represente a variabilidade localmente nas suas estruturas de dados. Além disso, eles recodificaram os pontos de variação da LPS para simular os casos de testes com um modelo de checagem.

Ganesam et al. (2010, 2012) apresentam uma análise da abordagem de teste unitário desenvolvida e utilizada pela LPS de sistemas de voo da NASA, com o objetivo de entender, revisar e recomendar estratégias para melhorar a infraestrutura existente de testes unitários do sistema de voo. Além disso, procuram capturar as lições aprendidas e as melhores práticas que podem ser utilizadas nos testes unitários de outras LPSs.

Feng, Liu e Kerridge (2007) propõem uma abordagem de testes unitários em LPSs que são baseadas em aspecto. Segundo esse trabalho, LPSs baseadas em aspecto facilitam a criação automática de casos de testes de aspecto que vão verificar requisitos de qualidade. Feng, Liu e Kerridge (2007) reforçam que as abordagens existentes de teste unitário são fracas no que diz respeito aos testes de requisitos de qualidade. A abordagem de Feng, Liu e Kerridge (2007) é então adequada para verificar e melhorar os requisitos não funcionais de uma LPS, como dependências, desempenho, e padrões de programação. Feng, Liu e Kerridge (2007) também propõem uma ferramenta que ajuda na abordagem e na verificação.

4 PROPOSTA

A proposta desta dissertação é mostrar como as práticas ágeis de TDD e refatoração podem auxiliar na construção de LPSs de abordagem reativa, facilitando a evolução de uma LPS a partir de uma aplicação existente, e facilitando a execução e reutilização dos testes unitários.

Inicialmente é mostrado como uma LPS de abordagem reativa pode ser construída com o auxílio das práticas de TDD e refatoração. Em seguida é apresentado o *framework* desenvolvido para dar suporte ao uso destas práticas na construção de LPSs de abordagem reativa.

Nesta dissertação, foi escolhida a abordagem de desenvolvimento reativo para ser usada em conjunto com as práticas ágeis porque entre as suas vantagens está o custo mais baixo para o desenvolvimento, uma vez que os *core assets* não são construídos no início do processo (CLEMENTS; NORTHROP, 2001). Desta forma, o risco de desenvolvimento de componentes que não serão úteis é menor, uma vez que não há necessidade de proporcionar as variabilidades de produtos com antecedência e que o conhecimento sobre o domínio pode crescer à medida que as aplicações são desenvolvidas (NETO et al., 2011).

4.1 VISÃO GERAL DO USO DE TDD E REFATORAÇÃO EM LPSS DE ABORDAGEM DE DESENVOLVIMENTO REATIVO

Nesta proposta, assim como ocorre no desenvolvimento reativo de LPSs, a primeira aplicação deve ser desenvolvida focando somente em satisfazer os seus requisitos. O desenvolvimento desta aplicação deve ser dirigido por testes e estes testes são incluídos no repositório de *core assets* da LPS. Os requisitos desta primeira aplicação são, então, utilizados para montar o primeiro *feature model* da LPS.

Para as aplicações seguintes, as funcionalidades podem ser desenvolvidas de três maneiras diferentes. No primeiro caso, novas funcionalidades podem ser definidas como *features* novas, e neste caso o desenvolvimento também é dirigido por testes e os testes são incluídos no repositório de *core assets* da LPS. No segundo caso as funcionalidades podem modificar *features* existentes, e neste caso o código de produção[5] e os testes são refatorados para que as variantes sejam ativadas. No terceiro caso, a aplicação pode utilizar funcionalidades das aplicações anteriores, e neste caso as *features* já estão disponíveis no repositório de *core assets*.

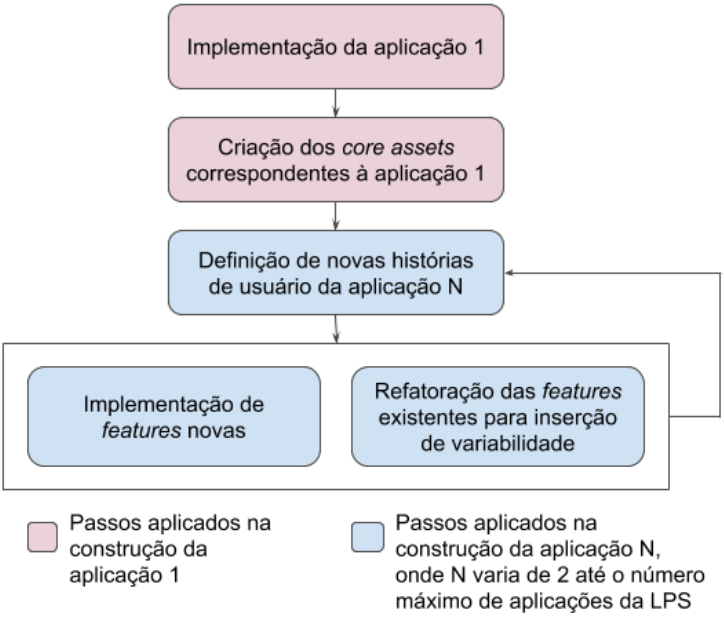
[5] Código de produção é o código que compõe o produto pronto

Durante a construção da aplicação seguinte, as *features* das aplicações anteriores que não são reusadas viram opções da LPS, e se forem independentes, elas não precisam ser compiladas em conjunto com o restante do código. Já as *features* que serão reusadas, podem ou não sofrer alterações e serão compiladas e configuradas pela classe *configurator*.

O mecanismo principal para ativar a variabilidade é o arquivo de configuração. Cada aplicação terá o seu próprio arquivo de configuração, ativando e configurando as *features* que serão incluídas e os correspondentes testes.

O fluxo da execução dos passos da abordagem proposta para a construção reativa de uma LPS utilizando as práticas TDD e refatoração é mostrado na Figura 8. As duas primeiras atividades são relacionadas ao desenvolvimento da primeira aplicação, que inicialmente não faz parte da LPS, enquanto as outras atividades são repetidas a medida que cada nova aplicação da LPS é construída.

Figura 8 – Diagrama mostrando os passos da abordagem proposta



4.1.1 Passo 1: Implementação da aplicação 1

A primeira aplicação é desenvolvida focando somente em satisfazer os seus requisitos. Esta aplicação é criada como se fosse a única aplicação, sem que sejam feitas tentativas de prever futuras variabilidades. O desenvolvimento desta aplicação é dirigido por testes e ao final do seu desenvolvimento tem-se o código fonte da aplicação e uma suíte de testes unitários. Essa suíte de testes é a garantia do funcionamento da aplicação.

4.1.2 Passo 2: Criação dos *core assets* correspondentes à aplicação 1

O código em produção e as suítes de teste da primeira aplicação se tornam artefatos do repositório de *core assets*, juntamente com qualquer outro artefato produzido, como requisitos, diagramas e documentação. As *features* da primeira aplicação são transcritas para o primeiro *feature model* da LPS.

4.1.3 Passo 3: Definição de novas histórias de usuário da aplicação N

Todas as funcionalidades da aplicação seguinte são levantadas e descritas na forma de histórias de usuário, que é uma técnica comum para definir requisitos em métodos ágeis (COHN, 2004). Estas novas funcionalidades podem ser mapeadas para *features* novas (Passo 4) ou podem modificar *features* existentes (Passo 5). As funcionalidades que são iguais às das aplicações anteriores são reutilizadas, não gerando novas histórias pois já foram descritas anteriormente.

4.1.4 Passo 4: Implementação de *feature* novas

As novas funcionalidades da aplicação seguinte que não tenham relação com as *features* anteriores são desenvolvidas normalmente com TDD. Neste caso, são desenvolvidos novos testes unitários e código de produção através da forma *red/green/refactor*. Tanto os testes como o código vão sendo agregados ao repositório de *core assets.*

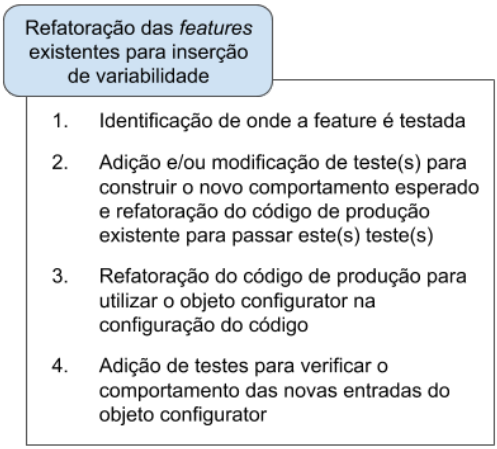
4.1.5 Passo 5: Refatoração das *features* existentes para inserção de variabilidade

Nos casos onde as novas funcionalidades são variações de *features* previamente adicionadas na LPS, são aplicadas modificações

nos testes unitários e no código de produção existente para que seja possível ativar essa nova variabilidade. Os testes existentes são alterados para a inserção dos pontos de variação e serão configurados para que as aplicações seguintes e também as existentes continuem funcionando corretamente. Os novos testes vão conduzindo as alterações no código fonte dos artefatos existentes para a inserção dos pontos de variação que serão posteriormente configurados nas aplicações seguintes.

As *features* da aplicação seguinte que são variações de *features* existentes são definidas conforme a Figura 9 e são detalhados a seguir:

Figura 9 – Detalhamento do passo 5



4.1.5.1 Passo 5.1: Identificação de onde a *feature* é testada

Nesse passo é preciso localizar, dentro das suítes de testes, quais testes estão garantindo que a *feature* da aplicação existente está sendo testada. Dependendo da variabilidade que está sendo inserida, serão necessárias modificações no testes, quando o teste existente falhar para o novo comportamento. Os testes dessa *feature* farão parte da suíte de testes que será executada sempre que a *feature* for configurada para fazer parte de um produto.

4.1.5.2. Passo 5.2: Adição e/ou modificação de novo(s) teste(s) para construir o novo comportamento esperado e refatoração do código de produção existente para passar este(s) teste(s)

Seguindo a prática de TDD, os novos comportamentos serão adicionados por meio da inserção de novos testes unitários e da refatoração do código existente.

4.1.5.3. Passo 5.3: Refatoração do código de produção para utilizar o objeto *configurator* na configuração do código

Para que os pontos de variação sejam ativados em tempo de execução, o código deve estar preparado para ler a configuração e definir o comportamento da aplicação. Isso é feito por meio do objeto *configurator* que irá ler o arquivo de configuração e fornecerá a variante necessária ao ponto de variação em tempo de execução.

4.1.5.4. Passo 5.4: Adição de testes para verificar o comportamento das novas entradas do objeto *configurator*

Assim como as outras classes, a classe que instancia o objeto *configurator* também precisa ser testada. Esses testes servem para validar as entradas possíveis para cada variante, e também servem para testar os arquivos de configuração existentes.

O mecanismo principal para ativar a variante é o arquivo de configuração. Além de centralizar as configurações para a montagem da aplicação, o arquivo de configuração também serve de documentação dos pontos de variação de cada aplicação. O formato do arquivo escolhido foi o *JavaScript Object Notation* (JSON) por ser um arquivo leve, de fácil leitura e que comumente é utilizado para troca de dados.

Cada nova aplicação terá uma evolução do arquivo de configuração (Figura 10), ativando e configurando o que for necessário (exceto a primeira, que foi criada sem a previsão de variabilidade). O arquivo de configuração é criado quando são adicionadas *features* que são variações de *features* que já existiam no repositório de *core assets*. Nesse arquivo, para cada variabilidade, é incluída a chave da variante e o valor da configuração daquela *feature*. Juntamente com esse arquivo é criada uma classe de testes para testar as combinações de chave e valor. Também é criada uma classe no código para ler o arquivo de configuração e instanciar um objeto *configurator*, que será utilizado na aplicação para informar quais variantes serão ativadas. O arquivo continua sendo modificado à medida que novos pontos de variação e variantes vão sendo adicionadas à LPS. A Figura 10 mostra um exemplo

de arquivo de configuração para uma aplicação que inclui as *features* *featureKeyOne* e *featureKeyTwo*.

