Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Yun Hu Lee

Arquitetura para Linhas de Produto de Software de E-Commerce usando Play Framework

Florianópolis

2017

Yun Hu Lee

Arquitetura para Linhas de Produto de Software de E-Commerce usando Play Framework

Monografia submetida ao Programa de Graduação em Ciências da Computação a obtenção do Grau de Bacharel
Orientadora: Prof. Dr. Patrícia Vilain

Florianópolis

2017

Yun Hu Lee

Arquitetura para Linhas de Produto de Software de E-Commerce usando Play Framework

Esta Monografia foi julgada adequada para obtenção do Título de “Bacharel” e aprovada em sua forma final pelo Programa de Graduação em Ciências da Computação

Florianópolis, 28 de Julho de 2017.

_____

Prof. Dr.

Rafael Luiz Cancian

Coordenador do Curso

Banca Examinadora:

_____

Prof. Dr. Patrícia Vilain

Orientadora

_____

Prof. Dr. Leandro José Komosinski

Universidade Federal de Santa Catarina

_____

Prof. Dr. Raul Sidnei Wazlawick

Universidade Federal de Santa Catarina

AGRADECIMENTOS

À minha mãe, Eun Kyung pelo seu amor e apoio; ao meu pai falecido, Jung Jai pela sua presença e fé; ao meu irmão Yun Sei por estar presente.

À minha orientadora, Prof. Patrícia Vilain, pelos valiosos conselhos e forte apoio durante o processo de realização deste trabalho.

A todos os meus amigos e pessoas especiais que me incentivaram no decorrer do curso e na realização deste trabalho.

Meus sinceros agradecimentos.

RESUMO

A linha de produto de software baseia-se em um planejamento de reutilização de peças já desenvolvidas em novos projetos automobilísticos, sem a necessidade de reprojetar toda a plataforma de criação. Isso traz, além da questão econômica, um aumento do público alvo e melhora na qualidade das peças, já que essas, por serem utilizadas novamente, são revistas com mais frequência, proporcionando uma confiabilidade maior. O trabalho apresenta uma arquitetura de LPS utilizando o Play Framework no contexto de um sistema de comércio eletrônico. A proposta é composta por três atividades de desenvolvimento (análise de requisitos, arquitetura e codificação). A abordagem foi exemplificada e validada por meio três aplicações a partir da LPS construída.

Palavras-chave: Linha de Produto de Software. E-Commerce. Scala. Play Framework.

ABSTRACT

A SPL is based on a reuse of parts developed in automobile industry, without the need to redesign an entire platform. It has been possible an increase of the target public and improvement in the quality of product, since these are reviewed more frequently, thus can provide greater reliability. This work presents a SPL architecture using Play Framework in the context of an e-commerce system. The proposal consists of three development activities (requirements analysis, architecture and code). The approach was exemplified and validated by three applications from the constructed SPL.

Keywords: Software Product Lines. E-Commerce. Scala. Play Framework.

Lista de Figuras

Lista de Quadros

Sumário

1. Introdução

Este capítulo apresenta uma introdução do que é elaborado neste trabalho, introduzindo ao leitor o contexto, a motivação e justificativa para a sua elaboração, bem como os objetivos a serem alcançados e a estrutura do trabalho.

1.1. Contextualização do tema

Cada vez mais as organizações são confrontadas por uma existente pressão do mercado consumidor, que anseia, constantemente, por inovações em diferentes necessidades. Estes desejos são atendidos sob a forma de diversos produtos customizados, que levam em conta a real expectativa de um determinado segmento e as tendências do mercado.

Na tentativa de satisfazer os mais variados desejos dos clientes sem comprometer os indicadores de produtividade e custo, surgiu na indústria automobilística do século XX um novo conceito de produção. Este consistia em realizar compartilhamento de plataforma para a fabricação de diversos modelos (POHL, BÖCKLE e LINDEN, 2005).

Em princípio, tal conceito baseia-se em um planejamento de reutilização de peças já desenvolvidas em novos projetos automobilísticos, sem a necessidade de reprojetar toda a plataforma de criação. Isso traz, além da questão econômica, um aumento do público alvo e melhora na qualidade das peças, já que essas, por serem utilizadas novamente, são revistas com mais frequência, proporcionando uma confiabilidade maior (LINDEN, SHMID e ROMMES, 2007).

Na indústria de software, um produto desenvolvido com esta abordagem é chamado de Linhas de Produto de Software (LPS) e neste contexto o termo “plataforma” significa uma coleção de artefatos reusáveis (POHL, BÖCKLE e LINDEN, 2005). Estes artefatos são reutilizados de forma sistemática e consistente para construção de softwares. Artefatos reutilizáveis abrangem todos os tipos de artefatos durante desenvolvimento de software, tais como modelos de requisitos, modelos de arquitetura, componentes de software e planos de teste.

Nos últimos anos, sistemas de web evoluíram rapidamente de uma simples colação de páginas estáticas para aplicações complexas e ricas de conteúdo. O aumento da relevância econômica e de complexidade dessas aplicações trazem junto a necessidade de adoção de técnicas e modelos que trazem um ganho maior no tempo de desenvolvimento e em fatores de qualidade. E por outro lado, metodologias tradicionais são frequentemente baseadas nas habilidades e técnicas individuais de programadores e que nem sempre são aplicados os princípios de engenharia de software.

O crescente aumento do comércio eletrônico, vem motivando o desenvolvimento de novas tecnologias e padrões. Metodologias e técnicas de desenvolvimento de sistemas para E-Commerce têm sido propostas pela

literatura e aplicadas na prática por empresas de grande porte como IBM, Google e Microsoft (ALBERTIN, 2004).

Sistemas e-commerce se caracterizam basicamente por fornecer serviços relacionados a compras na internet. Estes formam uma infraestrutura comum de serviços. Com base nisso, entende-se que seja possível gerenciar tais serviços, similaridades ou variabilidades, por meio da abordagem de LPS. Assim, vários sistemas podem ser desenvolvidos por meio da instanciação de tal infraestrutura comum para o domínio de sistemas E-Commerce.

1.2. Motivação

Das várias razões que levam as organizações adotarem o paradigma de LPS, os motivos mais fortes são, geralmente, baseados nas considerações econômicas. Devido a sua característica de reutilizar extensivamente os artefatos já produzidos, ocorre uma melhoria na redução de tempo e de custo de desenvolvimento.

De fato, vários estudos de caso já demonstraram um sucesso nas organizações que adotaram esta abordagem. E não só isso, elas levaram sua capacidade de produção para além da sua área de mercado dominante - para outras áreas ainda não exploradas - elevando rapidamente sua participação naquele mercado também.

Um grande número de casos de sucesso das organizações tornou essa área de pesquisa ascendente, pois propiciou o desenvolvimento de ferramentas que permitam uma implementação efetiva de uma LPS.

Junto com esses casos de sucesso, o desenvolvimento de ferramentas relacionadas ao desenvolvimento de LPS ganhou força. Uma das abordagens para construção de uma LPS é usar ferramentas como AHEAD e Feature-House, as quais usam o recurso de técnicas de geração de código para criar diferentes produtos de software. Outra opção é usar linguagens de programação que possuem um suporte para desenvolver uma LPS. Em particular, a linguagem Scala oferece a expressividade necessária para desenvolver uma LPS. Porém, ainda não é possível encontrar casos reais que utilizam esta linguagem para o desenvolvimento de LPS (THAKER, 2008).

Sistema de E-Commerce pode ser considerado como uma aplicação de web derivado de um conjunto de recursos reusáveis e de artefatos os quais capturam abstrações específicas no domínio como por exemplo: carrinho de compras, cadastro de usuário, forma de pagamento, em um sistema de e-commerce.

1.3. Objetivos

Com base na contextualização do tema e na motivação, são apresentados a seguir o objetivo geral e os objetivos específicos deste trabalho.

1.3.1. Objetivo Geral

Este trabalho propõe uma arquitetura de LPS de E-Commerce usando Play Framework em Scala. Após a construção do modelo proposto, o mesmo será implementado para demonstrar o funcionamento: algumas aplicações serão produzidas combinando os artefatos já desenvolvidos na fase de análise de domínio. O objetivo é estudar a viabilidade da abordagem num cenário de mundo real.

1.3.2. Objetivos Específicos

Este trabalho apresenta os seguintes objetivos que devem ser alcançados para satisfazer o escopo deste trabalho.

1. Análise de domínio de Sistemas E-Commerce: tem como objetivo modelar domínio dos Sistemas E-Commerce, identificação das características comuns e variáveis entre os sistemas E-Commerce.

2. Escolha de ferramentas e metodologias que serão utilizadas no processo de desenvolvimento da LPS.

3. Desenvolvimento da arquitetura do domínio e a construção dos componentes correspondentes às *features*.

4. Desenvolvimento de produtos da LPS através do reuso sistemático dos artefatos do repositório de artefatos comuns.

1.3.3. Estrutura do Trabalho

O trabalho é apresentado em seis capítulos. No capítulo dois é feita a fundamentação teórica do trabalho onde são apresentados os principais conceitos de LPS e as ferramentas de suporte utilizadas. No capítulo três é apresentada a revisão do estado da arte através de uma revisão sistemática da literatura. No capítulo quatro é descrita a proposta de desenvolvimento da LPS apresentando a análise de domínio do problema e em seguida a construção das *features*. O capítulo cinco apresenta um estudo de caso mostrando três exemplos com base na LPS construída. O capítulo seis conclui o trabalho, mostrando o que foi feito, quais foram as contribuições e sugestões de trabalhos futuros.

2. Fundamentação teórica

Neste capítulo são descritos alguns conceitos que dão o embasamento teórico necessário para o entendimento do trabalho desenvolvido. Serão

abordados os conceitos da LPS e as ferramentas que serão utilizadas para a execução do mesmo.

Uma vez que os conceitos e tecnologias empregadas são de complexidade elevada, não foi considerada uma abordagem profunda de cada conceito, pois foge ao escopo deste trabalho.

2.1. Linha de Produto de Software (LPS)

De acordo com (CLEMENTS e NORTHROP, 2001), uma Linha de Produto de Software (LPS) é uma família de produtos que compartilham um conjunto comum e gerenciado de *features* ou funcionalidades. Estas satisfazem as necessidades de um mercado específico e são desenvolvidas a partir de recursos comuns de uma maneira pré-definida.

2.1.1. Atividades da Linha de Produto de Software

A metodologia de desenvolvimento da LPS requer em dois processos. A primeira, chamada de engenharia de domínio, é responsável por estabelecer uma plataforma reusável e definir a comunalidade e a variabilidade de uma linha de produto. A segunda é a engenharia de aplicação. É através desta que os produtos da plataforma estabelecida na engenharia de domínio são derivados (POHL, BÖCKLE e LINDEN, 2005).

A metodologia de desenvolvimento da LPS requerem duas atividades – Engenharia de Domínio(ED), na qual os componentes que fazem parte da LPS são desenvolvidos já visando o reuso, e a Engenharia de Aplicação(EA) é fase onde as aplicações serão produzidas utilizando os artefatos criados na atividade anterior.

2.1.1.1. Engenharia de Domínio (ED)

ED é o processo para entender e definir o escopo da família de sistema através da análise de domínio. E em seguida é desenvolvido um conjunto de assets configuráveis e reusáveis. Análise de domínio envolve uma pesquisa para obter o entendimento do domínio do ponto de vista do especialista de domínio. Isso é obtido através de reuniões com especialista de domínio, consultas em documentações e realização de enquetes. O aspecto importante da análise de domínio é obter uma definição clara da terminologia e de processos, as quais ajudam reduzir a discrepância entre os requisitos e projeto (BOOCH, 1991). A construção de ativos reusáveis significa a criação de componentes ou padrões de projeto para ser usado na criação de produtos concretos.

2.1.1.2. Engenharia de Aplicação (EA)

Este é o processo de especificação e de configuração de produtos individuais baseados na informação de configuração fornecida por engenheiro de aplicação e por ativos reusáveis gerados na etapa de ED. O processo pode ser tanto manual ou automático dependendo do framework.

Novos requisitos podem ser descobertos durante EA e podem ser passados à atividade de ED para estender o escopo da linha de produto e definir novos ativos reusáveis, assim novos produtos podem ser gerados. Portanto ED e EA são processos complementares. O custo inicial que envolve a atividade ED é maior comparada a uma metodologia de desenvolvimento convencional, porém o investimento é amortizado sobre vários produtos gerados e sobre a qualidade de software melhorado.

2.1.2. Variabilidade

A chave do paradigma da LPS é a modelagem e a gestão de variabilidade. A variabilidade de software refere-se a habilidade do sistema de ser eficientemente customizado ou de ser configurado para ser utilizado em contextos diferentes. A variabilidade é alcançada através de pontos de variação. Tais pontos na arquitetura ou na implementação significam alguma variância em funcionalidade.

2.1.3. Artefatos reusáveis

Artefatos reusáveis descrevem qualquer artefato que fazem parte do processo de reuso. Existem várias formas e granularidades para artefatos reusáveis, incluindo padrão de projeto orientado a objeto [GAM95], classes individuais, componentes e mixins (classes que contém métodos para ser usados por outras classes sem ter que ser a classe pai) [SZY03], linguagens de domínio específico (DSLs) [CZA05A], padrão de negócio de domínio específico como Enterprise Architecture Patterns [FOW03a], ou frameworks que implementam um estilo arquitetural em comum [SHA96]. Esses artefatos fazem parte do espaço de problema e do espaço de solução. A relação de problema de espaço de problema e de solução é mostrada na figura abaixo.

Figura 1 - A relação de problema de espaço de problema e de solução (POHL, BÖCKLE e LINDEN, 2005)

O espaço de problema descreve o conjunto de artefatos que representam o conhecimento de domínio, como os requisitos, glossários, DSLs e modelos do domínio. O espaço de solução descreve o conjunto de artefatos implementados que são usados para gerar um produto, como algoritmos, componentes concretos e *templates* de código. Os dois espaços são relacionados através de conhecimento de transformação, o qual descrevem mapeamentos da configuração de produto para a implementação. Mapeamentos entre os dois não são necessariamente um-para-um. Por exemplo, múltiplos DSLs podem ser mapeados para a mesma implementação, onde diferentes DSLs são disponíveis para usuários com conhecimentos e aptidão diferentes.

2.1.4. Modelo de Feature

O Modelo de *Feature* foi proposto por (KANG, 1990) como um mecanismo para gerenciar a variabilidade em uma família de sistema. Este também foi uma parte do trabalho de *Feature-Oriented Domain Analysis* (FODA). FODA definiu algumas estruturas básicas do modelo de *feature*, como *features* opcionais e obrigatórios e o relacionamento entre conjuntos de *features*, como grupos de *features* ou-exclusivo. A notação original de FODA foi estendida através de modelagem baseada em cardinalidade (CZARNECKI, HELSEN e EISENECKER, 2005), de atributos de *features*, atributos de referência, configurações de multi-estágio e multi-nível (CZARNECKI, HELSEN e EISENECKER, 2005), restrições externas, e cardinalidade de *features* agrupadas (CZARNECKI e KIM, 2005). Essas extensões permitem um grau maior de expressão e abre um potencial maior para suas aplicações em LPS. O processo de remoção de variabilidade do Modelo de *Feature* através da seleção ou eliminação de *features* é chamado de configuração.

A chave do paradigma da LPS é a modelagem e a gestão de variabilidade. A variabilidade de software refere-se à habilidade do sistema de ser eficientemente customizado ou de ser configurado para ser utilizado em

contextos diferentes. A variabilidade é alcançada através de pontos de variação. Tais pontos na arquitetura ou na implementação significam alguma variância em funcionalidade.

Já o conceito de *feature* refere-se a uma propriedade ou funcionalidade de um sistema que é relevante a interesses dos envolvidos no projeto. As partes comuns e variáveis entre produtos de uma mesma família podem ser representadas como *features*. Geralmente estas são capturadas na engenharia de domínio usando o modelo de *feature*, o qual é frequentemente empregado para modelar variabilidades em uma LPS.

Uma *feature* do modelo de *feature* pode ser classificada como: (1) alternativa, (2) obrigatória e (3) opcional. A alternativa é representada por arestas que estão conectadas por um arco. Se o arco for vazio deve-se escolher apenas uma das alternativas (XOR), se for preenchido é permitido escolher mais de uma alternativa (OR). Uma *feature* opcional é representada por uma aresta conectada por um círculo vazio e quando o círculo é preenchido torna-se classificado como obrigatório (DEURSEN e KLINT, 2001).

As informações das partes comuns e variabilidades capturadas durante a modelagem das *features* servem como base para o desenvolvimento de artefatos da LPS (J LEE, 2010).

Pode-se definir a descrição dos produtos possíveis de serem construídos pela LPS no escopo da 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 através de um modelo *feature*. Segundo (POHL, BÖCKLE e LINDEN, 2005), o modelo de *feature* serve como documentação das *features* de uma LPS e facilita a discussão entre os interessados no projeto. A Figura abaixo mostra o modelo de *feature* 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.

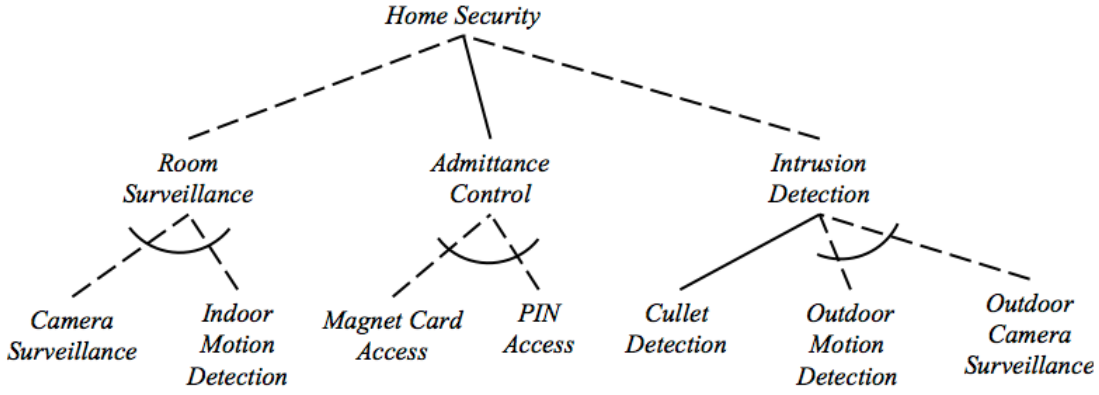

Figura 2 – Modelo de Feature de um sistema de automatização residencial (POHL, BÖCKLE e LINDEN, 2005),

2.2. Sistema de E-Commerce

Sistema de E-Commerce se refere a sistemas que habilitam qualquer tipo de transação ou negócio através da internet. As empresas devem suportar um grande número de visitantes e transações além de coordenar múltiplos *stakeholders* para entregar o produto ou serviço para consumidores.

Existem três tipos comuns de transações: *Business-to-Consumer* (B2C), *Business-to-Business* (B2B) e *Consumer-to-Consumer* (C2C) (LAU, 2006). As empresas, como a Submarino e Amazon, vendem seus produtos ou serviços para indivíduos, sendo enquadradas como B2C. Já o modelo de negócio da empresa Alibaba é na base de B2B, pois ela intermedia transações de serviços ou produtos entre empresas. Por último, o modelo de negócio C2C permite que o próprio consumidor final venda diretamente para outro consumidor final. As principais empresas que fazem a intermediação são: Mercadolivre, E-bay e Amazon Marketplace.

2.3. Ferramentas de Suporte

Esta seção descreve de forma breve as ferramentas utilizadas durante o desenvolvido do trabalho.

2.3.1.1. Scala

Scala foi desenvolvida em 2001 por Martin Odersky e por seu grupo na *École Polytechnique Fédérale de Lausanne* (EPFL) na Suíça. É uma linguagem de programação de propósito geral, diga-se multiparadigma, projetada para expressar padrões de programação comuns de uma forma concisa, elegante e *type-safe*. Pode-se dizer que ela é uma linguagem de programação que desestimula ou impede erros de tipagem. Ela incorpora recursos de linguagens orientadas a objetos e funcionais (ODERSKY, SPOON e VENNERS, 2008). Também é plenamente interoperável com Java. Scala é software de código aberto. Mesmo sendo recente no mercado, vem conquistando cada vez mais espaço nos últimos anos. Empresas gigantes como o Twitter, Foursquare, Linkedin e Quora adotaram a linguagem para desenvolvimento de seus serviços. Uma das primeiras diferenças entre Scala e uma linguagem como Java é que Scala tem maior foco no paradigma funcional.

2.3.1.2. SBT

SBT (Simple Build Tool) é uma ferramenta de automação de Build para projetos em Scala e em Java. Ela simplifica o processo de desenvolvimento de *build* e auxilia na manutenção do projeto.

Existem funcionalidades que vão além daquelas fornecidas por outras ferramentas convencionais, tornando SBT a principal ferramenta de automação de *build* de projetos de Scala. Entre as funcionalidades pode-se destacar algumas que são úteis para desenvolvimento deste trabalho (SAXENA, 2013):

- SBT pode ser configurado usando a linguagem de domínio específico baseado em Scala e ser estendido conforme a necessidade do projeto.
- Visto que o processo de compilação de Scala é bastante demorado, SBT implementa vários algoritmos e heurísticas para otimizar o processo de compilação (SAXENA, 2013), tornando este processo bastante rápido.
- Os comandos de SBT podem rodar em modo de execução engatilhado. Isso significa que as tarefas específicas podem ser executadas de acordo com a regra específica definida.
- SBT possui suporte para framework de testes como ScalaCheck, Specs2 e ScalaTest. Adicionalmente, testes de JUnit também podem ser executados usando o plugin *junit-interface*. SBT permite executar testes de forma seletiva de acordo com alguma regra ou através de argumentos passados para o framework de teste.
- Por último, a execução de testes em paralelo. O paralelismo de tarefas refere-se à execução de um ou mais tarefas independentes de forma concorrente. Paralelismo garante o uso escalável e eficiente de recursos do sistema, sendo esta uma função útil quando se trata de execução de testes (SAXENA, 2013).

2.3.1.3. Play framework

Play é um *framework* para desenvolvimento de aplicação de web de código aberto. Ele é escrito em Scala e em Java e segue o padrão de arquitetura MVC (Modelo-Apresentação-Controle) (HILTON, BAKKER e CANEDO, 2013).

Desde a versão 2.0, o núcleo deste framework foi reescrito em Scala. A parte de Build e Deployment migrou para o SBT e passou a usar o sistema de apresentação (*template*) na linguagem Scala.

Play foi altamente inspirado por Ruby on Rails e Django, e é similar a essa família de *frameworks*. Play é totalmente independente do ambiente de JavaEE e muito mais leve comparado a este. Isso torna o Play mais simples de desenvolver em relação a outras plataformas baseadas em Java.

Comparando a outros *frameworks*, este é:

1) Sem-estado (*Stateless*) e totalmente RESTful. Isso significa que considera cada requisição como uma transação independente e não há vínculo com qualquer requisição anterior, de forma que a comunicação consista em pares de requisição e resposta independentes. Além do mais, não existe a sessão de Java EE por conexão como acontece em frameworks baseados em Java (HILTON, BAKKER e CANEDO, 2013).

2) Tal framework funciona com métodos estáticos: todos pontos de entrada de todos controladores são declarados como estáticos.
3) Entrada e saída assíncrona: pelo fato de utilizar JBoss Netty como servidor web, Play pode fornecer requisições longas de forma assíncrona em vez de amarrar thread de HTTP fazendo regra de negócio (*business logic*) como framework de Java EE (HILTON, BAKKER e CANEDO, 2013).
4) Arquitetura modular: assim como Rails e Django, Play utiliza o conceito de módulos.
5) Suporte à linguagem Scala: Play utiliza Scala internamente, pode-se utilizar tanto o API de Scala como o de Java.

3. Estado da arte

Este capítulo faz uma revisão sistemática de literatura com o objetivo de apresentar o estado da arte sobre metodologias e técnicas de desenvolvimento de LPS para sistemas e-commerce em Scala. Para tanto, foram abordadas as seguintes questões: (1) identificação de quais são os métodos específicos utilizados no desenvolvimento de LPS para sistemas e-commerce; (2) e/ou usando Scala.

O capítulo está organizado da seguinte forma: a seção 1 apresenta os procedimentos de coleta dos dados; a seção 2 apresenta uma análise sobre os resultados obtidos com a revisão sistemática; a seção 3 apresenta uma síntese dos trabalhos selecionados para leitura na íntegra e que efetivamente contribuem com o estado da arte do tema.

3.1. Coleta de dados

Uma revisão sistemática de literatura é uma abordagem rigorosa e bem definida para identificar, avaliar e interpretar todas as pesquisas disponíveis com relação a um tema específico de interesse (KITCHENHAM, 2004).

Os elementos que fornecem evidências de pesquisa sobre um tema especifico são classificados como estudos primários. Na realização desta revisão sistemática foram focados dois pontos principais: (i) identificar quais são os estudos existentes no desenvolvimento de uma LPS para sistemas e-commerce usando Scala; e (ii) identificar técnicas específicas aplicadas a LPS para sistemas e-commerce usando Scala.