Figura 10 – *Template* do arquivo de configuração

```
1. {
2.   "featureKeyOne": "someValue",
3.   "featureKeyTwo": 2
4. }
```

Os arquivos JSON não aceitam comentários e, portanto, é necessário ter algum outro arquivo de suporte que documente quais são as variantes possíveis de se inserir para cada ponto de variação. É possível utilizar um outro arquivo JSON para documentar quais são os valores possíveis para uma determinada entrada. A Figura 11 mostra um exemplo onde *featureKeyOne* aceita os valores *someValue* e *otherPossibleValue* e *featureKeyTwo* aceita os valores 1, 2 e 3.

Figura 11 – *Template* de documentação do arquivo de configuração

```
1. {
2.   "featureKeyOne": ["someValue", "otherPossibleValue"],
3.   "featureKeyTwo": [1, 2, 3]
4. }
```

Cada nova aplicação vai agregando funcionalidades para a LPS. Para as demais aplicações geradas após a primeira, funcionalidades novas ou variações vão sendo adicionadas executando os passos 3, 4 e 5. Por meio do TDD é criada uma rede de testes para garantir que as aplicações anteriores continuem funcionando de acordo com seus requisitos. Para auxiliar na criação e execução dessa rede de testes, também é proposto um *framework* de testes que dará esse suporte, e que é descrito a seguir.

4.2 *FRAMEWORK* DE TESTES

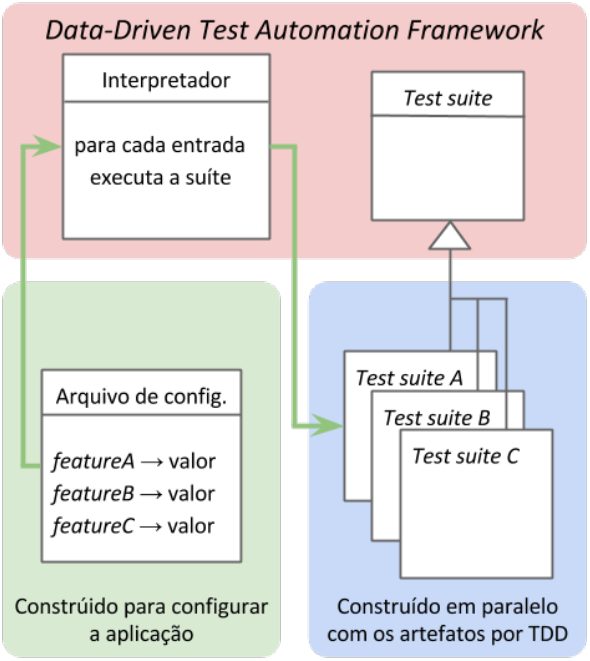
Por ser um desenvolvimento dirigido por testes, os testes unitários vão sendo escritos conforme a aplicação vai sendo desenvolvida. Para cada ponto de variação, um ou mais testes unitários

são necessários para testar todas as variantes da *feature*, que serve como forma de agrupamento dos testes, usando técnicas como *annotation*s, reflexão, inversão de controle, ou qualquer outra técnica que permita a seleção de determinados testes. Além disso, são necessários métodos utilitários de teste (MESZAROS, 2007) para permitir a escolha, configuração e execução de testes para uma determinada configuração de uma *feature*. Assim como uma classe interpretadora (GAMMA et al., 1994), responsável por ler o arquivo de configuração, para executar estes métodos utilitários.

O novo *framework* desenvolvido nesta dissertação é denominado *Data-Driven Test Automation Framework,* ou DDTAF. Uma visão geral do DDTAF é mostrada na Figura 12, onde é possível ver os artefatos arquivo de configuração e suítes de teste, que serão construídos e farão parte do repositório de *core assets*. A Figura 12 também mostra quando os artefatos serão construídos: durante a configuração da aplicação (retângulo verde) e durante o desenvolvimento dos artefatos por meio do TDD (retângulo azul). O *framework* de testes (retângulo vermelho) possui o interpretador e a superclasse de suíte de testes (*Test suite*) que será estendida durante o desenvolvimento de novos artefatos de código. A seta verde simboliza a execução dos testes, onde o interpretador lê as entradas do arquivo de configuração e executa as suítes de testes específicas para cada configuração. O *framework* de testes deve ser configurado em um servidor de integração contínua como o Jenkins[6] ou o Team City[7] para ser executado a cada vez que um novo código é enviado para o sistema de controle de versão.

[6] http://jenkins-ci.org
[7] http://www.jetbrains.com/teamcity

Figura 12 – *Framework* de testes: *Data-Driven Test Automation Framework*



Durante a inclusão de novos artefatos de software no repositório de *core assets*, o DDTAF será utilizado para realizar testes de regressão, pois conforme as alterações de código vão sendo feitas nos *core assets* para acomodar novas funcionalidades, será possível verificar se as aplicações individuais continuam funcionando.

As suítes de testes, utilizadas pelo DDTAF, vão sendo construídas em paralelo com a construção das aplicações seguintes à primeira. Conforme as suítes de testes vão sendo construídas e os pontos de variação vão sendo inseridos na LPS, o arquivo de configuração também vai sendo construído para poder ativar esses pontos. O DDTAF vai utilizar esse mesmo arquivo de configuração para executar as suítes de testes relacionadas com as *features* configuradas.

Dentro do DDTAF, se encontra o interpretador (GAMMA et al., 1994) que é responsável por ler o arquivo de configuração e as variantes incluídas no arquivo e por executar as suítes de testes, relacionadas à uma determinada *feature* variável que foi implementada, passando os parâmetros necessários para cada configuração.

O arquivo de configuração tem uma chave e um valor para a configuração de cada recurso e é criado para descrever onde e como os recursos variáveis serão configurados. Este arquivo também será utilizado pelo *Data-Driven Teste Automation Framework* para executar o conjunto de testes relacionados a cada *feature*. Uma das utilidades da execução dos testes, durante o desenvolvimento de *feature* específicas do aplicativo, é verificar se novos recursos não estão em conflito com o comportamento do resto da aplicação. Os testes também são executados quando a versão final for construída (ou seja, quando todo o aplicativo estiver configurado e desenvolvido) para garantir que tudo está funcionando corretamente, ou seja, cumprindo sua função como teste de regressão.

Cada arquivo de configuração de cada aplicação torna-se parte do repositório de *core assets*, de modo que durante a construção ou modificação de artefatos seja possível executar testes a partir desta configuração de variante e garantir que as mudanças nos artefatos do repositório de *core assets* não introduzam erros que possam afetar a aplicação representada por aquele arquivo de configuração. Dessa forma, os arquivos de configuração podem ser reusados como testes de regressão após o desenvolvimento de novas *features* ou durante a manutenção dos artefatos existentes (por exemplo, durante a refatoração).

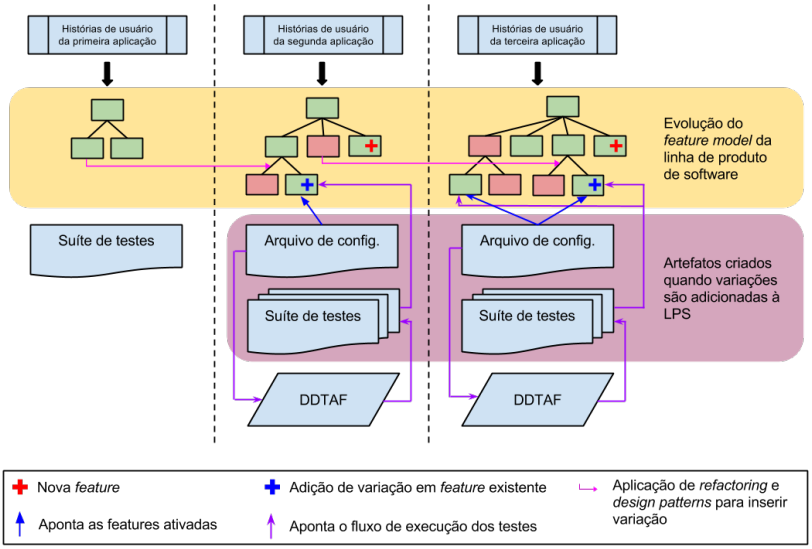
Após o arquivo de configuração ter sido completado, todos os testes podem ser automatizados por meio de uma ferramenta de integração contínua. Estes testes também podem ser executados após mudanças no repositório de artefatos, para cada arquivo de configuração existente, garantindo, desta forma, que as configurações da aplicações da LPS continuam válidas mesmo após modificações no código. Esses testes executados de forma automatizada também servem como testes de regressão.

O DDTAF aplica padrões de testes unitários no contexto de LPS: para facilitar a escrita e execução dos testes principalmente durante a construção da aplicação. Usando os Métodos Utilitários de Teste (Seção – 2.5.5), os testes são configurados por meio de seus parâmetros e, depois, são executados seguindo o padrão de Teste de Quatro Fases (Seção – 2.5.1). O interpretador é responsável por saber qual conjunto de testes deve executar para cada entrada no arquivo de configuração. O interpretador faz parte de uma adaptação do padrão de Testes Guiados por Dados (Seção – 2.5.3), pois ao invés do interpretador ler as entradas e saídas esperadas, ele lê o arquivo para passar os parâmetros dos testes que devem ser executados. Com isso, tem-se o DDTAF como um

Framework de Automação de Testes (Seção – 2.5.2), pois é responsável por executar as suítes de testes e registrar os resultados.

A Figura 13 mostra a evolução de uma LPS, exemplificando a construção de três aplicações. A cada aplicação, novas histórias de usuário vão aparecendo e adicionando *features* à LPS ou adicionando pontos de variação ou variantes às *features* já existentes. Os retângulos verdes no *feature model* indicam as *features* que foram selecionadas para montar a aplicação. Abaixo é possível ver o *framework* de testes e sua utilização ao longo do tempo do desenvolvimento da LPS.

Figura 13 – Evolução de uma LPS



5 EXEMPLO

Para ilustrar a utilização da abordagem proposta e o uso e operação do *framework* de testes, foi escolhida a aplicação desenvolvida no livro "*Test-Driven iOS Development*" (LEE, 2012) como a primeira aplicação de uma LPS de abordagem reativa. A escolha de uma aplicação já existente, que foi desenvolvida dirigida por testes, foi feita para reduzir o viés de escrever a primeira aplicação. Além disso, esta aplicação não foi desenvolvida seguindo a abordagem de LPS, o que é importante para o desenvolvimento reativo de uma LPS seguindo a presente proposta. Esta primeira aplicação serve como uma base para o desenvolvimento das aplicações seguintes, onde novas características e variabilidades são inseridas. Os testes da primeira aplicação são refatorados para que os *core assets* continuem a suportar múltiplas aplicações.

A seguir, é inicialmente descrita a primeira aplicação. Em seguida, descreve-se a segunda, terceira e quarta aplicação e as modificações que foram feitas nos *core assets*.

5.1 PRIMEIRA APLICAÇÃO

A primeira aplicação é para o sistema operacional iOS que disponibiliza para o usuário as questões mais recentes do fórum online Stack Overflow[8] sobre tópicos relacionados com o desenvolvimento para dispositivos móveis com iOS. As questões são organizadas por tópicos e os usuários também conseguem visualizar as respostas que foram dadas a essas questões. O nome dessa aplicação é BrowserOverflow.

Descrevendo mais detalhadamente essa visão geral, tem-se as seguintes histórias de usuário da aplicação:

- **Lista de tópicos:** A aplicação inicia mostrando uma lista de tópicos relacionados com iOS, como mostrado na Figura 14: iPhone, Cocoa Touch, UiKit, Objective-C e Xcode. Cada tópico está associado à uma categoria do Stack Overflow.

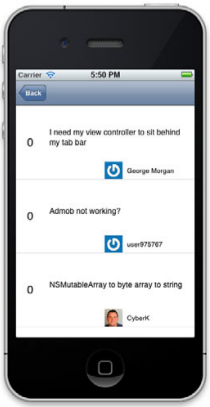
[8] Disponível em http://stackoverflow.com

Figura 14 – Tela inicial da aplicação com tópicos para serem escolhidos

Fonte: Lee (2012).