3.2. Perguntas de Pesquisa

Para conduzir toda a metodologia da revisão sistemática, foram definidas uma questão de pesquisa primária (Q1) e uma questão secundária (Q2):

- Q1 – estudos que relacionam a abordagem de LPS com o desenvolvimento de sistemas e-commerce; e
- Q1.1 – usando Scala
- Q2 - estudos que apresentam metodologias e/ou técnicas específicas de desenvolvimento de sistemas e-commerce com base na abordagem de LPS.

3.2.1. Estratégia de Busca

Tais questões de pesquisa norteiam a definição de termos de busca que é fundamental para a realização de uma revisão sistemática de literatura. Os termos a serem definidos devem englobar o maior número possível de estudos relacionados ao tema em questão. Para tanto, estes foram definidos no Quadro 1 com base em três palavras-chave principais e suas variações: “software product line”, “e-commerce”, “scala”.

Termos de busca
1. (E-Commerce OR “Electronic Commerce”) AND (SPL OR SPLE OR (software AND (product line OR product lines OR product family OR product families)))
2. (E-Commerce OR “Electronic Commerce”) AND (Scala AND (SPL OR SPLE OR (software AND (product line OR product lines OR product family OR product families))))

Quadro 1 - Termos utilizados na busca

As fontes de dados eletrônicas indexadas que foram selecionadas para o levantamento dos dados estão expostas no Quadro 2.

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

Quadro 2 - Bases de dados e seus endereços eletrônicos

3.2.2. Critério de seleção

Além da definição dos termos de busca e das fontes de dados, critérios de inclusão e exclusão são extremamente importantes, pois guiam a leitura na íntegra dos trabalhos mais relevantes. Assim, os critérios de inclusão estabelecidos para atender a cada uma das questões de pesquisa são:

- Relacionados a desenvolvimentos de produtos e-commerce com LPS; e/ou usando Scala;
- Apresentam princípios, metodologias e técnicas utilizadas para o desenvolvimento de LPS para sistemas e-commerce, e/ou usando Scala;
- Publicados entre janeiro de 2005 e abril de 2017;
- Publicados em língua inglesa.

Os critérios de exclusão são:

- Qualquer estudo que não se enquadre no critério de inclusão;
- Estudos duplicados, encontrados anteriormente em outra(s) fonte(s);
- Estudos que não puderam ser recuperados (não-disponíveis).

Uma vez recuperados os estudos por meio de busca às fontes de dados selecionadas, deve-se realizar um processo de seleção preliminar por meio da verificação dos critérios de inclusão e exclusão definidos, bem como da leitura dos títulos e dos resumos de cada estudo recuperado. Dessa forma, eliminam-se trabalhos que não satisfazem tais critérios de inclusão e exclusão. Se algum critério de exclusão referente à Q1 for identificado, o trabalho será descartado para leitura na íntegra.

3.3. Análise dos Resultados

A consulta foi realizada em 20/04/2017. Houve um retorno de 21 trabalhos, porém haviam 2 trabalhos repetidos em bases diferentes, resultando em 19 trabalhos. O Quadro 3 mostra quantos trabalhos foram retornados, incluindo os redundantes. Após a remoção das redundâncias, foram selecionados os trabalhos para serem avaliados na etapa seguinte. O termo 2, que é uma pesquisa mais específica, não retornou nenhum estudo.

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

Quadro 3 - Resultado das buscas por trabalhos primários

3.4. Síntese dos Trabalhos Selecionados

Nesta seção são apresentados os estudos efetivamente selecionados durante a revisão sistemática, ressaltando as técnicas e métodos aplicados em

cada um. Os trabalhos selecionados e as respostas relacionadas são mostrados no Quadro 4 a seguir. O título das subseções é o título do artigo para melhor identificação dos estudos.

Título	**Questões**
A Software Product Line Approach for E-Commerce Systems (Laguna, M.A. Hernández, C.)	Q1 e Q2
Best Practices of RUP in Software Product Line Development	Q1
Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates	Q1
Best Practices of RUP in Software Product Line Development	Q1
Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line Model	Q1

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

A Software Product Line Approach for E-Commerce Systems

O trabalho do grupo de pesquisa (MIGUEL A. LAGUNA, 2010) desenvolveu uma LPS de E-Commerce usando ferramentas convencionais como CASE e MS Visual Studio em linguagem C#. Uma das contribuições deste trabalho é a utilização de ferramentas convencionais para que engenheiros e organizações de pequenas e médias empresas não especialistas na área possam se aproximar com mais facilidade do paradigma de LPS para desenvolvimento de web. A ferramenta Feature Modeling Tool (FMT) foi desenvolvida pelo grupo, esta permite a modelagem de *features* de uma LPS e gerar a estrutura de pacotes. Uma vez modelada a LPS e os suas *features*, a FMT gera automaticamente todos os pacotes que representam os produtos específicos. Esse processo é mostrado a seguir.

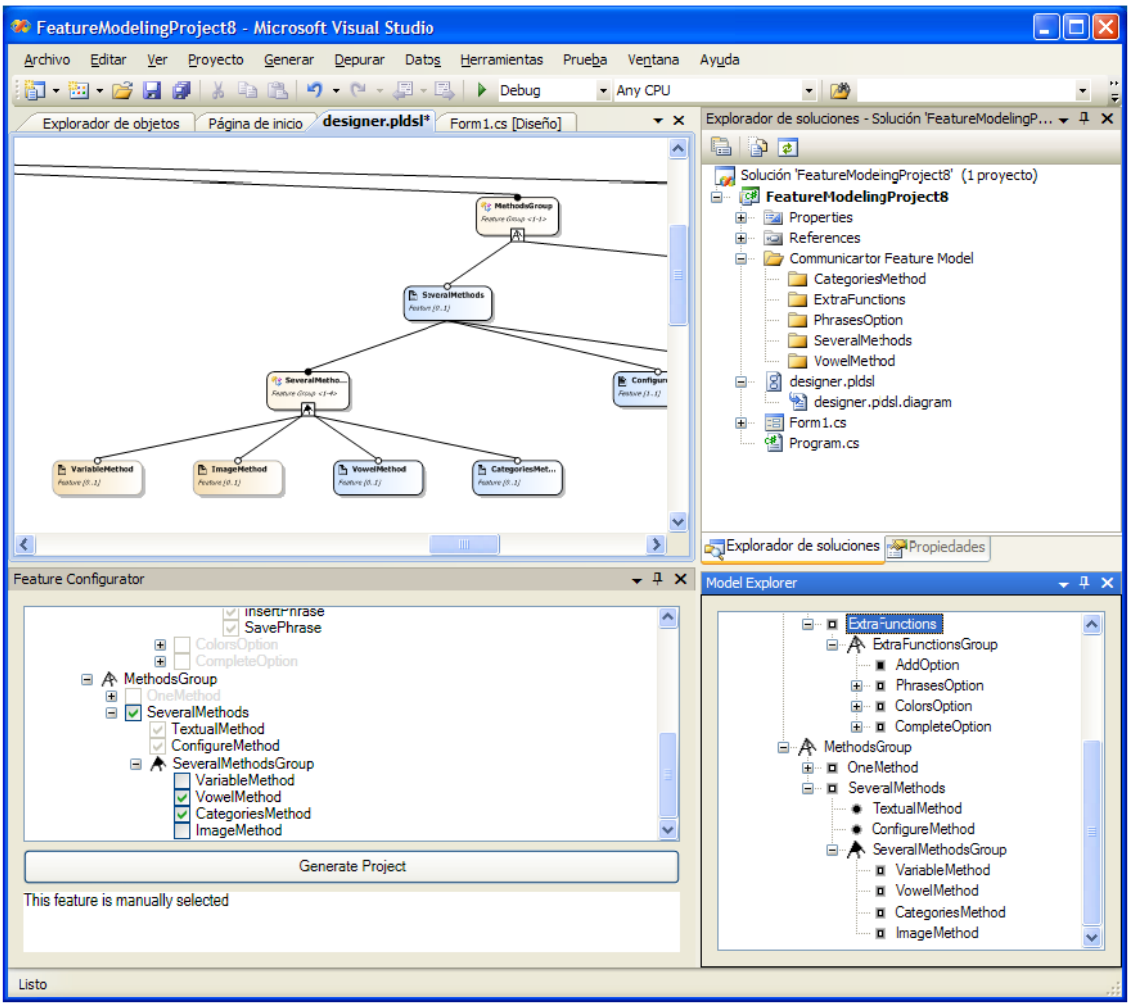

Figura 3- Modelagem de Características de LPS com a FMT (LAGUNA E HERNÁNDEZ (2010)

Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates

(LAU, 2006) faz uma análise de domínio de uma LPS de E-Commerce usando modelos baseados em *features* proposto por (KANG, 1990), porém sem a parte da implementação do modelo.

O objetivo é demonstrar a viabilidade da abordagem num cenário de mundo real. Ademais, com base na experiência desta abordagem, o autor criou um guia de sugestões e recomendações para usuários dessa abordagem. As recomendações incluem: 1) investigar a representação de ordem em modelos de *feature* e em diagramas de atividade, 2) estender semânticas de classes e modelos de diagrama de atividade de um trabalho já existente e 3) adicionar mecanismos que tratam de consistência de toda parte integrada.

Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line Model

(TAWHID e PETRIU, 2008) apresentam uma abordagem para a transformação de modelos UML de LPS para gerar modelos de desempenho para produtos específicos. A entrada para a abordagem proposta é o modelo de origem (*source model*) que é um modelo UML com anotações de desempenho usando o perfil MARTE. O modelo de origem para sistemas e-commerce consiste em: modelo de características representado como pacotes de casos de uso, modelo de casos de uso, modelo de classes e modelo de implantação.

A Figura 4 apresenta o modelo de características para sistemas e-commerce. É possível notar características comuns <<common feature>> e <<kernel>>, alternativas inclusivas <<alternative feature>> e <<alternative>> e opcionais <<optional>>. Por exemplo, Browse Catalog e Confirm Shipment são obrigatórias, enquanto Pay by CredicC e Pay by Check são opcionais.

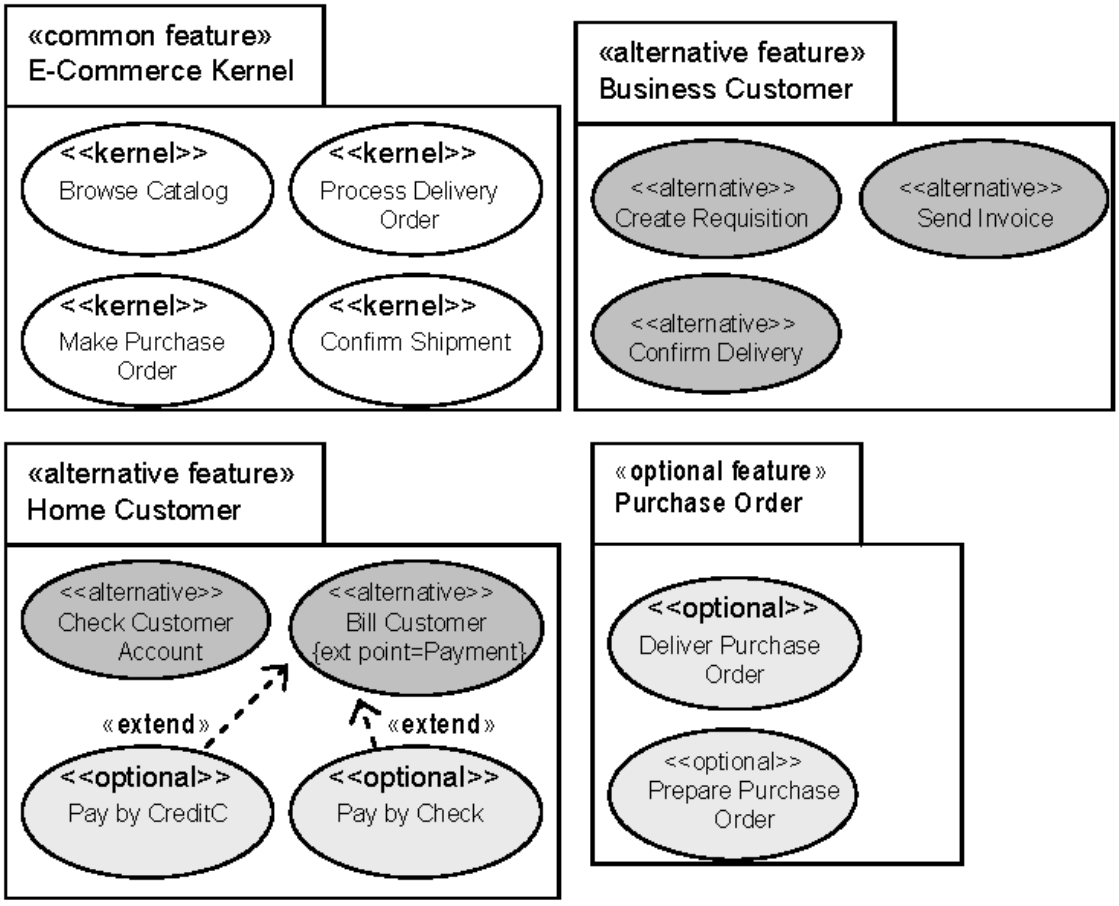

Figura 4 - Características de LPS para Sistemas E-Commerce como Pacotes de Casos de Uso (TAWHID E PETRIU, 2008).