- **Acessar as questões mais recentes:** Ao selecionar um tópico, uma outra lista aparece mostrando as 20 questões mais recentes e que possuem a categoria associada ao tópico. Essa lista é apresentada em ordem cronológica, da mais recente para a menos recente. Como é possível ver na Figura 15, cada célula da lista mostra o título da questão, o usuário que fez a questão (nome e imagem) e a pontuação da questão (quantidade de pessoas que votaram a favor ou contra essa questão).

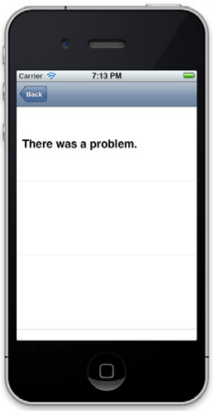
Figura 15 – Tela com a lista de questões relacionadas a algum tópico



Fonte: Lee (2012).

- **Disponibilidade de conexão com a internet:** Para carregar a lista de questões mais recentes e as respostas é necessário que exista conexão com a internet, porém também deve ser previsto que a conexão com o site stackoverflow.com também falhe. O aplicativo deve, então, informar o usuário caso algo impossibilite que as informações sejam carregadas, seja por falta de conexão com a internet ou por falha de comunicação com o site, como mostrado na Figura 16.

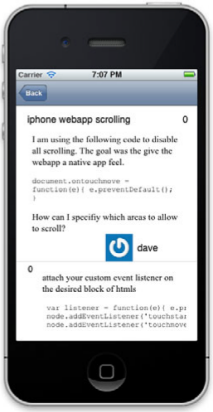
Figura 16 – Tela com mensagem de erro caso não seja possível conectar com a internet



Fonte: Lee (2012).

- **Acessar as respostas de uma questão:** Ao selecionar uma questão, uma outra tela é aberta. Nessa tela, mostrada na Figura 17, aparece a descrição da questão (título e explicação) e abaixo da descrição aparece uma lista de respostas. Se houver uma resposta escolhida como a correta, ela aparece primeiro com uma marcação indicando que foi escolhida pela autor. Se não houver resposta escolhida, então as respostas são listadas conforme a pontuação que possuem. Para cada resposta, é mostrado o nome e a imagem do usuário que deu a resposta.

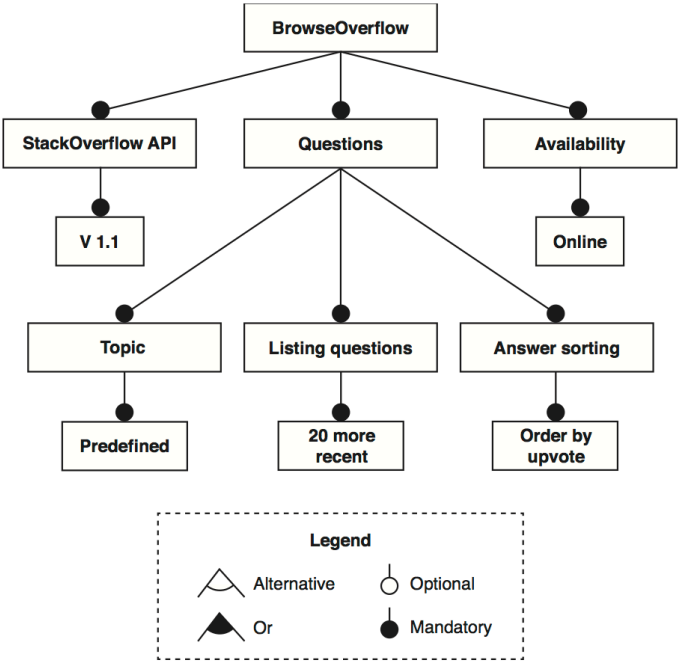
Figura 17 – Tela mostrando o detalhamento de uma questão com as respostas abaixo



Fonte: Lee (2012).

As histórias de usuário foram, então, transcritas para um *feature model* para mostrar quais *features* foram desenvolvidas e para posteriormente comparar com o *feature model* evoluído após a construção das aplicações seguintes. O *feature model* é mostrado na Figura 18. Como existe, inicialmente, somente uma aplicação, todas as *features* desta primeira aplicação são consideradas obrigatórias.

Figura 18 – *Feature model* da primeira aplicação



Além do desenvolvimento da aplicação ter sido dirigido por testes, ele foi feito incrementalmente de acordo com cada camada. Primeiro foi desenvolvida a camada de modelo, em seguida a camada de *controller* e, finalmente, foi feita a integração entre as duas camadas. Essa primeira aplicação possui 24 classes e 182 testes unitários para testar e satisfazer as *features* propostas. O código fonte da aplicação é aberto[9].

O desenvolvimento da primeira aplicação dirigido por testes e sem a pretensão de prever demandas futuras ou variabilidades equivale ao primeiro passo da abordagem proposta (Seção 4.1.1). As 24 classes, os 182 testes unitários e o *feature model* se tornam parte do repositório de *core assets*, o que equivale ao segundo passo (Seção 4.1.2).

Para exemplificar a modificação de *features* da LPS e também dos testes existentes das aplicações anteriores, assim como o uso do

[9] Disponível em https://github.com/iamleeg/BrowseOverflow

framework de testes DDTAF são desenvolvidas outras três aplicações descritas a seguir.

5.2 SEGUNDA APLICAÇÃO

Seguindo o terceiro passo (Seção 4.1.3) as *features* da segunda aplicação serão descritas como histórias de usuários. As seguintes histórias de usuários são variações das histórias da primeira aplicação descritas anteriormente:

- **Acessar as 30 perguntas mais recentes:** Ao selecionar um tópico, a lista de questões que aparece deve mostrar as 30 questões mais recentes, ao invés de 20 como era na primeira aplicação.
- **Usar a *Application Programing Interface* (API) 2.0:** A versão da API utilizada para requisitar as questões e respostas deve ser a 2.0, ao invés da 1.1 utilizada na primeira aplicação. Essa API é responsável por entregar os dados do site Stack Overflow para a aplicação em formato JSON.
- **Acessar o conteúdo sem conexão com a internet:** O conteúdo (questões e respostas) que já foi visualizado deve estar disponível para o usuário caso não haja conexão com a internet.

Nessa segunda aplicação não foram adicionadas novas *features*. Novas *features* teriam sido desenvolvidas normalmente por meio de TDD, de acordo com o quarto passo (Seção 4.1.4).

5.2.1 História do Usuário 1: Acessar as 30 Perguntas Mais Recentes

A seguir, os passos necessários para a inserção da nova história de usuário que modifica a quantidade de questões de 20 para 30 são mostrados.

Primeiramente é necessário verificar onde essa *feature* é testada procurando entre os testes unitários da primeira aplicação, seguindo o passo 5.1 (Seção 4.1.5.1). No modelo proposto para a primeira aplicação, cada tópico continha entre 0 e 20 questões. O diagrama de classes correspondente a essa história de usuário é mostrado na Figura 19 a seguir.

Figura 19 – Relação entre Tópicos e Questões

Fonte: Adaptado de Lee (2012).

Na primeira aplicação, na classe responsável por testar os tópicos (*TopicTests*) existe o método *testLimitOfTwentyQuestions* (Figura 20). Esse foi o teste encontrado por ser responsável por garantir o comportamento esperado para a quantidade de questões mostradas por tópico no aplicativo, e portanto é o teste que precisa ser modificado para aceitar a variação do comportamento na aplicação seguinte.

Figura 20 – Método para testar o limite de 20 questões da primeira aplicação

```
1. - (void)testLimitOfTwentyQuestions
2. {
3.     Question *q1 = [[Question alloc] init];
4.     for (NSInteger i = 0; i < 25; i++) {
5.             [topic addQuestion: q1];
6.     }
7.     XCTAssertTrue(
8.         [[topic recentQuestions] count] < 21);
9. }
```

Como na primeira aplicação tinha-se o limite de 20 questões e na segunda aplicação deseja-se o limite de 30 questões, pode-se generalizar essa *feature* como sendo um limite de *n* questões, sendo esse *n* definido no arquivo de configuração pelo responsável em montar a aplicação.

Assim, um novo método, chamado *testLimitOfQuestions* (Figura 21), é criado para testar o limite de questões que deve ser definido em tempo de compilação. Esse teste também recebe o parâmetro *limitOfQuestions* que é a forma de associar o teste com a *feature* para ser executado pelo DDTAF.

Figura 21 - Método adicionado para testar o nome limite de questões, com parâmetro para ser executado pelo *framework* de testes

```
1. - (void)testLimitOfQuestions:(int)limitOfQuestions
2. {
3.   Question *q1 = [[Question alloc] init];
4.   for (NSInteger i = 0; i < topic.limit; i++) {
5.           [topic addQuestion: q1];
6.   }
7.   XCTAssertTrue(
8.    [[topic recentQuestions] count] < topic.limit);
9. }
```

Para tanto, é criado um novo campo *limit* na classe *Topic* para armazenar esse valor, mostrado na linha 1 da Figura 22. Entretanto, essas mudanças acarretam mudanças em outros lugares. O método *addQuestions*, que antes armazenava o número 20 diretamente no código, agora utiliza o atributo *limit*, configurado na linha 9.

Figura 22 - Adicionado um novo campo à classe *Topic* e um novo argumento ao construtor

```
1.  @synthesize limit;
2.
3.  - (id)initWithName:(NSString *)newName
4.                 tag:(NSString *)newTag
5.  {
6.      if ((self = [super init])) {
7.          name = [newName copy];
8.          tag = [newTag copy];
9.          limit = [Configurator readEntry:@"limitOfQuestions"];
10.         questions = [[NSArray alloc] init];
11.     }
12.     return self;
13. }
```

Para que o teste *testLimitOfQuestions:limitOfQuestions* seja executado pelo DDTAF é necessário que seja criado o arquivo de configuração. Para cada ponto de variação da LPS existe uma chave descritiva no arquivo, e essa chave referencia um valor que é a definição da configuração necessária para ativação desse ponto. O arquivo JSON para configurar o número de questões é mostrado na Figura 23.

Figura 23 – Arquivo de configuração JSON e do *feature model* para a *feature* de limite de questões

```
1.  {
2.    "limitOfQuestions": 30
3.  }
```

Esse arquivo de configuração é, então, lido pelo interpretador do DDTAF responsável pela execução de testes (Figura 24). Então, para cada entrada do arquivo de configuração uma suíte de testes é executada, correspondente à execução dos testes para cada *feature*. O valor de cada entrada, passado por parâmetro, é usado para configurar a suíte de testes.

Figura 24 – Método principal de classe *Interpreter* que faz parte do DDTAF

```
1.  - (void)executeTests {
2.      configFile = [self readConfigurationFile];
3.      for (Entry *entry in configFile.entries){
4.          [self runSuiteOfTestsForEntry:entry];
5.      }
6.  }
```

A Figura 25 mostra como são executados os testes dessa *feature*, depois do interpretador do *framework* de testes ler a entrada do arquivo de configuração JSON. Neste caso mais simples o valor real usado no arquivo serve como parâmetro de entrada para o caso de teste unitário. Este teste exercita a *feature* de listagem de questões, onde o limite de perguntas mostradas é definido para cada produto.

Figura 25 – Método responsável por executar todos os testes relacionados com a *feature* de limite de questões, faz parte do DDTAF

```
1.  - (void)runLimitOfQuestionsTests{
2.      int limit =
3.          [configurationFile limitOfQuestions];
4.      [self testLimitOfQuestions:limit];
5.  }
```

Após todos os testes terem passado, verificou-se que não era mais necessário o teste chamado *testLimitOfTwentyQuestions*, que foi, então, removido. A adição do novo teste e a alteração do código seguem o passo 5.2 (Seção 4.1.5.2).

Com essas alterações foi necessário adicionar algum mecanismo capaz de ativar a variação. O mecanismo utilizado para fazer essa configuração em tempo de execução é uma classe que gera um objeto *configurator* para acessar o arquivo de configuração e fornecer a configuração no ponto necessário.

Então foi necessário refatorar o código do construtor da classe *Topic* para utilizar o objeto *configurator* na configuração do código, mostrado na linha 9 da Figura 26, o que corresponde ao passo 5.3 (Seção 4.1.5.3).

Figura 26 - Mudanças no método de adicionar questões antes (acima) e depois (abaixo), destacados em azul.

```
1.  - (void)addQuestion:(Question *)question
2.  {
3.      NSArray *newQuestions =
4.         [questions arrayByAddingObject: question];
5.      if ([newQuestions count] > 20) {
6.          newQuestions =
7.              [self sortQuestionsLatestFirst:
8.                  newQuestions];
9.          newQuestions =
10.             [newQuestions subarrayWithRange:
11.                 (NSMakeRange(0, 20)];
12.     }
13.     questions = newQuestions;
14. }
```

```
1.  - (void)addQuestion:(Question *)question
2.  {
3.      NSArray *newQuestions =
4.         [questions arrayByAddingObject: question];
5.      if ([newQuestions count] > limit) {
6.          newQuestions =
7.              [self sortQuestionsLatestFirst:
8.                  newQuestions];
9.          newQuestions =
10.             [newQuestions subarrayWithRange:
11.                 (NSMakeRange(0, limit)];
12.     }
13.     questions = newQuestions;
14. }
```

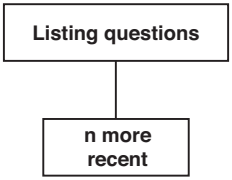
Como esta foi a primeira *feature* modificada, nesse momento foi necessário criar mais uma classe de teste (*ConfiguratorTests*) responsável por testar a classe que fará a leitura do arquivo de configuração e instanciação do objeto *configurator*. O objeto *configurator* é acessado pelo sistema para ativar as variantes. A Figura 27 mostra três métodos, da classe *ConfiguratorTests*, utilizados para testar a entrada "*limitOfQuestions*", necessária para cumprir o último passo 5.4 (Seção 4.1.5.4). As informações necessárias para instanciar o objeto *configurator* vem de um arquivo JSON que é criado para cada aplicação.

Figura 27 – Três métodos da classe *ConfiguratorTests* que testam a entrada *limitOfQuestions*

```
1.  - (void)testLimitOfQuestionsShouldBeAnInteger {
2.      int limitOfQuestions = [Configurator readEntry:@"limitOfQuestions"];
3.      NSScanner *scanner = [NSScanner scannerWithString:
          [NSString stringWithFormat:@"%d", limitOfQuestions]];
4.      XCTAssertTrue([scanner scanInt:&limitOfQuestions]);
5.  }
6.
7.  - (void)testLimitOfQuestionsShouldBeGreaterThanZero {
8.      int limitOfQuestions = [Configurator readEntry:@"limitOfQuestions"];
9.      XCTAssertTrue(limitOfQuestions > 0);
10. }
11.
12. - (void)testLimitOfQuestionsShouldBeLessThanOneHundred {
13.     int limitOfQuestions = [Configurator readEntry:@"limitOfQuestions"];
14.     XCTAssertTrue(limitOfQuestions < 100);
15. }
```

Com essas modificações é possível ver a alteração feita na *feature* da listagem de questões que foi generalizada para listar as n questões mais recentes ao invés das 20 como era na primeira aplicação. É possível visualizar essa modificação no trecho do *feature model* da Figura 28.

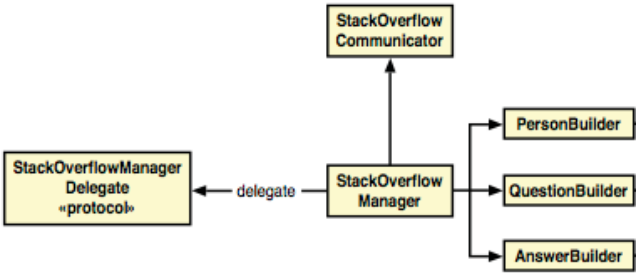
Figura 28 – *Feature model* para a *feature* de limite de questões



5.2.2 História do Usuário 2: Usar a API 2.0

No caso da variação da API, temos um cenário mais complexo. Na primeira aplicação, uma fachada chamada *StackOverflowManager* é responsável por se comunicar com a classe *StackOverflowCommunicator*, de acordo com o diagrama de classes apresentado na Figura 29. As requisições para a API V1.1 do StackOverflow estão centralizadas nessa classe.

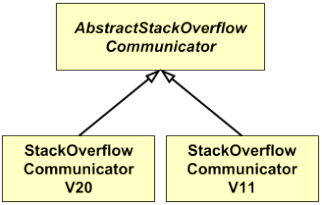
Figura 29 - Relação entre a fachada e as outras classes



Fonte: Lee (2012).

Assim, a classe que testa o *StackOverflowCommunicator* tem que ser generalizada para que teste o *StackOverflowCommunicator* independente de qual versão da API está sendo testada. Dessa forma, os testes guiam a construção de um adaptador (GAMMA et al., 1994) chamado *AbstractStackOverflowCommunicator*. Essa técnica de extrair o adaptador durante a refatoração foi proposta em (KERIEVSKY, 2004). Outras classes de testes são usadas para a criação das classes específicas com os detalhes de cada API, *StackOverflowComunicatorV11* e *StackOverflowComunicatorV20* (Figura 30).

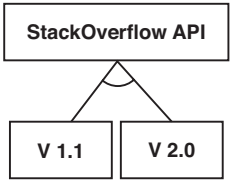
Figura 30 - Novos *adapters* da *StackOverflowCommuicator*



Após essas modificações o arquivo de configuração evolui e ganha mais uma entrada possível. A nova entrada e sua configuração são mostradas na Figura 31 e abaixo está a parte do *feature model* com as alternativas dessa *feature*.

Figura 31 – Evolução do arquivo de configuração e do *feature model* para a *feature* de API do Stack Overflow

```
1.  {
2.    "limitOfQuestions": 30,
3.    "stackOverflowApiVersion": 2.0
4.  }
```



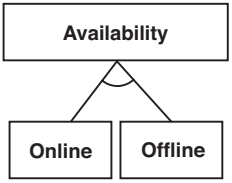
5.2.3 História do Usuário 3: Acessar o Conteúdo Sem Conexão com a Internet

A variação da persistência também acarreta em mudanças em diversos locais. A primeira aplicação tem vários mecanismos para notificar o usuário caso a conexão com o servidor falhe e caso não exista conexão. Alguns desse mecanismos vão necessitar de alterações para acomodar a nova *feature* que permite que os dados sejam salvos no dispositivo e acessados posteriormente quando não for possível carregar dados do servidor devido à falta de conexão.

Como visto anteriormente, a classe *StackOverflowManager* atua como fachada (GAMMA et al., 1994) entre *StackOverflowCommunicator* e as classes de *builder*: (GAMMA et al., 1994) *PersonBuilder*, *QuestionBuilder* e *AnswerBuilder* (Figura 29). A fachada pede para a classe *StackOverflowCommunicator* fazer requisições para obter os JSONs de resposta do Stack Overflow. Quando um pedido é bem sucedido, o *StackOverflowManager* passa o JSON com a resposta para as classes de *builder*. E então os JSON's devem ser persistidos neste processo também. Quando as requisições falharem, o *StackOverflowManager* deve verificar se tem um JSON anterior, contendo este mesmo pedido, salvo na memória. Após aplicar estas mudanças no código temos o arquivo de configuração para a segunda aplicação conforme mostrado na Figura 32.

Figura 32 - Arquivo de configuração da segunda aplicação e o *feature model* para a *feature* de disponibilidade de acesso a internet

```
{
  "limitOfQuestions": 30,
  "stackOverflowApiVersion": 2.0,
  "contentAvailability": "offline"
}
```



As variantes possíveis de serem configuradas são documentados também usando a notação JSON, conforme a Figura 33. Porém, outras formas de documentação poderiam ser utilizadas como Wikis ou arquivos de texto.

Figura 33 – Documentação das possíveis configurações

```
{
  "limitOfQuestions": [0-100],
  "stackOverflowApiVersion": [1.1, 2.0],
  "contentAvailability": ["online", "offline"],
}
```

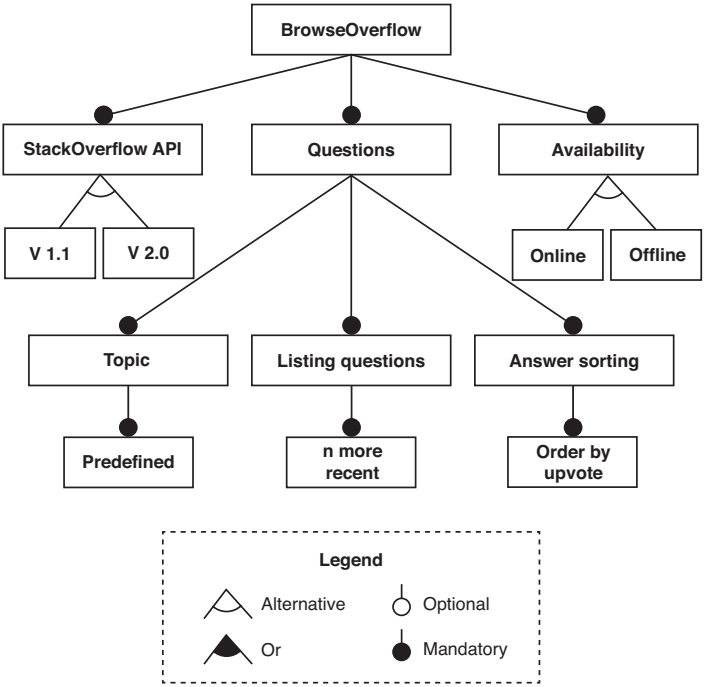
É necessário criar um arquivo de configuração (Figura 34) para a primeira aplicação para que seja possível executar os testes que exercitam suas funcionalidades.

Figura 34 – Arquivo de configuração da primeira aplicação

```
{
  "limitOfQuestions": 20,
  "stackOverflowApiVersion": 1.1,
  "contentAvailability": "online"
}
```

O *feature model* atualizado da LPS pode ser visto na figura 35.

Figura 35 - *Feature model* atualizado após o desenvolvimento da aplicação 2



5.3 TERCEIRA APLICAÇÃO

Assim como na segunda aplicação, e qualquer aplicação subsequente à primeira, o seu desenvolvimento é iniciado pelo terceiro passo da abordagem (Seção 4.1.3), definindo as novas histórias de usuário da aplicação. Na terceira aplicação mais duas *features* foram adicionadas, a primeira é uma *feature* nova e a segunda é uma modificação de uma *feature* existente. Elas são descritas a seguir:

- **Inserir novos tópicos:** Na tela de tópicos o usuário deve poder inserir novos tópicos para procurar questões.
- **Ordenar as respostas cronologicamente:** A lista de respostas de uma questão deve ser ordenada cronologicamente, mostrando as respostas mais recentes primeiro.

Além dessas novas *features* a aplicação será configurada com as seguintes configurações (1) o número máximo de questões recentes será de 25, (2) a versão 2.0 da API será utilizada, e (3) fornecerá conteúdo offline.

5.3.1 História de Usuário 1: Inserir novos tópicos

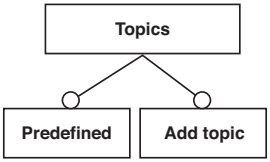
Como a *feature* de inserir novos tópicos é uma nova *feature*, o desenvolvimento começa normalmente com a prática de TDD. Novos testes unitários são escritos para guiar o desenvolvimento de novos comportamentos no código de produção.

A nova funcionalidade adicionada na aplicação 3 não fez reuso de nenhuma classe de produção e nenhuma classe de teste, resultando na adição de uma nova classe de produção e uma nova classe de testes.

Como essa *feature* é opcional para as aplicações anteriores, também é adicionada uma entrada no arquivo de configuração e o *feature model* também evolui, como é possível verificar na Figura 36.

Figura 36 - Arquivo de configuração e o *feature model* evoluídos com a nova *feature*

```
1.  {
2.    "limitOfQuestions": 25,
3.    "stackOverflowApiVersion": 2.0,
4.    "contentAvailability": "online",
5.    "isPossibleToAddTopic": true
6.  }
```



5.3.2 História de Usuário 2: Ordenar as respostas cronologicamente

Para adicionar a variação na ordenação das respostas é necessário encontrar nos testes onde esse comportamento está sendo validado. A classe responsável por testar comportamentos relacionados com respostas é chamada *AnswerTests*, nessa classe o método de teste chamado *testLowerScoringAnswerComesAfterHigher* é responsável por

verificar se duas respostas entre si ficam ordenadas corretamente do ponto de vista da sua pontuação, como mostrado na Figura 37.