Este trabalho não se mostra relevante para o desenvolvimento de sistemas e-commerce. Porém, sua contribuição está na proposta de modelagem de variabilidade em modelos UML para uma LPS de sistemas e-commerce.

4. Proposta de Desenvolvimento da LPS

Nesta proposta, o desenvolvimento da LPS começa com a análise de domínio onde as informações sobre sistemas, que compartilham um conjunto comum de recursos e dados, são coletadas (KANG, 1990). E de acordo com (POHL, BÖCKLE e LINDEN, 2005), existem três fases para seguir no processo de análise de domínio.

1. Análise de contexto: Definir a extensão (ou limites) de um domínio para análise.
2. Modelagem de domínio: descrever os problemas dentro do domínio que são abordados por software.
3. Modelagem de arquitetura: criar a arquitetura de software que implementa a solução para o problema no domínio.

Nas fases 1 e 2 são estudados as características e os requisitos de um sistema E-Commerce e são definidas as *features* que farão parte da LPS. Em paralelo, é criado um Modelo de *Feature, o* qual representa de forma compacta todos produtos da LPS em termos de *features*.

Terminada a definição do Modelo de *Features* e requisitos documentados, começa-se a fase 3, onde ocorre a implementação de cada uma das *features*.

Em seguida, o mecanismo de variação será implementado. Através da variação, as aplicações com características diferentes serão criadas de forma estática. Passada esta fase, inicia-se o processo de automação de Build, assim gerando aplicações diferentes.

4.1. Análise de Domínio

Uma *feature* é definida como um "aspecto, qualidade ou característica proeminente ou aspecto distintivo visível ao usuário de um sistema" (KANG, 1990]. O foco do desenvolvimento de uma LPS é a criação sistemática e eficiente de programas similares. A análise de domínio orientada a *features* é dedicada à identificação de *features* em um domínio a ser coberto por uma determinada LPS.

As *features* descritas neste capítulo são relacionadas à interface e muitas são diretamente visíveis ao cliente. A Figura 5 ilustra como o modelo de *feature* foi usado para especificar um sistema E-Commerce configurável. O software de cada aplicação é determinado pelas funcionalidades que ele fornece. A *feature* raiz (E-Commerce) identifica a LPS.

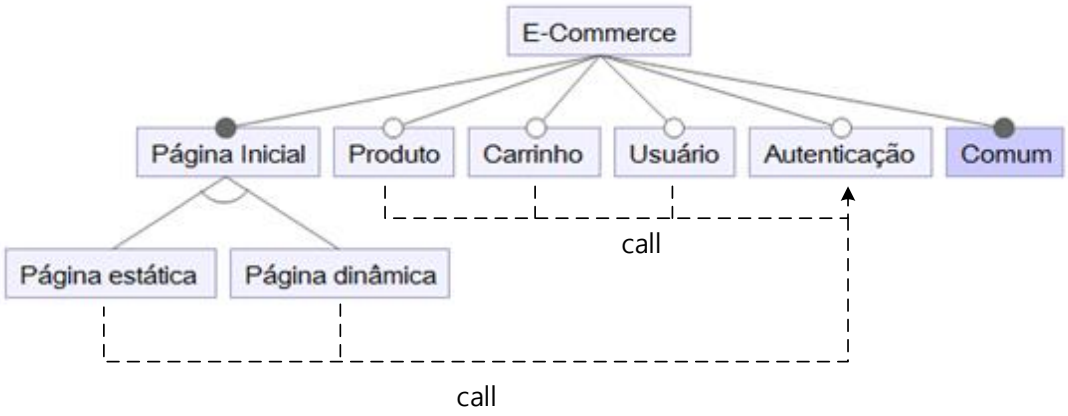

Figura 5 - Árvore de feature

As *features* opcionais são representadas com um circulo vazio, tal como 'Produto', e elas podem ou não fazer parte de uma aplicação final. Por outro lado, as *features* obrigatórias, tais como 'Página Inicial' são representadas por círculos cheios. Features alternativas podem ser exclusivas (XOR) ou não exclusivas. A primeira indica que apenas uma sub-caracteristica pode ser selecionada a partir das alternativas. Por exemplo, "Página estática" e "Página dinâmica" são *features* alternativas para a "Página inicial". A segunda indica que uma ou mais características podem ser selecionada, tais como "Produto", "Carrinho", "Usuário" que permitem a seleção de uma opção em uma aplicação.

Nota-se também que foi utilizada uma linha pontilhada com a seta apontada para a *feature* "Autenticação" para representar uma relação de inclusão de *feature*. Nesse contexto, a *feature* Autenticação deve ser incluída e chamada antes de invocar qualquer método em outras *features* que possui a esta relação. É bom salientar que esta notação é proposta por este trabalho pois as notações existentes não apresentavam a semântica necessária.

4.1.1. Página Inicial

Toda loja eletrônica possui uma página inicial. É a primeira página que um cliente encontra quando o site é acessado através de endereço *Uniform Resource Locator* (URL). E também a loja eletrônica pode ser configurada para redirecionar à página inicial se o URL pelo qual está tentando acessar estiver apontando para uma sessão expirada, ou ser uma página inválida ou restrita. O conteúdo primário da página inicial consiste de uma mensagem de boas-vindas e produtos em destaque. Como mostrada na Figura 6, a página pode ser uma combinação de conteúdo estático e dinâmico. A página é classificada como dinâmica se algum elemento da página é gerado automaticamente.

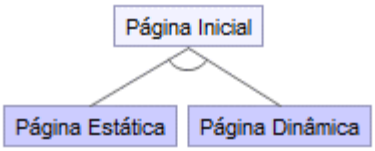

Figura 6 - Mapeamento da feature: “Página Inicial”

- Conteúdo estático: é adequado para conteúdo que não muda ou muda com pouca frequência.
- Conteúdo dinâmico: é ideal para conteúdo que muda com frequência. Geralmente o conteúdo é gerado sob demanda pelo servidor.

4.1.2. Produto

A *feature* Produto possui como uma *feature* opcional. A informação sobre produtos exibida na Figura 7 mostra todos atributos que descrevem as características de um produto.

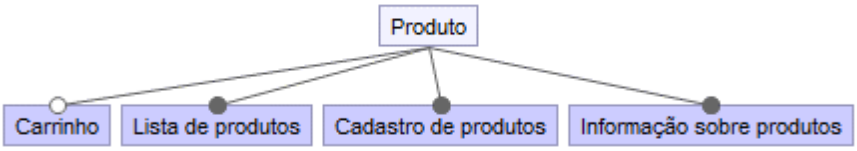

Figura 7 - Mapeamento da feature: "Produto"

- Lista de produtos: é uma página web que descreve detalhadamente sobre um único produto. Normalmente a informação é renderizada na forma textual associado com ativos armazenados. E existe um link para adicionar o item no carrinho de compras. Todos produtos possuem sua própria página.

4.1.2.1. Informação sobre produtos

A informação sobre produtos exibida na Figura 8 mostra todos atributos que descrevem as características de um produto.

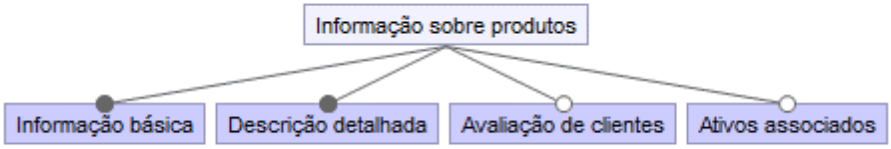

Figura 8 - Mapeamento da feature: "Informação sobre produtos"

Informação básica

É um conjunto mínimo de informações que caracterizam um produto. Os elementos mais básicos são “nome do produto” e “identificador único”.

Descrição detalhada

Fornece informações adicionais sobre produtos ou qualquer detalhe que seja relevante aos clientes. Esta descrição não possui uma estrutura específica, sendo, normalmente, um campo de texto comum. A descrição detalhada é exibida na página do produto.

Avaliação de clientes

A avaliação permite que clientes compartilhem suas opiniões sobre produtos atribuindo pontos e/ou registrando comentários. Essa informação é mostrada na página do produto. O tipo de dado a ser usado para representar a avaliação é um valor numérico numa escala pré-definida. O formato de comentários é um campo de texto livre. Por motivos de segurança e interatividade, a função de avaliação deve ser restrita apenas aos usuários cadastrados.

Ativos associados

É um conjunto de arquivos que descrevem ou tem propósito ilustrativo do produto. Estes são usados na página do produto junto com as outras informações textuais. Estes arquivos são imagens e podem ter quantidade e formatos diferentes de um produto para outro.

- Imagens *thumbnail*: são imagens de tamanho pequeno para ser carregado rapidamente e geralmente são utilizados na lista de produtos para manter o espaço na página.
- Galeria de imagens: contém um conjunto de imagens.

4.1.3. Carrinho de compras

O carrinho de compras mantém a informação de itens que clientes desejam comprar durante a sessão. A *feature* possui uma página que mostra o conteúdo do carrinho e as funções como a adição e remoção de itens (Figura 9).

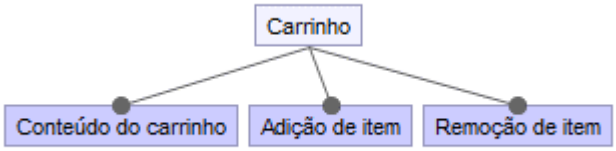

Figura 9 - Mapeamento da feature: "Carrinho de compras"

- Página de conteúdo do carrinho de compras: O conteúdo da página permite o cliente visualizar todos itens que foram colocados no carrinho. Essa página também permite que clientes modifiquem a quantidade de itens ou removam os itens do carrinho de compras. Cada produto é listado com a quantidade e o subtotal. E o valor total da mercadoria e o link para fechamento de pedido é mostrado na página.

4.1.4. Usuário

Esta *feature* é referente ao cadastro de novos usuários (Figura 10). O cadastro permite que as informações de clientes sejam solicitadas e persistidas. Assim, os clientes não precisam entrar com suas informações toda vez que efetivar a compra. Além disso, a informação pode servir para criar estratégias de marketing segmentado onde o objetivo é direcionar a visibilidade de produtos especificamente para o seu público alvo.

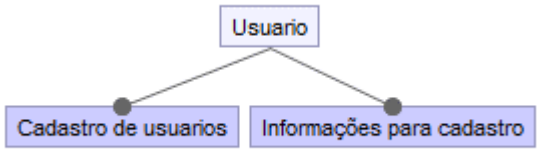

Figura 10 - Mapeamento da feature: "Usuário"

4.1.4.1. Informações para cadastro

O cliente deve fornecer informações pessoais e esta informação é armazenada. Um perfil de cliente possui os seguintes campos (a maioria dos campos é opcional): credencial de *login*, endereço, informação de cartão de crédito e informação pessoal. O campo obrigatório é a credencial de *login*, o qual o cliente usa para se identificar no sistema. A credencial inclui um identificador único, como um endereço de e-mail e a senha. Outro campo é opcional:

- Endereço de entrega: especifica o endereço para qual a encomenda será entregue. Foi considerado que o sistema tem suporte a um único endereço de entrega.

4.1.5. Autenticação

Com a autenticação habilitada, algumas ações do sistema são restritas para usuários não cadastrados. Abaixo, apresentam-se duas políticas que podem ser adotadas: visualização com cadastro e visualização sem restrição. As funções referentes à autenticação são: login, logout, autenticar (Figura 11).

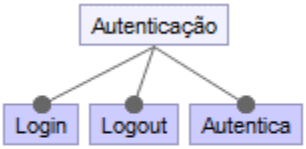

Figura 11 - Mapeamento da feature: "Autenticação"

4.1.6. Comum

A *feature* Comum está associada ao acesso ao banco de dados e o corpo de HTML no qual serão renderizadas as diferentes páginas (Figura 12).

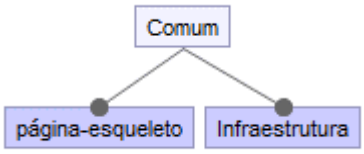

Figura 12 - Mapeamento da feature: "Comum"

4.1.7. Compras

Compras é um grupo de *features* relacionadas ao *workflow* de compra de produtos. Começa com o processo de colocar produtos no carrinho e termina com a realização de pedido. Este consiste de três *features* obrigatórios: carrinho de compras, fechamento de pedido e confirmação de pedido (Figura 13).

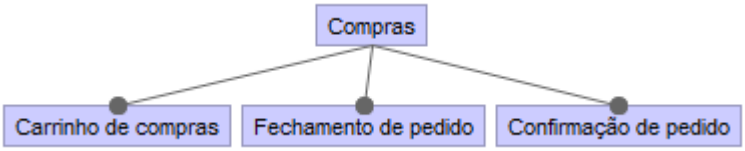

Figura 13 - Mapeamento da feature: "Compras"

Este processo inicia quando o cliente encerra o pedido de compra e segue para a etapa de pagamento. Então, o cliente entra com uma das opções de pagamento disponíveis e informa sobre a entrega. Por fim, encerra-se o processo quando o cliente confirma o pedido. A Figura 13 mostra o seu modelo de *features*.

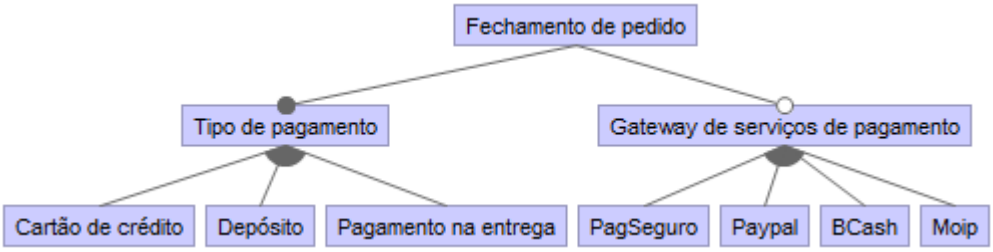

Figura 14 - Mapeamento da feature: "Fechamento de pedido"

Um dos tipos de pagamento deve ser selecionado para que o sistema processe o pagamento e efetivar a venda.

- Tipos de pagamento: forma de pagamento que será disponibilizado no sistema como cartão de crédito, depósito na conta, pagamento na entrega.
- Gateways de serviços de pagamento: permite o uso de serviço de intermediação de pagamentos oferecidos por empresas especializadas em gerenciar informações de pagamentos, verificação de fraude e estabelecer acordos com a instituições financeiras. As empresas mencionadas acima são: PagSeguro, Paypal entre outras.

4.1.8. Confirmação de pedido

Fornece uma confirmação ao cliente enviando uma mensagem como: “pedido recebido com sucesso”. Então um número do pedido é gerado e informado ao cliente. Esta *feature* é obrigatória porque os clientes aguardam um feedback do sistema depois de ter submetido um pedido; caso contrário eles podem achar que o pedido não foi processado e, então, submeter um outro pedido. A confirmação de pedido pode ser mostrada em uma página do sistema ou enviada por e-mail ou notificada por telefone (Figura 15).

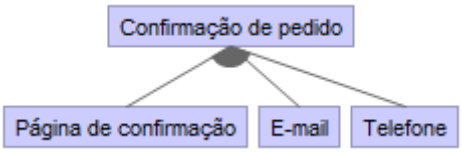

Figura 15 - Mapeamento da feature: "Confirmação de pedido"

- Página de confirmação: uma página será exibida imediatamente depois do processo de fechamento de pedido ser concluído
- E-mail de confirmação: Um e-mail de confirmação pode ser enviado imediatamente depois do processo de fechamento de pedido ser concluído.
- Telefone: uma ligação é feita no número do cliente após concluir o processo de fechamento de pedido. É útil quando se trata de transação de alto valor agregado.

4.2. Arquitetura da LPS de E-Commerce

O modelo de arquitetura é dividido em três partes: a construção de *features*, onde serão implementadas as *features*, o mecanismo de variação estática, e, por último, a instanciação de produtos através da combinação de artefatos. A Figura 17 ilustra a arquitetura proposta.

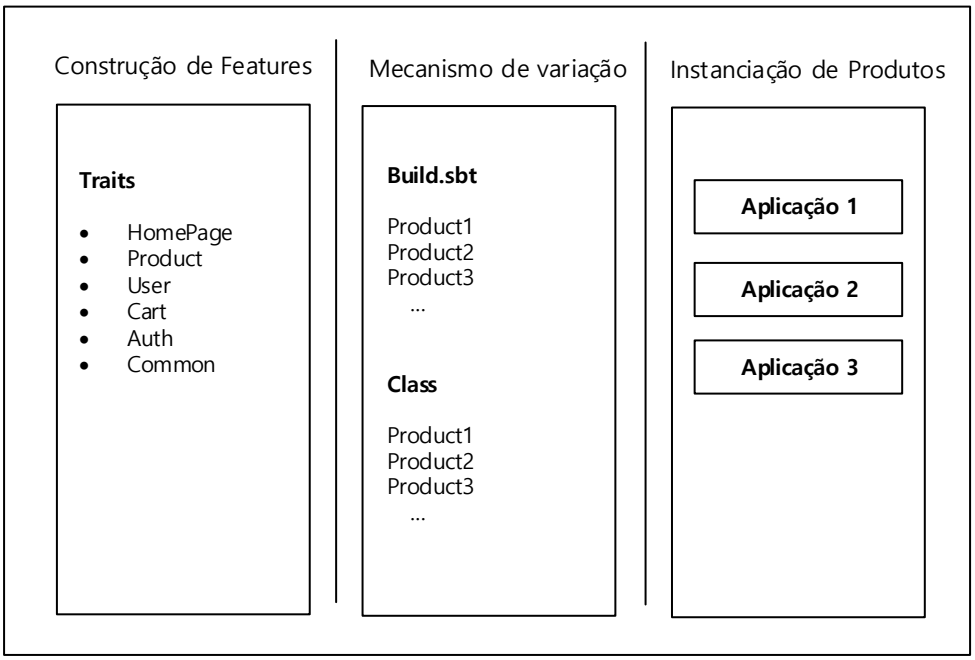

Figura 16- Arquitetura da LPS de E-Commerce

4.2.1. Construção de *Features*

As *features* capturados na fase de análise de domínio foram implementados usando um recurso da linguagem chamado *trait*. O conceito *trait* é similar a uma interface em Java e é utilizado para definir tipos de objetos especificando somente as assinaturas dos métodos suportados. Como em Java 8, Scala permite que *traits* sejam parcialmente implementadas. Isso significa que é possível definir uma implementação padrão para alguns métodos. Dessa forma, esse recurso foi utilizado para implementação de métodos das *features*. O mapeamento entre *features* e *trait* é ilustrado na Figura 18.

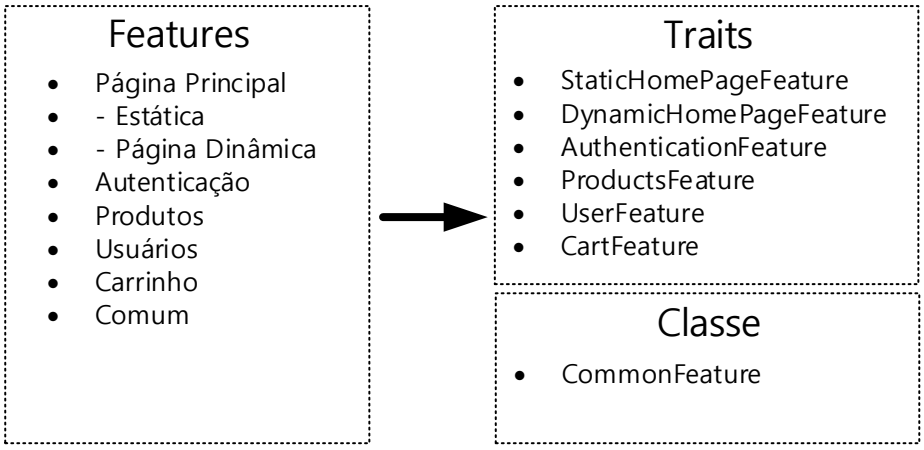

Figura 17 - Relação entre Feature e Trait

A relação de métodos implementados em cada *Trait* é descrita abaixo:

StaticHomePage

- *Homepage*: renderiza a página principal de conteúdo estático.

DynamicHomePage

- *Homepage*: renderiza a página principal de conteúdo dinâmico.

Authentication

- *Login*: renderiza o página de login.
- *Logout*: processa o logout do sistema.
- *Authenticate*: processa a autenticação a partir de dados recebidos da página do *login.*

Product

- *productList*: renderiza a página com lista de produtos,
- *itemDetails*: renderiza detalhes de um produto.

Cart

- *showCart* : mostra o conteúdo do carrinho..
- *addItemToCart*: adiciona o item ao carrinho.
- *removeItemFromCart*: remove o item do carrinho

User

- *signUp*: renderiza o formulário de cadastro de um novo usuário.
- *signUpSuccess*: renderiza uma página com a mensagem de cadastro com sucesso.
- *addUser*: adiciona um novo usuário na base da informação recebida do formulário de cadastro.

A *feature* Comum é organizada em três camadas:

- Infraestrutura: é a camada onde será colocada toda a implementação referente à persistência de dados.
- Domínio: é a camada mais importante. Nela deve conter toda regra de negócio do sistema e deve ser implementada primeiramente. Então, para completar toda a lógica da camada, será necessária alguma interface ou API da camada de infraestrutura. O melhor tipo de teste para esta camada é o teste de unidade porque não há envolvimento com a interface. A regra de negócio será implementada dentro de controlador de Play framework.
- Interface e aplicação: é onde a lógica da camada de Domínio será representada aos usuários do sistema. Para essa camada, o tipo de teste mais adequado é o teste funcional e de integração.

Um aspecto importante para projetar a aplicação no Play é definir o esquema para requisições de HTTP. Isso é definido no arquivo de configuração de rota e a camada de controlador implementa essa interface.

Mais especificamente, controladores são classes de Scala que definem a interface HTTP da aplicação, e a configuração de rota determina qual método controlador será chamado com a dada requisição HTTP. Esses métodos de controladores são chamados de ações e nesse contexto o controlador é uma coleção de métodos de ação, como mostrado na Figura 19.

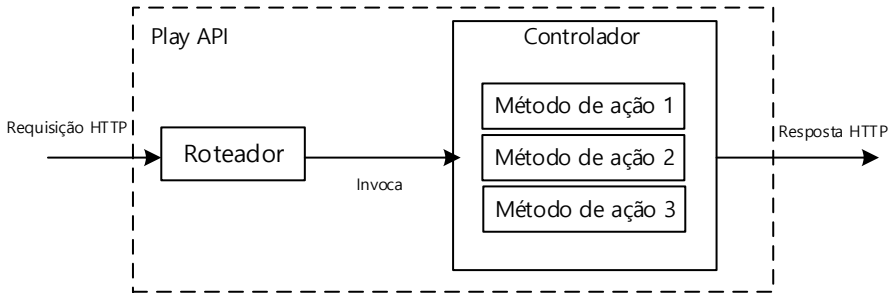

Figura 18 - Arquitetura do Play Framework

No contexto deste trabalho as classes de controladores foram implementadas usando *Trait*. Assim cada *Trait* implementa métodos da camada de controlador que gerencia a requisição de HTTP. O conjunto de requisições é definido no arquivo de configuração de rota. Isso quer dizer que cada rota definida está mapeada como um método da camada de controlador.

```
#Home
GET     /                           controllers.products.Product1.homePage

#Product
GET     /productlist                controllers.products.Product1.productList(p:Int ?=0, s:Int ?= 2, f ?= "")
GET     /productlist/:id            controllers.products.Product1.itemDetails(id: Long)
```

Uma rota é composta por um método HTTP, URL e a ação do controlador. A primeira rota é definida com o GET e URL raiz “/” que está mapeada com o método *homePage*. A segunda rota especifica o método GET e está associada com o endereço /productlist. E a terceira rota está mapeada com um endereço com id do produto.

4.2.2. Mecanismo de variação

Feita a construção de *features*, o próximo passo é implementar o mecanismo de variação usando a ferramenta SBT. Neste trabalho foi implementada uma forma de variação estática através da criação de classes sendo que cada uma das classes implementa um conjunto possível combinações de *features* diferentes.