Figura 37 – Teste responsável pela a ordenação das respostas

```
1. - (void)testLowerScoringAnswerComesAfterHigher {
2.     otherAnswer.score = answer.score + 10;
3.     XCTAssertEqual([answer compare: otherAnswer],
               NSOrderedDescending, @"Higher score comes first");
4.     XCTAssertEqual([otherAnswer compare: answer],
               NSOrderedAscending, @"Lower score comes second");
5. }
```

Esse teste é, então, refatorado e mostrado na Figura 38. A lógica comum é extraída para um Método Utilitário de Teste, que também é um Teste Parametrizado. O teste invoca o método utilitário passando por parâmetro a configuração escolhida para a *feature*. O método de teste também recebe um parâmetro, para que o DDTAF possa executá-lo.

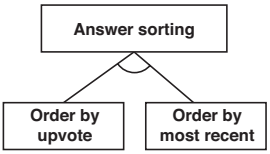
Figura 38 – Alteração no teste de ordenação e teste parametrizado

```
1.  - (void)testLowerScoringAnswerComesAfterHigher:(NSString *) answerSorting {
2.      [self configureAnswersForTestSorting:answerSorting];
3.      XCTAssertEqual([answer compare: otherAnswer], NSOrderedDescending, @"otherAnswer comes first");
4.      XCTAssertEqual([otherAnswer compare: answer], NSOrderedAscending, @"answer comes second");
5.  }
6.
7.  - (void) configureAnswersForTestSorting:(NSString *) answerSorting {
8.      if ([answerSorting isEqualToString:@"upvote"])
9.          otherAnswer.score = answer.score + 10;
10.     if ([answerSorting isEqualToString:@"date"]) {
11.         otherAnswer.date = [NSDate distantFuture];
12.         answer.date = [NSDate distantPast];
13.     }
14. }
```

Após essas modificações o arquivo de configuração atualizado e o *feature model* ficam de acordo com a Figura 39.

Figura 39 - Arquivo de configuração da terceira aplicação e o *feature model* para a *feature* de ordenação das respostas

```
{
  "limitOfQuestions": 25,
  "stackOverflowApiVersion": 2.0,
  "contentAvailability": "online",
  "isPossibleToAddTopic": true,
  "answerSorting": "mostRecent"
}
```



A documentação do arquivo de configuração também é atualizado (Figura 40) mostrando quais são as possíveis variantes para os pontos de variação.

Figura 40 – Documentação das possíveis configurações atualizado

```
{
  "limitOfQuestions": [0-100],
  "stackOverflowApiVersion": [1.1, 2.0],
  "contentAvailability": ["online", "offline"],
  "isPossibleToAddTopic": [true, false],
  "answerSorting": ["upvote", "mostRecent"]
}
```

Ao final do desenvolvimento da terceira aplicação os arquivos de configuração das duas primeiras aplicações precisam ser atualizados e ficam conforme a Figura 41, com o arquivo de configuração da primeira aplicação acima e da segunda aplicação abaixo.

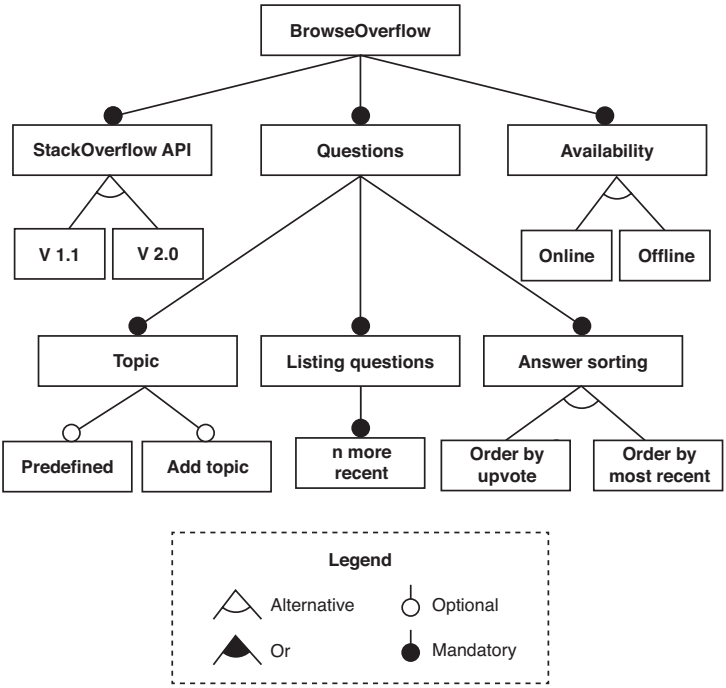
Figura 41 - Arquivo de configuração da primeira aplicação (acima) e segunda aplicação (abaixo)

```
1.  {
2.    "limitOfQuestions": 20,
3.    "stackOverflowApiVersion": 1.1,
4.    "contentAvailability": "online",
5.    "isPossibleToAddTopic": false,
6.    "answerSorting": "upvote"
7.  }
```

```
1.  {
2.    "limitOfQuestions": 30,
3.    "stackOverflowApiVersion": 2.0
4.    "contentAvailability": "offline"
5.    "isPossibleToAddTopic": false,
6.    "answerSorting": "upvote"
7.  }
```

O *feature model* atualizado da LPS pode ser visto na Figura 42.

Figura 42 - *Feature model* atualizado após o desenvolvimento da aplicação 3



5.4 QUARTA APLICAÇÃO

A quarta aplicação é construída utilizando *features* já existentes na LPS. Ela foi construída para ilustrar como variantes diferentes podem ser ativadas para construir aplicações diferentes. E como neste caso não existe a necessidade de escrever nenhuma *feature* nova, nenhum dos passos da abordagem precisa ser executado.

A quarta aplicação possui a seguinte configuração (Figura 43): (1) limita em 40 a quantidade das mais recentes questões que aparecem na lista depois de um tópico ser selecionado, (2) usa a versão 1.1 da API do Stack Overflow, (3) fornece apenas o conteúdo online, (4) não permite ao usuário adicionar novos tópicos, e (5) ordena as respostas por votos positivos.

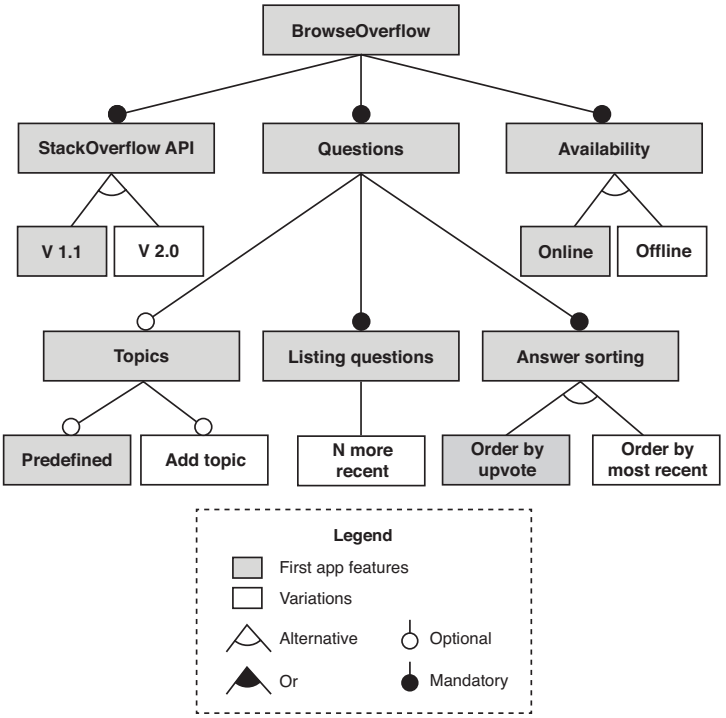
Figura 43 - Arquivo de configuração da quarta aplicação

```
{
  "limitOfQuestions": 40,
  "stackOverflowApiVersion": 1.1,
  "contentAvailability": "online",
  "isPossibleToAddTopic": false,
  "answerSorting": "upvote"
}
```

Os testes podem ser executados de forma automatizada por um servidor de integração contínua que execute o framework de automação de testes. Assim, durante o processo de melhoria ou modificação dos artefatos do repositório de *core assets*, é fácil de verificar se a primeira, segunda, terceira e quarta aplicação (neste exemplo), já configuradas, continuam funcionando.

Ao final do desenvolvimento das quatro aplicações o *feature model* da LPS evolui e agora possui *features* que são obrigatórias, opcionais, alternativas e/ou excludentes, como mostrado na Figura 44. Para o caso da *feature* de quantidade de questões (*limitOfQuestions*) agora temos uma *feature* mais genérica. As *features* "StackOverflow API" (*stackOverflowApiVersion*), "disponibilidade de internet" (*contentAvailability*) e "ordenação de respostas" (*answerSorting*) tem duas alternativas cada uma, e apenas uma poderá ser selecionada. Eles diferem de "tópicos" (*isPossibleToAddTopic*), onde as variações podem ser utilizadas juntamente. A *feature* "tópicos" é opcional porque não é necessária para a construção de uma aplicação onde o usuário precise filtrar listas de perguntas. Todas as outras *features* são necessárias para construir uma aplicação individual.

Figura 44 - *Feature model* após o desenvolvimento das aplicações



O código da LPS está disponível através do link: https://github.com/GlaucoNeves/BrowseOverflowProductLine

5.5 RESULTADOS

Após o desenvolvimento das quatro aplicações foi possível observar que a abordagem proposta contribuiu para a construção de uma LPS a partir de uma aplicação inicial que foi desenvolvida sem esse propósito. A maior parte do código de produção e testes unitários da primeira aplicação são reusados nas aplicações seguintes, conforme é possível ver no Quadro 5 a seguir.

Quadro 5 – Testes unitários das aplicações

Aplicação	Testes reusados	Testes modificados	Testes adicionados	Total de testes
1	-	-	182	***182***
2	167	15	5	***187***
3	185	2	8	***195***
4	195	-	-	***195***

Na segunda e na terceira aplicação, a prática de TDD ajudou a guiar as modificações nos testes unitários e no código quando houve a necessidade de modificar o comportamento existente para adicionar pontos de variação na LPS.

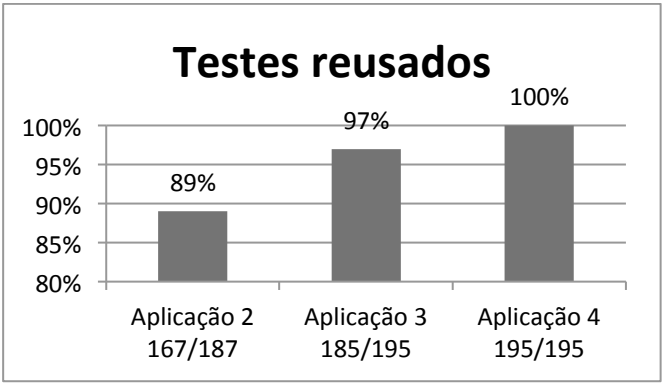
A aplicação dois incluiu 2 *features* existentes na primeira aplicação e modificou outras 3 *features*, e durante o seu desenvolvimento 15 testes precisaram ser modificados e 5 novos testes foram adicionados.

Na terceira aplicação, além das features existentes nas aplicações um e dois, uma nova *feature* foi adicionada também. Novas *features* trazem novos testes e novo código de produção que são agregados ao repositório de *core assets*. Assim, 8 novos testes unitários foram adicionados ao repositório. No caso de novas *features*, os testes existentes não são reusados, no entanto, existindo alguma lógica comum, esta pode ser reusada por testes diferentes, como por exemplo no método *setUp* das classes de testes. E no caso da terceira aplicação, 2 testes foram modificados para a adição de um novo ponto de variação na LPS.

Em relação à quarta aplicação, esta foi configurada somente com opções existentes da LPS. Assim, o código de produção e testes unitários foram reusados a partir das aplicações passadas, enquanto o DDTAF ficou responsável pela execução dos testes a partir do arquivo de configuração da quarta aplicação para garantir que a aplicação estivesse funcionando.