O mecanismo principal para executar a variação é o arquivo de *build* do projeto raiz da LPS. Para cada aplicação é definida um projeto no contexto da SBT, isso significa que cada projeto deve estar mapeado a um diretório onde está definida a classe que implementa as *features* daquela aplicação especifica. Segue abaixo um exemplo de definição simples de um projeto “root” no diretório raiz.

```
lazy val root = (project in file("."))
```

É importante mencionar que um diretório não pode ser compartilhado por mais de um projeto, isso quer dizer que para cada aplicação da LPS é necessário associar uma pasta específica. Segue abaixo o exemplo da definição do projeto product1 especificado no arquivo *build*.

```
// StaticHomePage, Product, Cart, Authentication, User
lazy val product1 = (project in file("prod/product1")).enablePlugins(PlayScala)
  .aggregate(features)
  .dependsOn(features).settings(
  scalaVersion := "2.11.8",
  name := "product1",
  PlayKeys.devSettings ++=
    Seq(("play.http.router", "product1.Routes"),
      ("db.default.driver", "org.h2.Driver"),
      ("db.default.url", "jdbc:h2:mem:play"),
      ("play.evolutions.enabled", "true"))
)
```

Figura 19 - Definição da aplicação Product 1

Com o projeto definido no *build*, uma classe que diz sobre quais as *features* serão combinadas é criada dentro do diretório do projeto. O código da Figura 21 mostra a definição da classe Product1 que implementa as *features* como Página inicial estático, Produto, Carrinho, Autenticação e Usuário.

```
class Product1 @Inject() (
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  override val messagesApiInj: MessagesApi
)
    extends CommonFeature(userServiceInj, productServiceInj, categoryService, messagesApiInj,
      Seq("StaticHomePage", "Product", "Cart", "Auth", "User"))
    with StaticHomePageFeature
    with ProductFeature
    with CartFeature
    with AuthenticationFeature
    with UserFeature {
```

Figura 20 - Definição da classe do Product 1

4.2.3. Instanciação de Produtos

Cada uma das combinações de *features* que forma uma aplicação completa da LPS é definida usando uma classe. Cada uma dessas classes implementa um conjunto de *features* que atende a necessidade do cliente. Um exemplo de definição de uma classe é exemplificada abaixo:

```
class Product
    extends CommonFeature
    with DynamicHomePageFeature
    with ProductListFeature
    with CartFeature
    with AuthenticationFeature
    with UserFeature {
}
```

Nesse código, uma aplicação de LPS é composta pelas *features*: página dinâmica, produto, carrinho, autenticação e usuário. Dessa forma, quando o usuário iniciar essa aplicação de exemplo, o Play irá gerar uma classe Routes que possui um construtor como segue:

```
class Routes(
  Product1_1: controllers.products.Product1
) extends GeneratedRouter { . . . }
```

Com a classe gerada, quando o projeto for inicializado, o Play irá instanciar um objeto desta classe e carregar o '*carregador de aplicação'.* Um exemplo simplificado é apresentado abaixo:

```
class MyApplicationLoader (
  context: Context
) extends ApplicationLoader {

  def load(context: Context) = {
    new MyComponentsContext(context).application
  }
}

class MyComponents (
  context: Context
) extends BuiltInComponentFromContext(context) {

   lazy val router = new Routes(product1Controller)
   lazy val product1Controller = new Product1
}
```

5. Exemplos de Aplicações da LPS

Para ilustrar a instanciação das aplicações serão considerados três produtos a partir da LPS proposta no capítulo 4. Para cada uma das aplicações são mostrados o arquivo de configuração e algumas telas.

A primeira aplicação será feita misturando todas as *features* implementadas junto com a página inicial de conteúdo estático. A segunda e a terceira serão geradas variando algumas *features* e assim mostrar como as diferentes versões podem ser criadas.

A execução do projeto é iniciada pelo comando run “nome_do_produto” na linha de comando da SBT. A variável especificada seguido do comando run é o nome do projeto definido como lazy val no arquivo de configuração de build.

5.1. Primeira Aplicação

A primeira aplicação da LPS tem como base algumas características comuns em uma aplicação E-Commerce tais como a capacidade de listar produtos e manter as informações manipuladas, que é uma característica básica e serve como base para as demais funcionalidades. A árvore de *feature* é ilustrada na Figura 22.

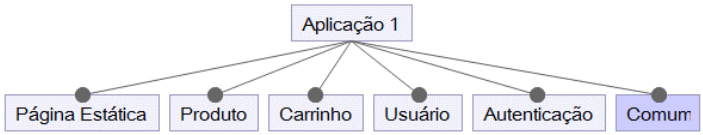

Figura 21 - Modelo de árvore da Aplicação 1

Essa é uma aplicação que compõe as *features* como “página estática”, “produto”, “carrinho”, “autenticação” e “usuário”. Uma característica importante da aplicação é que todo conteúdo do site é restritivo, isto é, o usuário deve efetuar o login no sistema para visualizar o conteúdo do site. A definição da classe que compõe as features é mostrada na Figura 23.

```
class Product1 @Inject() (
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  override val messagesApiInj: MessagesApi
)
    extends CommonFeature(userServiceInj, productServiceInj, categoryService, messagesApiInj,
      Seq("StaticHomePage", "Product", "Cart", "Auth", "User"))
    with StaticHomePageFeature
    with ProductFeature
    with CartFeature
    with AuthenticationFeature
    with UserFeature {
```

Figura 22 - Definição da classe Product1

A Figura 24 mostra mostra a configuração de rota da primeira aplicação.

```
#Home
GET     /                           controllers.products.Product1.homePage

#Product
GET     /productlist                controllers.products.Product1.productList(p:Int ?=0, s:Int ?= 2, f ?= "")
GET     /productlist/:id            controllers.products.Product1.itemDetails(id: Long)

#Cart
GET     /cart                       controllers.products.Product1.showCart
GET     /cart/remove/:id            controllers.products.Product1.removeItemFromCart(id:String)
GET     /cart/add/:id               controllers.products.Product1.addItemToCart(id: Long)

#Authentication
GET     /login                      controllers.products.Product1.login
GET     /logout                     controllers.products.Product1.logout
GET     /authenticate               controllers.products.Product1.authenticate

#User
GET     /signup                     controllers.products.Product1.signUp
POST    /signup                     controllers.products.Product1.addUser
GET     /signup/success             controllers.products.Product1.signUpSuccess
```

Figura 23 - Rota da Aplicação 1

5.1.1. Telas da Primeira Aplicação

O comportamento esperado desta aplicação quando a URL raiz é acessada é redirecionar à página de login caso o usuário ainda não tenha se logado. Qualquer tentativa de acesso a outras páginas resultará em redirecionamento para página de login.

A Figura 25 ilustra o acesso ao endereço raiz e o seu redirecionamento à URL /login.

Figura 24 - Acesso ao endereço raiz

5.2. Segunda Aplicação

Essa instância da LPS é criada sem a função de autenticação, desse modo o site pode ser acessado por qualquer cliente não cadastrado O modelo de árvore e o resultado do acesso ao endereço raiz é mostrado na Figura 26.

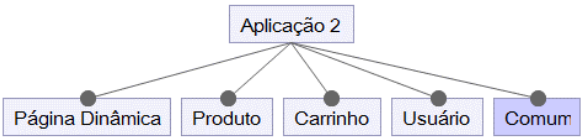

Figura 25 - Modelo de árvore da Aplicação 2

5.2.1. Telas da Segunda Aplicação

Outra característica que diferencia a segunda aplicação da primeira é que esta renderiza uma página inicial de conteúdo dinâmico. Assim, os itens de produtos são consultados e buscados do banco de dados para serem renderizados como ilustrado na Figura 27.

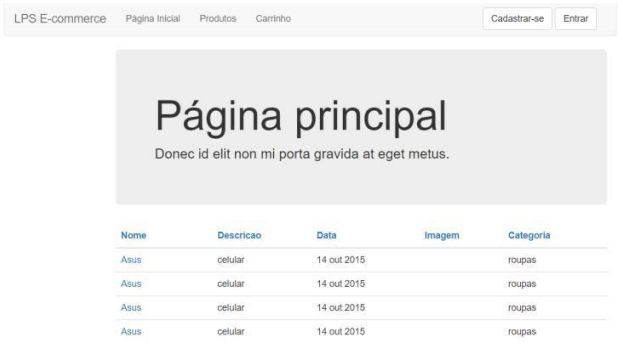

Figura 26 - Página principal da Aplicação 2

A página de detalhes do produto é acessado clicando no nome do produto na lista de produtos. Assim o Play renderizará os detalhes do produto, conforme mostrado na Figura 28.

Figura 27 - Detalhes do item

Para adicionar o item ao carrinho o usuário deve clicar no link “Adicionar ao carrinho” que aparece na página de detalhes ou o item também pode ser adicionado diretamente por um URL: */cart/add/7*. A Figura 29 mostra o carrinho de compras.

Figura 28 – Carrinho

5.3. Terceira Aplicação

A terceira aplicação é uma versão da LPS que apresenta somente uma página principal que renderiza um conteúdo dinâmico, buscando a informação do banco de dados (Figura 30).

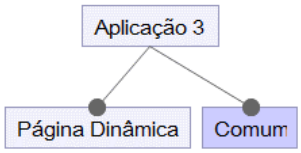

Figura 29 - Modelo de árvore da Aplicação 3

A Figura 31 mostra o resultado do acesso à página raiz dessa aplicação que renderiza uma página estática.

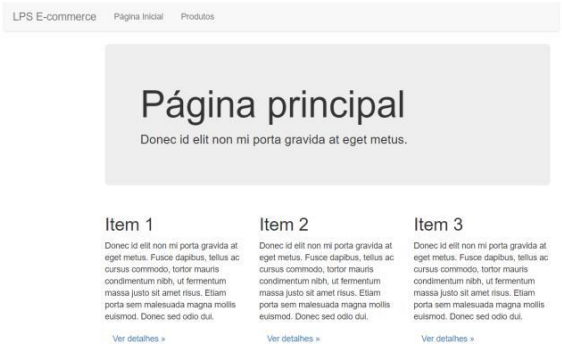

Figura 30 - Página principal da aplicação 3

Neste capitulo foram mostradas três versões diferentes da LPS com o objetivo de mostrar a capacidade de variabilidade permite que sejam lançadas versões estáveis e funcionais da LPS a cada nova aplicação, facilitando a manutenção e evolução das funções que são desenvolvidas.

6. Conclusão e Trabalhos Futuros

Este trabalho propôs um protótipo de arquitetura de LPS utilizando o Play Framework no contexto de um sistema de comércio eletrônico. A proposta é composta por três atividades de desenvolvimento (análise de requisitos, arquitetura e codificação). Na fase da codificação foi mostrado como as *features* capturadas na análise de domínio foram implementadas usando *trait* em Scala como uma forma de mecanismo de variabilidade.

A abordagem foi exemplificada e validada por meio três aplicações simples. O trabalho também descreveu as tecnologias e ferramentas que foram utilizadas durante o desenvolvimento, como Play Framework, Scala e SBT.

As seguintes contribuições são resultados diretos desse trabalho:

(i) Proposta de um protótipo de uma arquitetura de LPS de E-Commerce

A implementação da proposta foi realizada a partir da análise de domínio do problema, em seguida foi feita a codificação das *features* capturadas na fase anterior e, por último, o mecanismo de variabilidade estática através da criação de classes.

(ii) Definição de uma notação no modelo de *feature*

Durante a fase da análise de domínio foi introduzida uma notação que representa a inclusão de chamada de métodos na *feature* chamadora.

Como trabalho futuro, novas *features* poderão ser adicionadas com o objetivo de enriquecer esse estudo com uma visão mais ampla. Também seria interessante buscar uma forma viável de instanciar um produto sem especificar previamente todas as combinações possíveis em arquivo Build, tornando o processo de geração da aplicação mais dinâmica.

Por fim, observa-se que esse trabalho é um passo inicial na busca de novas formas de implementação de LPS usando o Play Framework e para avaliação de novos estudos, podendo ser comparado em outros contextos do mundo real.

7. Referências

ALBERTIN, A. L. **Comércio Eletrônico:** Modelo, Aspectos, e Contribuições de sua Aplicação. 5. ed. [S.l.]: Editora Atlas, 2004.

BOOCH, G. **Object-oriented design with applications**. [S.l.]: Benjamin/Cummings, 1991.

CLEMENTS, P.; NORTHROP, L. **Software Product Lines:** Practices and Patterns. 3. ed. [S.l.]: Addison-Wesley Prossional, 2001.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Formalizing cardinality-based feature models and their specialization, In Software Process Improvement and Practice, 2005.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Staged configuration through specialization and multi-level configuration of feature models, In Software Process Improvement and Practice, 2005.

CZARNECKI, K.; KIM, C. H. P. Cardinality-based feature modeling and constraints: a progress report, In Proc. International Workshop on Software Factories, 2005.

DEURSEN, A. V.; KLINT, P. Domain-specific language design requires feature. **Journal of Computing and Information Technology**, 2001.

HILTON, P.; BAKKER, E.; CANEDO, F. **Play for Scala**. [S.l.]: Mannings Publications Co, 2013.

HINOJOSA, D. **Testing in Scala**. [S.l.]: O'Reilly Media, Inc., 2013.

J LEE, D. M. M. N. A Feature-oriented Approach for Developing Reusable Product Line Assets of Service-based Systems. **The Journal of Systems and Software**, 2010.

KANG, K. C.. C. S. G.. H. J. A.. N. W.. P. A. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. [S.l.]: Software Engineering Institute: Carnegie Mellon University. 1990.

KOSKELA, L. **Effective Unit Testing**. [S.l.]: Manning, 2013.

LAU, S. Q. **Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates**. Ontario: University of Waterloo, 2006.

LINDEN, F. J. V. D.; SHMID, K.; ROMMES, E. **Software Product Lines in Action**. 1. ed. [S.l.]: Springer, 2007.

MIGUEL A. LAGUNA, C. H. **A Software Product Line Approach for E-Commerce Systems**. IEEE International Conference on E-Business Engineearing. [S.l.]: [s.n.]. 2010.

NEVES, G. S.; VILAIN, P. **Test Logic Reuse Through Unit Test Patterns**. IEEE International Conference on Information Reuse and Integration. San Francisco: [s.n.]. 2014.

ODERSKY, M.; SPOON, L.; VENNERS, B. **Programming in Scala**. [S.l.]: Artima, 2008.

POHL, K.; BÖCKLE, G.; LINDEN, F. V. D. **Software Product Line Engineearing**. [S.l.]: Springer, 2005.

SAXENA, S. **Getting Started with SBT for Scala**. [S.l.]: Packt Publishing Ltd , 2013.

TAWHID, R.; PETRIU, D. C. **Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line Model**. International Workshop on Software and Performance. [S.l.]: [s.n.]. 2008.

THAKER, S. E. A. Safe composition of product lines. **Generative Programming and Component Engineering**, 2008.

APÊNDICE A – Artigo

Arquitetura para Linhas de Produto de Software de E-Commerce usando Play Framework

Yun Hu Lee

Universidade Federal de Santa Catarina

Florianópolis, BR

Resumo. A linha de produto de software baseia-se em um planejamento de reutilização de peças já desenvolvidas em novos projetos automobilísticos, sem a necessidade de reprojetar toda a plataforma de criação. Isso traz, além da questão econômica, um aumento do público alvo e melhora na qualidade das peças, já que essas, por serem utilizadas novamente, são revistas com mais frequência, proporcionando uma confiabilidade maior. O trabalho apresenta uma arquitetura de LPS utilizando o Play Framework no contexto de um sistema de comércio eletrônico. A proposta é composta por três atividades de desenvolvimento (análise de requisitos, arquitetura e codificação). A abordagem foi exemplificada e validada por meio três aplicações a partir da LPS construída.

Introdução

Sistema de E-Commerce pode ser considerado como uma aplicação de web derivado de um conjunto de recursos reusáveis e de artefatos os quais capturam abstrações específicas no domínio como por exemplo: carrinho de compras, cadastro de usuário, forma de pagamento, em um sistema de e-commerce. Este trabalho apresenta os seguintes objetivos que devem ser alcançados para satisfazer o escopo deste trabalho.

1. Análise de domínio de Sistemas E-Commerce: tem como objetivo modelar domínio dos Sistemas E-Commerce, identificação das características comuns e variáveis entre os sistemas E-Commerce.

2. Escolha de ferramentas e metodologias que serão utilizadas no processo de desenvolvimento da LPS.

3. Desenvolvimento da arquitetura do domínio e a construção dos componentes correspondentes às *features*.

4. Desenvolvimento de produtos da LPS através do reuso sistemático dos artefatos do repositório de artefatos comuns.

Proposta de Desenvolvimento da LPS

Nesta proposta, o desenvolvimento da LPS começa com a análise de domínio onde as informações sobre sistemas, que compartilham um conjunto comum de recursos e dados, são coletadas (KANG, 1990). E de acordo com (POHL, BÖCKLE e LINDEN, 2005), existem três fases para seguir no processo de análise de domínio.

4. Análise de contexto: Definir a extensão (ou limites) de um domínio para análise.
5. Modelagem de domínio: descrever os problemas dentro do domínio que são abordados por software.
6. Modelagem de arquitetura: criar a arquitetura de software que implementa a solução para o problema no domínio.

Nas fases 1 e 2 são estudados as características e os requisitos de um sistema E-Commerce e são definidas as *features* que farão parte da LPS. Em paralelo, é criado um Modelo de *Feature, o* qual representa de forma compacta todos produtos da LPS em termos de *features*.

Terminada a definição do Modelo de *Features* e requisitos documentados, começa-se a fase 3, onde ocorre a implementação de cada uma das *features*.

Em seguida, o mecanismo de variação será implementado. Através da variação, as aplicações com características diferentes serão criadas de forma estática. Passada esta fase, inicia-se o processo de automação de Build, assim gerando aplicações diferentes.

Análise de domínio

Uma *feature* é definida como um "aspecto, qualidade ou característica proeminente ou aspecto distintivo visível ao usuário de um sistema" (KANG, 1990]. O foco do desenvolvimento de uma LPS é a criação sistemática e eficiente de programas similares. A análise de domínio orientada a *features* é dedicada à identificação de *features* em um domínio a ser coberto por uma determinada LPS.

As *features* descritas neste capítulo são relacionadas à interface e muitas são diretamente visíveis ao cliente. A Figura abaixo ilustra como o modelo de *feature* foi usado para especificar um sistema E-Commerce configurável. O software de cada aplicação é determinado pelas funcionalidades que ele fornece. A *feature* raiz (E-Commerce) identifica a LPS.

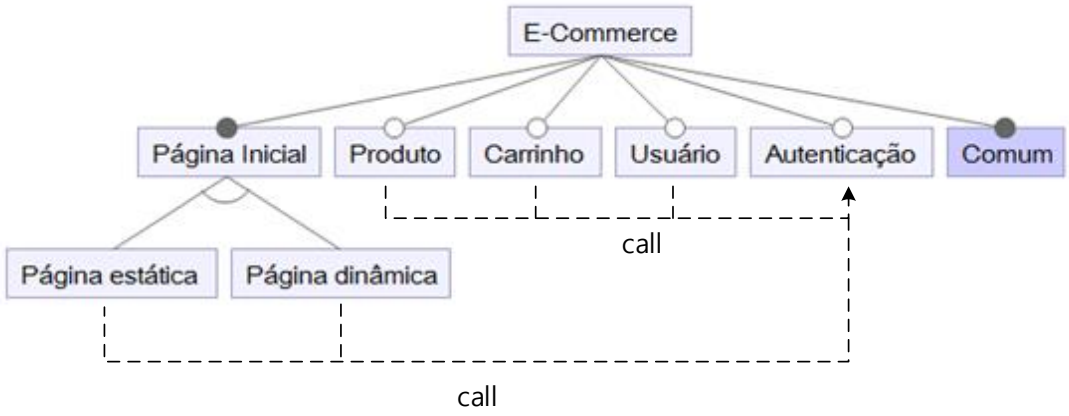

Fig 1 - Modelo de feature

Arquitetura da LPS

O modelo de arquitetura é dividido em três partes: a construção de *features*, onde serão implementadas as *features*, o mecanismo de variação estática, e, por último, a instanciação de produtos através da combinação de artefatos. A Figura 17 ilustra a arquitetura proposta.

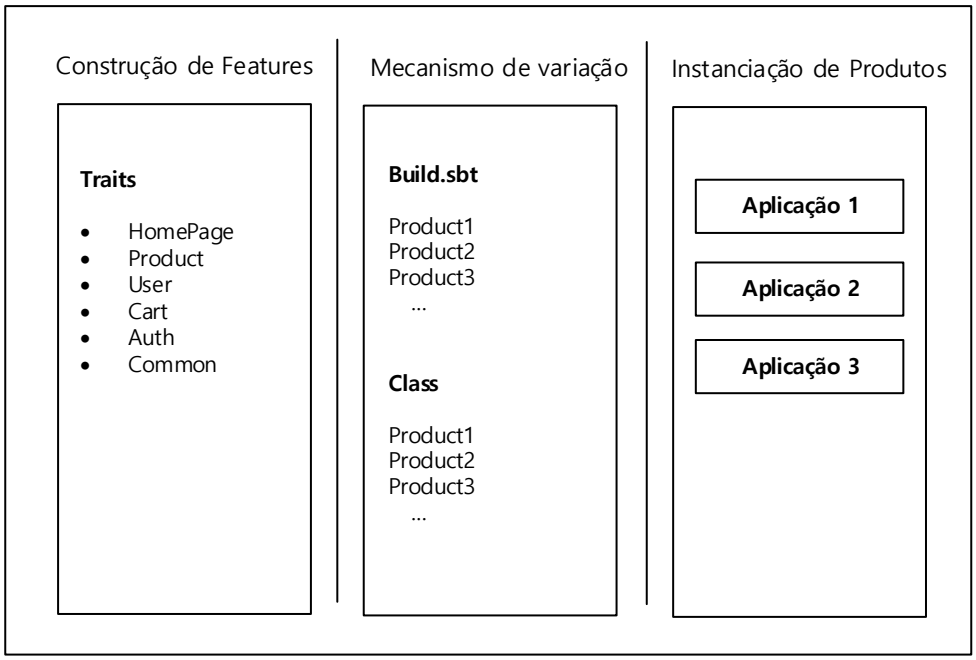

Fig 2 - Arquiteturax

Os principais métodos implementados em cada traits são listados abaixo:

StaticHomePage

- *Homepage*: renderiza a página principal de conteúdo estático.

DynamicHomePage

- *Homepage*: renderiza a página principal de conteúdo dinâmico.

Authentication

- *Login*: renderiza o página de login.
- *Logout*: processa o logout do sistema.
- *Authenticate*: processa a autenticação a partir de dados recebidos da página do *login.*

Product

- *productList*: renderiza a página com lista de produtos,
- *itemDetails*: renderiza detalhes de um produto.

Cart

- *showCart* : mostra o conteúdo do carrinho..
- *addItemToCart*: adiciona o item ao carrinho.
- *removeItemFromCart*: remove o item do carrinho

User

- *signUp*: renderiza o formulário de cadastro de um novo usuário.
- *signUpSuccess*: renderiza uma página com a mensagem de cadastro com sucesso.
- *addUser*: adiciona um novo usuário na base da informação recebida do formulário de cadastro.

Para ilustrar a instanciação das aplicações serão considerados três produtos a partir da LPS proposta no capítulo 4. Para cada uma das aplicações são mostrados o arquivo de configuração e algumas telas.

A primeira aplicação será feita misturando todas as *features* implementadas junto com a página inicial de conteúdo estático. A segunda e a terceira serão geradas variando algumas *features* e assim mostrar como as diferentes versões podem ser criadas.

A execução do projeto é iniciada pelo comando run “nome_do_produto” na linha de comando da SBT. A variável especificada seguido do comando run é o nome do projeto definido como lazy val no arquivo de configuração de build.

Primeira Aplicação

A primeira aplicação da LPS tem como base algumas características comuns em uma aplicação E-Commerce tais como a capacidade de listar produtos e manter as informações manipuladas, que é uma característica básica e serve como base para as demais funcionalidades. A árvore de *feature* é ilustrada abaixo.

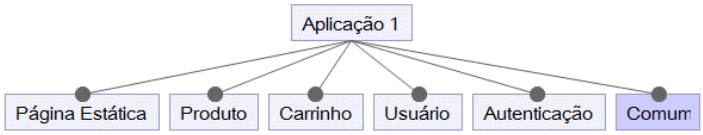

Fig - Modelo de árvore da Aplicação 1

Essa é uma aplicação que compõe as *features* como “página estática”, “produto”, “carrinho”, “autenticação” e “usuário”. Uma característica importante da aplicação é que todo conteúdo do site é restritivo, isto é, o usuário deve efetuar o login no sistema para visualizar o conteúdo do site. A definição da classe que compõe as features é mostrada abaixo.

```
class Product1 @Inject() (
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  override val messagesApiInj: MessagesApi
)
    extends CommonFeature(userServiceInj, productServiceInj, categoryService, messagesApiInj,
      Seq("StaticHomePage", "Product", "Cart", "Auth", "User"))
    with StaticHomePageFeature
    with ProductFeature
    with CartFeature
    with AuthenticationFeature
    with UserFeature {
```

Fig - Definição da classe Product1

Abaixo é mostrado a configuração de rota da primeira aplicação.

```
#Home
GET     /                           controllers.products.Product1.homePage

#Product
GET     /productlist                controllers.products.Product1.productList(p:Int ?=0, s:Int ?= 2, f ?= "")
GET     /productlist/:id            controllers.products.Product1.itemDetails(id: Long)

#Cart
GET     /cart                       controllers.products.Product1.showCart
GET     /cart/remove/:id            controllers.products.Product1.removeItemFromCart(id:String)
GET     /cart/add/:id               controllers.products.Product1.addItemToCart(id: Long)

#Authentication
GET     /login                      controllers.products.Product1.login
GET     /logout                     controllers.products.Product1.logout
GET     /authenticate               controllers.products.Product1.authenticate

#User
GET     /signup                     controllers.products.Product1.signUp
POST    /signup                     controllers.products.Product1.addUser
GET     /signup/success             controllers.products.Product1.signUpSuccess
```

Fig - Rota da Aplicação 1

Telas da Primeira Aplicação

O comportamento esperado desta aplicação quando a URL raiz é acessada é redirecionar à página de login caso o usuário ainda não tenha se logado. Qualquer tentativa de acesso a outras páginas resultará em redirecionamento para página de login.