Conforme as aplicações futuras vão explorando os pontos de variação criados ao invés de adicionar novas *features* ou variações, é possível observar o alto reuso dos testes, que pode chegar a 100% como no caso da aplicação 4, mostrado na Figura 45.

Figura 45 – Gráfico do reuso de testes



O gráfico da Figura 45 mostra que na aplicação 2 foram reusados 89% dos testes da primeira aplicação, enquanto os outros 11% de testes (15 testes) precisaram ser modificados para adição dos pontos de variação. Já na aplicação 3, 97% dos testes foram reusados, pois apenas 2 testes foram modificados para adicionar um novo ponto de variação. No caso da aplicação 4, 100% dos testes foram reusados, pois a aplicação foi construída apenas configurando pontos de variação existentes.

Assim como a quarta aplicação, diversas outras aplicações são possíveis de serem geradas combinando diferentes variações que a LPS possui.

Outro artefato que também pode ser reusado é o arquivo de configuração de cada aplicação. Cada arquivo de configuração vira parte do repositório de *core assets* e pode ser executado como parte de uma estratégia de testes de regressão, verificando que as configurações daquelas aplicações continuam válidas, mesmo após modificações no código de produção.

Ao final da quarta aplicação foram alteradas diversas partes do código de produção e testes. Ao todo foram modificados 57 arquivos do projetos, com adição de 2283 linhas nos arquivos e remoção de 249 linhas, incluindo classes de projeto e outros arquivos auxiliares, como arquivos de projeto e os arquivos de configuração.

Foram modificadas 8 classes e foram adicionadas outras 3 classes de código de produção para dar suporte à variabilidade da LPS. Também foi adicionada a classe do *framework* de testes (*MyTestCase*), que causou a alteração de todas as classes de testes que foram modificadas

para estender a classe de testes do *Data Driven Test Automation Framework*.

6 COMPARATIVO COM TRABALHOS RELACIONADOS

A seguir é apresentada uma comparação entre a proposta apresentada nesta dissertação e os trabalhos identificados no estado da arte por meio da revisão sistemática.

6.1 DO PONTO DE VISTA DA ABORDAGEM PARA A CONSTRUÇÃO REATIVA DE LPS

Ghanam e Maurer (2009, 2010) propuseram a utilização de uma abordagem *test-driven* para introduzir variabilidade refatorando o código existente. Dessa forma os testes podem guiar a inserção de novas formas de variabilidade do sistema quando houver demanda. Eles propõem a utilização do padrão de projeto *factory* (GAMMA et al., 1994) no processo de refatoração para que novas alternativas de uma *feature* sejam geradas e o padrão de projeto *decorator* (GAMMA et al., 1994) para geração de opções. As variabilidades são, então, configuradas em uma classe específica do código.

A solução apresentada por Ghanam e Maurer (2009, 2010) requer que as configurações fiquem em uma classe do projeto, enquanto o arquivo de configuração dessa dissertação foi projetado para ficar separado do código. Ter o arquivo de configuração separado do código facilita a inclusão de novos pontos de variação e seleção de variantes, tornando essa solução mais simples do que a apresentada por Ghanam e Maurer (2009, 2010).

Em (KAKARONTZAS; STAMELOS; KATSAROS, 2008) é proposta uma abordagem sistemática para a criação de novos componentes de software variáveis por meio da customização dos *core assets* existentes em uma LPS. A proposta de Kakarontzas, Stamelos e Katsaros (2008) considera dois tipos de variações, as de qualidade e as funcionais. Eles utilizam o TDD para dar assistência na evolução dos componentes de software de uma LPS. Essa abordagem sugere que seja criado um novo componente, que é uma extensão de um componente puro, quando novas funcionalidades forem adicionadas. Assim, quando não for necessária uma nova funcionalidade, basta utilizar o componente puro ou de uma hierarquia superior.

A hierarquia de componentes apresentada em (KAKARONTZAS; STAMELOS; KATSAROS, 2008) pode aumentar consideravelmente a complexidade da LPS. Com uma hierarquia grande, pode ser difícil selecionar alternativas e opções de requisitos para compor um novo produto. O arquivo de configuração proposto nesta

dissertação parece ser, então, uma solução mais simples do que a apresentada em (KAKARONTZAS; STAMELOS; KATSAROS, 2008) para centralizar a escolha das variabilidades, além de servir de guia para a montagem de uma nova aplicação.

6.2 DO PONTO DE VISTA DO FRAMEWORK DE TESTES

Kastner et al. 2012 propõem uma forma de executar os testes unitários sem a necessidade de gerar as aplicações individuais. A abordagem de Kaster et al. 2012 utiliza um interpretador ciente de variabilidade e exercita uma única vez os caminhos que não são afetados pela variabilidade. A variabilidade é recodificada para simular os casos de testes com um verificador de modelos. O verificador de modelos é responsável por testar todos os caminhos possíveis de serem executados pelo programa.

A proposta de Kaster et al. 2012 diferencia-se da proposta desta dissertação pois visa a aplicação dos testes sem a geração das aplicações individuais, enquanto nesta dissertação é proposto a utilização dos testes para conduzir a criação das novas aplicações. Os arquivos de configuração desta dissertação também servem para que o interpretador do framework de testes consiga executar testes de regressão, verificando que o arquivo continua sendo válido para gerar as aplicações anteriores.

Ganesan et al., (2010, 2013) apresentam uma análise da abordagem de testes unitários em LPS usado por uma equipe na NASA. Seu objetivo foi compreender, analisar e recomendar estratégias para melhorar a infraestrutura de testes unitários existente, bem como para capturar as lições aprendidas e as melhores práticas que podem ser usadas em testes unitários no contexto de LPSs em geral. Ganesan et al., (2010, 2013) propõem a utilização de um *stub framework* de modo que os módulos possam ser testados sem as outras dependências.

Tudo que é proposto por Ganesan et al. (2010, 2013) pode ser aplicado em conjunto com a proposta da presente dissertação. A principal diferença entre eles é o problema abordado. Ambos se concentram em testes unitários, mas (GANESAN et al., 2010, 2013) abordam o problema do isolamento de testes, enquanto o foco da presente dissertação é no uso de padrões de testes unitários, testes de regressão e automatização de testes, assim como no reuso dos testes unitários.

Feng, Liu e Kerridge (2007) mostram uma abordagem de testes focada em LPSs baseadas em aspecto. Na proposta de Feng, Liu e Kerridge (2007), as LPSs baseadas em aspecto facilitam a criação

automatizada de casos de testes de aspecto que vão verificar os requisitos de qualidade da LPS.

O trabalho de Feng, Liu e Kerridge (2007) difere desta dissertação, pois possui uma solução focada para LPSs baseadas em aspectos e também na verificação dos atributos de qualidade da LPS. Nesta dissertação, os testes unitários são responsáveis por verificar somente os atributos funcionais da LPS e dos produtos individuais.

7 CONCLUSÕES

Na pesquisa realizada no desenvolvimento desta dissertação pode-se ver que o desenvolvimento tradicional de LPSs possui as desvantagens do alto investimento inicial, do risco de não utilizar os artefatos desenvolvidos e da dificuldade de aplicação em domínios onde se tem pouco conhecimento. A Engenharia de Linha de Produto Ágil (ELPA) surge para suprir essas desvantagens por meio da utilização de práticas ágeis, de forma que na construção de uma LPS seja necessário menos planejamento antecipado e menos investimento. Ao mesmo tempo, Lamancha, Usaola e Velthius (2009) ressaltam que é altamente desejável alcançar o uso de padrões de teste em LPSs uma vez que os padrões são soluções comprovadas para problemas recorrentes. Além disso, vários estudos apontam que há espaço para melhorias na área de testes de LPS.

Além disso, a abordagem proposta é apropriada para ser aplicada no mercado de aplicativos para dispositivos móveis, um mercado dinâmico e que tem crescido e se transformado rapidamente, pois facilita a utilização de LPS sem sofrer as desvantagens do desenvolvimento tradicional destas (NEVES; VILAIN, 2014a).

Também foi desenvolvido um *framework* de testes para dar suporte à prática de testes unitários no desenvolvimento de LPSs construídas através de uma abordagem reativa. Esse novo *framework*, denominado *Data-Driven Test Automation Framework* (DDTAF), foi criado com o propósito de adaptar padrões de testes unitários que facilitem a verificação das aplicações geradas por uma LPS. Os seguintes padrões de teste foram utilizados: Teste de Quatro Fases, *Framework* de Automação de Teste, Teste Guiado por Dados, Teste Parametrizado e Método Utilitário de Teste. O DDTAF também serve para ajudar a preencher a lacuna relacionada à seleção de teste, à reutilização de código de teste, testes de regressão, automação de testes e testes unitários.

Os passos da abordagem proposta nesta dissertação foram aplicados com sucesso na construção reativa de uma LPS. Começou-se com uma aplicação real desenvolvida com a prática de TDD. Esta aplicação foi desenvolvida sem pretensão de criar outras aplicações com o mesmo código ou testes. Depois novas *features* foram adicionadas no formato de histórias de usuário. As novas *features* eram variações de algumas *features* já existentes. Os novos pontos de variação foram criados a partir de novos testes e foi criado um arquivo de configuração para que cada aplicação pudesse ativar a variabilidade desejada. Por

meio do arquivo de configuração é mais fácil saber quais são os pontos de variação da aplicação, além de centralizar o local de configuração dos mecanismos de variabilidade e servir de guia para a construção de uma nova aplicação da LPS.

No exemplo foi observado que a abordagem e o *framework* de testes propostos facilitaram o reuso do código e dos testes. A maior parte do código de produção e testes unitários da primeira aplicação foi reusada na aplicações seguintes. Também foi possível observar uma alta porcentagem de reuso dos testes entre as aplicações, chegando a 100% quando novas aplicações são geradas selecionando variantes que já estão disponíveis na LPS. A prática de TDD ajudou a guiar as modificações nos testes unitários e no código quando houve a necessidade de modificar o comportamento existente para adicionar pontos de variação ou variantes na LPS. Enquanto isso o *framework* de testes, DDTAF, dava suporte a criação e execução dos testes responsáveis por testar os pontos de variação.

Como proposto anteriormente por Clements e Northrop (2001) o mesmo mecanismo de variabilidade usado para gerar as aplicações também é utilizado para executar os testes, o que facilita o desenvolvimento e execução dos testes. Os arquivos de configuração das aplicações do exemplo viraram parte do repositório de *core assets* e também foram utilizados na estratégia de testes de regressão, verificando que as configurações daquelas aplicações continuavam válidas, mesmo após modificações em outras partes da LPS.

Outro ponto a ser observado é que o *framework* de testes desenvolvido nesta dissertação poderá ser aplicado em LPSs construídas através de uma abordagem proativa (NEVES; VILAIN, 2014b), onde a engenharia de domínio é feita primeiro. Apesar do *framework* ter sido concebido para o desenvolvimento reativo de uma LPS, ele também pode ser utilizado em uma abordagem proativa. Ele também não requer que uma aplicação já esteja desenvolvida e poderia ser utilizado inicialmente no desenvolvimento dos *core assets* de uma LPS.

7.1 TRABALHOS FUTUROS

Nesta dissertação cinco padrões de teste unitário foram analisados e incluídos como parte do *framework* desenvolvido. Entretanto, outros padrões de teste unitário poderiam ser analisados detalhadamente, visto que padrões são aplicados em problemas recorrentes, é possível estudar a aplicação de outros padrões em problemas recorrentes de LPS.

Esta proposta ainda precisa ser utilizada em mais estudos de casos em empresas que desenvolvam produtos em segmentos onde: (1) não há muito conhecimento sobre o domínio para executar a engenharia de domínio; (2) não é possível prever as mudanças nos requisitos de produtos; (3) existe o risco de desenvolvimento de produtos que podem não ser reutilizados. Dessa forma estudar a viabilidade dessas práticas e estratégias permitindo ampliar e detalhar o conhecimento.

REFERÊNCIAS

BECK, Kent. **Test-Driven Development: By Example.** Boston: Addison-wesley Professional, 2002. 240 p.

CLEMENTS, Paul; NORTHROP, Linda. **Software Product Lines: Practices and Patterns.** 3. ed. : Addison-wesley Professional, 2001. 608 p.

COHN, Mike. **User Stories Applied: For Agile Software Development.** : Addison-wesley Professional, 2004. 304 p.

DÍAZ, Jessica et al. Agile product line engineering—a systematic literature review. **Journal Software—practice & Experience.** New York, p. 921-941. jul. 2011.

ENGSTRÖM, Emelie; RUNESON, Per. Software product line testing - A systematic mapping study. **Journal Information And Software Technology.** Newton, p. 2-13. jan. 2011.

FENG, Yankui; LIU, Xiaodong; KERRIDGE, J. **A product line based aspect-oriented generative unit testing approach to building quality components.** In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, 2007. COMPSAC 2007. 31ST ANNUAL INTERNATIONAL, 31., 2007, Beijing. **..** Beijing: Ieee, 2007. v. 2, p. 403 - 408.

GAMMA, Erich et al. **Design Patterns: Elements of Reusable Object-Oriented Software.** Boston: Addison-wesley Professional, 1994. 416 p.

GANESAN, Dharmalingam et al. Architecture-based unit testing of the flight software product line. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 14., 2010, Jeju Island. **Proceedings of the 14th international conference on Software product lines: going beyond.** Berlin: Springer-verlag, 2010. p. 256 - 270.

GANESAN, Dharmalingam et al. An analysis of unit tests of a flight software product line. **Science Of Computer Programming.** p. 2360-2380. 1 dez. 2013.

GHANAM, Yaser; MAURER, Frank. Extreme product line engineering refactoring for variability: A test-driven approach. **XP '10: Proceedings of Agile Processes in Software Engineering and Extreme Programming, 11th International Conference, XP 2010** (Lecture Notes in Business Information Processing, vol. 48), Trondheim, Norway, 1–4 Junho. Springer: Berlin, 2010; 43–57.

GHANAM, Yaser; MAURER, Frank. Extreme product line engineering: Managing variability and traceability via executable specifications. **Agile Conference, 2009 (AGILE '09)**. IEEE Computer Society: Silver Spring, MD, Agosto 2009; 41–48.

GIL, Antonio Carlos. **Métodos e técnicas de pesquisa social.** 3. ed. São Paulo: Atlas, 1991. 207 p.

KAKARONTZAS, George; STAMELOS, Ioannis; KATSAROS, Panagiotis. Product line variability with elastic components and test-driven development. **CIMCA '08: Proceedings of the 2008 International Conference on Computational Intelligence for Modelling Control and Automation.** IEEE Computer Society: Washington, DC, U.S.A., 2008: 146–151.

KÄSTNER, Christian et al. Toward variability-aware testing. In: INTERNATIONAL WORKSHOP ON FEATURE-ORIENTED SOFTWARE DEVELOPMENT, 4., 2012, Dresden.**Proceeding FOSD '12 Proceedings of the 4th International Workshop on Feature-Oriented Software Development.** New York: Acm, 2012. p. 1 - 8.

KERIEVSKY, Joshua. **Refactoring to Patterns.** : Addison-wesley Professional, 2004. 400 p.

KITCHENHAM, Barbara; CHARTERS, Stuart. **Guidelines for performing systematic literature reviews in software engineering (version 2.3).** Technical report, Keele University and University of Durham, 2007.

KRUEGER, Charles W. **Easing the transition to software mass customization.** Proceedings of the 4th International Workshop on Software Product-Family Engineering. 2001. p. 282 – 293.

LAMANCHA, Beatriz Pérez; USAOLA, Macario Polo; VELTHIUS, Mario Piattini. Software Product Line Testing - A Systemtic Review. In: 7TH INTERNATIONAL CONFERENCE ON SOFTWARE PARADIGM TRENDS (ICSOFT 2009), 7., 2009, Sofia. **Proceedings of the 7th International Conference on Software Paradigm Trends.** 2009. v. 1, p. 26 - 29.

LEE, Graham. **Test-Driven iOS Development.** Crawfordsville: Addison-wesley Professional, 2012. 256 p.

LEE, Jihyun; KANG, Sungwon; LEE, Danhyung. A survey on software product line testing. In: INTERNATIONAL SOFTWARE PRODUCT LINE CONFERENCE, 16., 2012, Salvador. **Proceeding SPLC '12 Proceedings of the 16th International Software Product Line Conference.** New York: Acm, 2012. v. 1, p. 31 - 40.

MCGREGOR, John D.. **Testing a Software Product Line.** Technical Report. Hanscom: Carnegie Mellon University, Software Engineering Institute, 2001. 68 p.

MESZAROS, Gerard. **XUnit Test Patterns:** Refactoring Test Code. Boston: Addison-wesley, 2007. 833 p.

NEVES, Glauco S.; VILAIN, Patrícia. **Reactive Variability Realization with Test Driven Development and Refactoring.** 26th International Conference on Software Engineering and Knowledge Engineering – SEKE, 2014a.

NEVES, Glauco S.; VILAIN, Patrícia. **Test Logic Reuse Through Unit Test Patterns: A Test Automation Framework for Software Product Lines.** 15th IEEE International Conference on Information Reuse and Integration – IRI, 2014b.

NETO, Paulo Anselmo da Mota Silveira et al. A systematic mapping study of software product lines testing. **Journal Information And Software Technology.** Newton, p. 407-423. mai. 2011.

POHL, Klaus; BÖCKLE, Günter; LINDEN, Frank J. van Der. **Software Product Line Engineering:** Foundations, Principles and Techniques. Secaucus: Springer-verlag New York, 2005. 473 p.

SILVA Edna L, MENEZES Estera M. **Metodologia da pesquisa e elaboração de dissertação.** 4. ed. rev. atual. – Florianópolis: UFSC, 2005. 138p.

SILVA, Ivonei Freitas da et al. Agile software product lines: a systematic mapping study. **Journal Software—practice & Experience.** New York, v. 41, p. 899-920. jul. 2011.

APÊNDICE A – Buscas nas Bases de Dados

IEEEXplore

Termo de busca 1
("test-driven" OR "test driven" OR "test first" OR tdd) AND (SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families")))
Termo de busca 2
("unit test" OR "unit testing") AND (SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families")))

ACM Digital Library

Termo de busca 1
(Title:("test-driven" or "test driven" or "test first" or tdd) or Abstract:("test-driven" or "test driven" or "test first" or tdd) or Keywords:("test-driven" or "test driven" or "test first" or tdd)) and (Title:(SPL or SPLE or (software and ("product line" or "product lines" or "product family" or "product families"))) or Abstract:(SPL or SPLE or (software and ("product line" or "product lines" or "product family" or "product families"))) or Keywords:(SPL or SPLE or (software and ("product line" or "product lines" or "product family" or "product families"))))
Termo de busca 2
((Title:("unit test" or "unit testing") or Abstract:("unit test" or "unit testing") or Keywords:("unit test" or "unit testing")) and (Title:(SPL or SPLE or (software and ("product line" or "product lines" or "product family" or "product families"))) or Abstract:(SPL or SPLE or (software and ("product line" or "product lines" or "product family" or "product families"))) or Keywords:(SPL or SPLE or (software and ("product line" or "product lines" or "product family" or "product families")))))

Scopus

Termo de busca 1
TITLE-ABS-KEY(("test-driven" OR "test driven" OR "test first" OR tdd) AND (SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))))
Termo de busca 2
TITLE-ABS-KEY(("unit test" OR "unit testing") AND (spl OR sple OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))))

CiteSeerX

Termo de busca 1
(title:("test-driven" OR "test driven" OR "test first" OR tdd) OR abstract:("test-driven" OR "test driven" OR "test first" OR tdd) OR keyword:("test-driven" OR "test driven" OR "test first" OR tdd)) AND (title:(SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))) OR abstract:(SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))) OR keyword:(SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))))
Termo de busca 2
(title:("unit test" OR "unit testing") OR abstract:("unit test" OR "unit testing") OR keyword:("unit test" OR "unit testing")) AND (title:(SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))) OR abstract:(SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))) OR keyword:(SPL OR SPLE OR (software AND ("product line" OR "product lines" OR "product family" OR "product families"))))

APÊNDICE B – Lista de Trabalhos Retornados

IEEEXplore

Termo de busca 1
Product Line Variability with Elastic Components and Test-Driven Development
Extreme Product Line Engineering: Managing Variability and Traceability via Executable Specifications
Code Conjurer: Pulling Reusable Software out of Thin Air
IEEE Recommended Practice for the Analysis of In-Band and Adjacent Band Interference and Coexistence Between Radio Systems
Termo de busca 2
A product line based aspect-oriented generative unit testingapproach to building quality components

ACM Digital Library

Termo de busca 1
Não houve retorno para o termo de busca 1
Termo de busca 2
Capturing performance assumptions using stochastic performance logic

Scopus

Termo de busca 1
Extreme product line engineering - Refactoring for variability: A test-driven approach
Product line variability with elastic components and test-driven develo
Proceedings - 7th Brazilian Symposium on Software Components, Architectures and Reuse, SBCARs 2013 - In Conjunction with CBSoft 2013 - 4th Brazilian Conference on Software: Theory and Practice
Proceedings of the CAiSE Forum 2011
Agile Processes in Software Engineering and Extreme Programming -

11th International Conference, XP 2010, Proceedings
DocumentBalancing Agility and Formalism in Software Engineering - Second IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007, Revised Selected Papers
Code conjurer: Pulling reusable software out of thin air
Agent-Oriented Software Engineering VII: 7th International Workshop, AOSE 2006 Revised and Invited Papers
Termo de busca 2
An analysis of unit tests of a flight software product line
Toward variability-Aware testing
Architecture-based unit testing of the flight software product line
A product line based aspect-oriented generative unit testing approach to building quality components
2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings
Towards test case reuse: A study of redundancies in android platform test libraries
Proceedings of the 17th European Conference on Software Maintenance and Reengineering, CSMR 2013
Proceedings of the 28th IEEE International Conference on Software Maintenance, ICSM 2012
Capturing performance assumptions using stochastic performance logic
Testing Software and Systems - 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Proceedings
Testing Techniques in Software Engineering - Second Pernambuco Summer School on Software Engineering, PSSE 2007, Revised Lectures
Systems thinking: Testing for software-based medical devices
Agent-Oriented Software Engineering VII: 7th International Workshop, AOSE 2006 Revised and Invited Papers
Risk reduction of on-board software development cycle

A development methodology for variant-rich automotive software architectures

CiteSeerX

Termo de busca 1
Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach
Matrikelnummer:
Termo de busca 2
Automated Testing of Graphical Models in Heterogeneous Test Environments
Capturing Performance Assumptions using Stochastic Performance Logic
The Coverage of the Object-Oriented Framework Application Class-Based Test Cases

APÊNDICE C – Fichamento dos Trabalhos Incluídos

Título
A product line based aspect-oriented generative unit testing approach to building quality components
Fonte
Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International (Volume:2)
Ano
2007
Autores
Yankui Feng, Xiaodong Liu, Kerridge, J.
Sumário
Este trabalho apresenta uma abordagem de testes unitários para uma LPS orientada a aspectos. Para isso, sabe-se que a LPS orientada a aspectos facilita a criação automática de casos de teste de aspectos com requisitos de qualidade específicos. O trabalho se baseia na observação de que abordagens e ferramentas de testes unitários existentes são fracos no fornecimento de mecanismos para realizar testes unitários que levem em conta a qualidade. Na execução deste trabalho, um repositório expansível de casos de teste de aspectos reusáveis foi desenvolvido. Além disso, uma ferramenta protótipo foi construída para verificar e facilitar a abordagem. Os benefícios desta abordagem incluem a melhora da efetividade e eficiência dos testes de componentes em features de qualidade e alto reuso de casos de teste de aspectos. Assim, o software baseado em componentes, tido como alvo, terá features com maior qualidade. Os experimentos realizados mostraram que a abordagem e a ferramenta são promissoras.

Título
Toward variability-aware testing
Fonte
FOSD '12 Proceedings of the 4th International Workshop on Feature-Oriented Software Development Pages 1-8
Ano
2012
Autores
Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch Sven Apel, Tillmann Rendel, Klaus Ostermann
Sumário
Neste trabalho foi feita a investigação sobre como executar teste unitários para todos os produtos de uma LPS sem gerar cada produto de forma isolada, por força bruta. Com a aprendizagem de análises cientes da variabilidade, foi (a) projetado e implementado um interpretador ciente de variabilidade e, de forma alternativa, (b) foi recodificada a variabilidade da linha de produtos para simular os casos de teste com um verificador de modelos. O interpretador internamente raciocina sobre a variabilidade, executando caminhos não afetados pela variabilidade apenas uma vez para toda a linha de produtos. O verificador de modelo alcança resultados semelhantes através da reutilização de poderosas análises que não têm nenhuma personalização. Foi feita uma investigação dos testes cientes da variabilidade utilizando estratégias de caixa branca (interpretador ciente da variabilidade), preta (codificação de variabilidade para JPF) e cinza (codificação de variabilidade para jpf-bdd). Em todas elas foi executado um caso de teste em todas as configurações de uma LPS de uma vez só, em oposição à força bruta ou à estratégia de amostragem. Foram percebidas coisas interessantes sobre o espectro entre as análises das caixas branca, cinza e preta, no que se refere ao esforço de implementação e flexibilidade.

Título
Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach
Fonte
The 11th International Conference on Agile Processes and eXtreme Programming (XP 2010)
Ano
2010
Autores
Yaser Ghanam, Frank Maurer
Sumário
Este trabalho inicia falando sobre as LPS e a sua necessidade de abordar a questão da variabilidade de *features*. O que significa que uma única *feature* pode exigir várias implementações para diferentes clientes, além das *features* poderem precisar de extensões opcionais que são necessárias por alguns, mas não todos os produtos. Por outro lado, os autores argumentam que a engenharia de LPS oferece às organizações uma vantagem econômica significativa devido à reutilização e oportunidades de customização em massa. E que, para as organizações ágeis, há uma barreira de adoção considerável devido à inicial natureza pesada de práticas LPS. Com base nessas premissas, este trabalho apresenta uma abordagem *bottom-up* orientada a testes para introduzir variabilidade em sistemas por meio de refatoração reativa de código existente. Para apoiar o trabalho, foi desenvolvido um *plug-in* do Eclipse para automatizar o processo de refatoração. A abordagem foi avaliada através de um estudo de caso para determinar a viabilidade e praticidade da abordagem. A contribuição deste trabalho é esta nova abordagem para introduzir variabilidade em sistemas existentes sob demanda, orientada a testes, para criar perfis de variabilidade para sistemas de software. Onde é utilizada refatoração sistemática a fim de injetar pontos de variação e variantes no sistema, sempre que necessário. Além do *plug-in* para Eclipse criado. A abordagem se mostrou viável e prática, com algumas limitações que serão trabalhadas. Os autores colocaram como trabalho futuro o estudo de testes de aceitação para descobrir como eles podem desempenhar um papel eficaz neste processo, pois eles são usados geralmente no nível de feature, o que permite que sejam pontos de ancoragem para a viabilidade de um sistema específico.

Título
Architecture-Based Unit Testing of the Flight Software Product Line
Fonte
Software Product Lines: Going Beyond Lecture Notes in Computer Science Volume 6287, 2010, pp 256-270
Ano
2010
Autores
Dharmalingam Ganesan, Mikael Lindvall, David McComas, Maureen Bartholomew, Steve Slegel, Barbara Medina
Sumário
Este trabalho apresenta uma análise da abordagem de teste unitário desenvolvida e utilizada pela NASA. O objetivo da análise é compreender, revisar e recomendar estratégias para melhorar a infraestrutura de testes de unidade existente, bem como capturar as lições aprendidas e as melhores práticas que podem ser usadas por outras equipes de LPS em seus testes unitários. A análise constatou que a abordagem de teste unitário no CFS tem muitas soluções práticas e boas que valem a pena considerar ao decidir como projetar a arquitetura de testes para uma LPS. Além disso, os autores revelam que o CFS tem sido aperfeiçoado ao longo de mais de 10 anos e passou por inspeções rigorosas e iniciativas de melhoria. Nesta pesquisa, o CFS abordou o difícil problema prático de teste de unidade onde os módulos muitas vezes dependem de outros módulos, tornando-os difíceis de separar. Ademais, os módulos podem também depender de features únicas e funções oferecidas pelos sistemas operacionais. Outro ponto levantado neste trabalho é que a abordagem da equipe CFS para testes unitários também lida com o uso de módulos (módulos reais ou *stubs* para testes) como um conjunto de pontos de variação, o que introduz um nível de flexibilidade que permite ao usuário do CFS usar também a mesma configuração para testes de integração incrementais, limitando assim os riscos que estão associados com testes de integração. Como trabalho futuro, os autores destacam a divulgação do *framework* de testes unitários que permite que os desenvolvedores de aplicativos façam testes unitários em suas aplicações sem executar os módulos principais.

Título
Extreme Product Line Engineering: Managing Variability & Traceability via Executable Specifications
Fonte
Agile Conference, 2009. AGILE '09.
Ano
2009
Autores
Yaser Ghanam, Frank Maurer
Sumário
Este artigo descreve uma abordagem para a gestão de LPS em um ambiente onde as práticas de XP são comuns. O trabalho descreve um caso em que se lida com a variabilidade no domínio de sistemas domésticos inteligentes para satisfazer uma série de requisitos. O trabalho investiga como a variabilidade e rastreabilidade de requisitos podem ser gerenciados através de especificações executáveis. Um estudo de caso foi utilizado para avaliar a abordagem, que forneceu percepções iniciais sobre a sua viabilidade e utilidade. Este trabalho contribuiu com uma abordagem nova e leve para gerenciar a variabilidade em LPS, e que permite que as organizações ágeis - especialmente as que adotam práticas XP – instanciem vários produtos a partir de um sistema central. Sendo que, esses produtos, embora diferentes, são tratados e gerenciados como um único sistema com pontos de variação. Os autores afirmam que combinar LPS e práticas do XP oferece vantagens significativas para os profissionais de software. Isso, além de reduzir a quantidade de retrabalho e o custo de produção de soluções customizadas, também faz com que seja viável para as organizações ágeis atingir clientes com necessidades diversas, sem ter que incomodar a agilidade de suas práticas. Assim, a abordagem proposta tem o potencial de reduzir substancialmente as barreiras de adoção da prática de LPS através de sua natureza incremental e não sobrecarregada. Ela utiliza artefatos de teste que são produzidos naturalmente em projetos XP, e permite que os clientes escolham a partir de variantes e contribuam para o modelo de variabilidade quando as variantes disponíveis não são satisfatórias para eles. Complementando este trabalho, os autores se encontram em processo de combinação de resultados do presente trabalho com conceitos como gestão de reutilização para ampliar as vantagens de adotar uma LPS ágil.

Título
Product Line Variability with Elastic Components and Test-Driven Development
Fonte
Computational Intelligence for Modelling Control & Automation, 2008 International Conference on
Ano
2008
Autores
George Kakarontzas, Ioannis Stamelos, Panagiotis Katsaros
Sumário
Neste trabalho foi proposto um novo método para a variabilidade e reutilização de componentes de LPS, que é impulsionado por suítes de testes. O método aborda a questão da introdução da variabilidade em uma LPS usando a customização dos *core assets* existentes. Tecnicamente a abordagem dos autores é uma abordagem de substituição do componente, onde novos componentes são criados reutilizando componentes já existentes e casos de teste. Onde o componente substituto pode ser criado internamente utilizando muitos dos mecanismos de variabilidade existentes (herança, substituição de componente, *plug-ins*, *templates*, parâmetros, geradores, aspectos, entre outros). Outro ponto abordado pelos autores se refere às suítes de testes que ajudam à correção da colocação da nova variante do componente no repositório de *core assets* que está organizado para os componentes de uma forma hierárquica para permitir a sua reutilização eficaz, busca e recuperação. Além disso, testes também auxiliam na verificação de que o produto está correto após a sua evolução, e conjuntos de teste estão disponíveis para repetir o processo de garantia de qualidade no novo produto. Como trabalho futuro os autores pretendem se concentrar no desenvolvimento do repositório de componentes elásticos e no uso de métricas específicas, como a similaridade entre os componentes para acelerar o processo de busca em repositórios com um grande número de componentes. Além de aplicar essa nova abordagem em vários estudos de caso para refinar ainda mais.

Título
An analysis of unit tests of a flight software product line
Fonte
Special Section on International Software Product Line Conference 2010 and Fundamentals of Software Engineering (selected papers of FSEN 2011)
Ano
2013
Autores
Dharmalingam Ganesan, Mikael Lindvalla, David McComasb, Maureen Bartholomewb, Steve Slegelb, Barbara Medinab, Rene Krikhaarc, Chris Verhoefc, Lisa P. Montgomeryd,
Sumário
Este trabalho apresenta uma análise da abordagem de teste unitário desenvolvida e utilizada por uma equipe da NASA, com objetivo de compreender, analisar e recomendar estratégias para melhorar a infraestrutura de testes utilizada pela NASA, assim como capturar as melhores práticas que podem ser usados por outras LPS. A análise constatou que esta abordagem de teste unitário incorpora soluções práticas e úteis, como permitir o teste unitário sem a necessidade de hardware e características especiais do sistema operacional, definindo implementações *stub* de módulos dependentes. É proposto o uso de um *framework* de *stubs* para diminuir a necessidade de componentes dependentes na hora da escrita e execução dos testes unitários. Além disso, é descrita a análise da estratégia de teste unitário das LPSs do CFS (Core Flight Software System) e casos de teste unitários de acompanhamento e de ambiente de teste. O CFS tem enfrentado com sucesso o problema da prática de testes unitários onde os módulos muitas vezes dependem de outros módulos. Eles também têm abordado a questão de que os módulos dependem muitas vezes características únicas e funções oferecidas pelos sistemas operacionais. O CFS abordou este problema através da introdução de *mocks* e camadas de abstração para lidar com a variabilidade em LPSs. A abordagem da CFS para teste unitário também permite usar a mesma configuração para testes de integração incrementais, dependendo da situação, limitando assim os riscos que estão associados aos testes de integração. No entanto, os testes unitários e o tipo de testes de integração incrementais e outras formas de testes devem ser realizados, a fim de detectar os tipos de defeitos que tais testes não conseguem detectar.

APÊNDICE D – Publicações

Publicação 1:

Título

Reactive Variability Realization with Test Driven Development and Refactoring

Conferência

26th International Conference on Software Engineering and Knowledge Engineering - SEKE

Qualis

B1

Resumo

Software product line is a practice that has proven its advantages since it can offer to a company the reduction of time to market, the decrease of development costs, the increase of productivity and the improvement of the final product quality. However, this practice requires a high initial investment and offers long-term risks to dynamic markets where changes are difficult to predict. One of these markets is the mobile application development, which presents a growing demand, with smartphone and tablets having already surpassed sales of PCs and notebooks. Currently, proposals bring the advantages of software product line for dynamic markets through the use of agile software development practices, which is called Agile Product Line Engineering (APLE). This paper investigates the use of test-driven development (TDD) and refactoring techniques for performing reactive variability in APLE. The variability mechanism chosen is the configuration file that allows achieving more than one platform, an important problem in mobile application development. In that manner, new products can be built as needed, without the high upfront investment, but with a code easier to maintain.

Publicação 2:
Título
Test Logic Reuse Through Unit Test Patterns: A Test Automation Framework for Software Product Lines

Conferência
15th IEEE International Conference on Information Reuse and Integration - IRI

Qualis
B2

Resumo
Software product line (SPL) brings benefits such as lower time-to-market, less development costs, increased productivity and improved quality. The quality assurance can be reached through the testing area, however this area still has challenges and gaps in the SPL development. Since not all testing techniques used in a single product development can be applied to SPL, because of artifacts variabilities, further adaptations and new proposals are required. Our proposal thus is to adapt some unit tests patterns to SPL needs. The Test Automation Framework and Data-Driven Test patterns can provide the reuse of test logic and the automation of implementation mechanisms, reducing the effort required to test the variations of each application. Thus, we propose the Data-Driven Test Automation Framework to be used during the application engineering to configure the tests through Parameterized Tests and verify the correctness of the generated applications. An example of a SPL is also presented.