A Figura abaixo ilustra o acesso ao endereço raiz e o seu redirecionamento à URL /login.

Fig - Acesso ao endereço raiz

Segunda Aplicação

Essa instância da LPS é criada sem a função de autenticação, desse modo o site pode ser acessado por qualquer cliente não cadastrado O modelo de árvore e o resultado do acesso ao endereço raiz é mostrado.

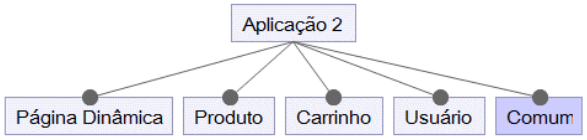

Figura - Modelo de árvore da Aplicação 2

Telas da Segunda Aplicação

Outra característica que diferencia a segunda aplicação da primeira é que esta renderiza uma página inicial de conteúdo dinâmico. Assim, os itens de produtos são consultados e buscados do banco de dados para serem renderizados como ilustrado a seguir.

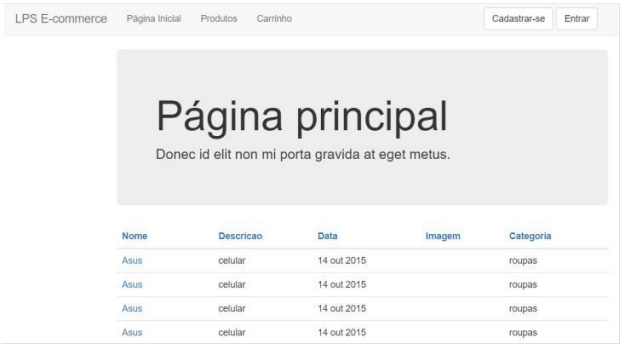

Figura - Página principal da Aplicação 2

A página de detalhes do produto é acessado clicando no nome do produto na lista de produtos. Assim o Play renderizará os detalhes do produto, conforme na figura abaixo.

Fig - Detalhes do item

Para adicionar o item ao carrinho o usuário deve clicar no link “Adicionar ao carrinho” que aparece na página de detalhes ou o item também pode ser adicionado diretamente por um URL: */cart/add/7*. É mostra o carrinho de compras.

Fig – Carrinho

Terceira Aplicação

A terceira aplicação é uma versão da LPS que apresenta somente uma página principal que renderiza um conteúdo dinâmico, buscando a informação do banco de dados.

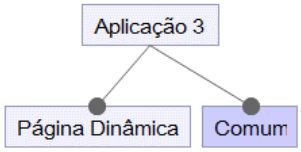

Figura - Modelo de árvore da Aplicação 3

O resultado do acesso à página raiz dessa aplicação que renderiza uma página estática é mostrana na figura abaixo.

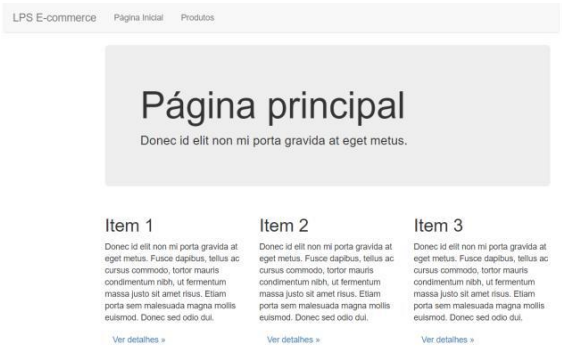

Figura - Página principal da aplicação 3

Conclusões

Este trabalho propôs um protótipo de arquitetura de LPS utilizando o Play Framework no contexto de um sistema de comércio eletrônico. A proposta é composta por três atividades de desenvolvimento (análise de requisitos, arquitetura e codificação). Na fase da codificação foi mostrado como as *features* capturadas na análise de domínio foram implementadas usando *trait* em Scala como uma forma de mecanismo de variabilidade.

A abordagem foi exemplificada e validada por meio três aplicações simples. O trabalho também descreveu as tecnologias e ferramentas que foram utilizadas durante o desenvolvimento, como Play Framework, Scala e SBT.

Referências Bibliográficas

CLEMENTS, P.; NORTHROP, L. **Software Product Lines:** Practices and Patterns. 3. ed. [S.l.]: Addison-Wesley Prossional, 2001.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Formalizing cardinality-based feature models and their specialization, In Software Process Improvement and Practice, 2005.

CZARNECKI, K.; HELSEN, S.; EISENECKER, U. Staged configuration through specialization and multi-level configuration of feature models, In Software Process Improvement and Practice, 2005.

CZARNECKI, K.; KIM, C. H. P. Cardinality-based feature modeling and constraints: a progress report, In Proc. International Workshop on Software Factories, 2005.

DEURSEN, A. V.; KLINT, P. Domain-specific language design requires feature. **Journal of Computing and Information Technology**, 2001.

HILTON, P.; BAKKER, E.; CANEDO, F. **Play for Scala**. [S.l.]: Mannings Publications Co, 2013.

J LEE, D. M. M. N. A Feature-oriented Approach for Developing Reusable Product Line Assets of Service-based Systems. **The Journal of Systems and Software**, 2010.

KANG, K. C.. C. S. G.. H. J. A.. N. W.. P. A. **Feature-Oriented Domain Analysis (FODA) Feasibility Study**. [S.l.]: Software Engineering Institute: Carnegie Mellon University. 1990.

LINDEN, F. J. V. D.; SHMID, K.; ROMMES, E. **Software Product Lines in Action**. 1. ed. [S.l.]: Springer, 2007.

APÊNDICE B – Código fonte

```scala
package controllers.products

import play.api.mvc._
import javax.inject.{ Inject, Singleton }

import play.api.mvc._
import service._
import utils._
import play.api.data.{ Form, _ }
import play.api.data.Forms._
import play.api.i18n.{ I18nSupport, MessagesApi }
import controllers.features._

class Product1 @Inject() (
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  override val messagesApiInj: MessagesApi
)
    extends CommonFeature(userServiceInj, productServiceInj, categoryService,
messagesApiInj,
      Seq("StaticHomePage", "Product", "Cart", "Auth", "User"))
    with StaticHomePageFeature
    with ProductFeature
    with CartFeature
    with AuthenticationFeature
    with UserFeature {

  // HomePageFeature
  override def homePage: EssentialAction = {
    IsAuth {
      super.homePage
    }
  }

  // ProductFeature
  override def productList(page: Int, orderBy: Int, filter: String): EssentialAction = {
    println("Product1 - productList")
    IsAuth {
      super.productList(page, orderBy, filter)
    }
  }

  override def itemDetails(id: Long): EssentialAction = {
    println("Product1 - productList")
    IsAuth {
      super.itemDetails(id)
    }
  }

  override def showCart: EssentialAction = {
    IsAuth {
      super.showCart
    }
  }
}
```

Product1.scala

```scala
package controllers.products

import play.api.mvc._
import javax.inject.{ Inject, Singleton }

import play.api.mvc._
import service._
import utils._
import play.api.data.{ Form, _ }
import play.api.data.Forms._
import play.api.i18n.{ I18nSupport, MessagesApi }

//Features
import controllers.features._

class Product2 @Inject() (
```

```
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  override val messagesApiInj: MessagesApi
)
    extends CommonFeature(userServiceInj, productServiceInj, categoryService,
messagesApiInj,
      Seq("DynamicHomePage", "Product", "Cart", "Auth", "User"))
    with DynamicHomePageFeature
    with ProductFeature
    with CartFeature
    with AuthenticationFeature
    with UserFeature {

  // HomePageFeature
  override def homePage: EssentialAction = {
    IsAuth {
      super.homePage
    }
  }

  // ProductFeature
  override def productList(page: Int, orderBy: Int, filter: String): EssentialAction = {
    println("Product2 - productList")
    IsAuth {
      super.productList(page, orderBy, filter)
    }
  }

  override def itemDetails(id: Long): EssentialAction = {
    println("Product2 - productList")
    IsAuth {
      super.itemDetails(id)
    }
  }

  override def showCart: EssentialAction = {
    IsAuth {
      super.showCart
    }
  }
}
```

Product2.scala

```
package controllers.products

import play.api.mvc._
import javax.inject.{ Inject, Singleton }

import play.api.mvc._
import service._
import utils._
import play.api.data.{ Form, _ }
import play.api.data.Forms._
import play.api.i18n.{ I18nSupport, MessagesApi }
import controllers.features._

class Product3 @Inject() (
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  override val messagesApiInj: MessagesApi
)
    extends CommonFeature(userServiceInj, productServiceInj, categoryService,
messagesApiInj,
      Seq("StaticHomePage", "Product"))
    with StaticHomePageFeature
    with ProductFeature {
}
```

Product3.scala

```scala
package controllers.features

import play.api.mvc._
import javax.inject.{ Inject, Singleton }

import play.api.mvc._
import service._
import utils._
import play.api.i18n.{ I18nSupport, MessagesApi }
import play.twirl.api.Html
import security.Secured

@Singleton
class CommonFeature @Inject() (
  userServiceInj: UserService,
  productServiceInj: ProductService,
  categoryService: CategoryService,
  val messagesApiInj: MessagesApi,
  implicit val featureList: Seq[String]
)
    extends Controller with I18nSupport with CommonHelper {

  val userService = userServiceInj
  val productService = productServiceInj
  val messagesApi = messagesApiInj

  def main(body: Option[Html] = None): Result = {
    Ok(views.html.index(body.get))
  }

  def index(body: Option[Html] = None): EssentialAction = Action { implicit request =>

    Ok(views.html.index(body.get))

  }
}

trait UserFeature extends CommonFeature {

  // o problema eh que o view precisa levar a request como parametro pq o view precisa
de Messages
  //
  def signUp = Action { implicit request =>
    super.main(Some(
      views.html.sign_up() // mudar
    ))
  }
  //once again, bindFromRequest, how can I
  def addUser = Action { implicit request =>
    FormHelper.signUpForm.bindFromRequest.fold(
      formWithErrors => {
        println(formWithErrors.data)
        BadRequest(
          views.html.index(body = views.html.sign_up())
        )
      }, {
        case (email, name, cpf, dateOfBirth, address, password) =>

          userService.insert(email, name, cpf, dateOfBirth, address, password)
          // aqui eh so definir o metodo nesse Trait e colocar o route no route file
          Redirect(routes.UserFeature.signUpSuccess())
      }
    )
  }

  def signUpSuccess = Action { implicit request =>
    super.main(Some(
      views.html.sign_up_success()
    ))
  }

}

trait DynamicHomePageFeature extends CommonFeature {

  def homePage: EssentialAction = {
```

```
    val page = 0
    val orderBy = 2
    val filter = ""

    super.index(Some(views.html.dynamic_homepage(
      productService.list(page = page, orderBy = orderBy, filter = ("%" + filter +
"%")),
      orderBy, filter
    )))
  }
}

trait StaticHomePageFeature extends CommonFeature {

  def homePage: EssentialAction = {
    super.index(Some(
      views.html.static_homepage()
    ))
  }

  /*
  override abstract def index(body: Option[Html] = None): EssentialAction = {

    //implicit val body = views.html.index()
    super.index(Some(views.html.index()))
  }*/
}

trait CartFeature extends CommonFeature {

  def showCart: EssentialAction = Action { implicit request =>
    println("*** ENTERED 'showCart' ACTION ***")
    println(s"  session data : ${request.session.data}")

    val cartWithoutEmail = request.session.data.filter(_._1 != "email")

    println(s"  cartWithoutEmail data : ${cartWithoutEmail}")

    val cart: List[SessionToCart] = cartWithoutEmail.map {
      case (k, v) =>

        val number = k.filter(_.isDigit)

        SessionToCart(number, v)

    }.toList

    println(s"Cart content : ${cart}")

    Ok(views.html.index(
      body = views.html.cart(cart)
    ))
  }

  def addItemToCart(id: Long) = Action { implicit request =>
    productService.findById(id).map { product =>
      Redirect(routes.ProductFeature.productList(0, 2, ""))
        .withSession(request.session + (product.id.toString -> product.name))
    }.getOrElse(NotFound)
  }

  def removeItemFromCart(id: String) = Action { implicit request =>


    val notEqual = request.session.data.filter { (cart) =>
      cart._1.filter(_.isDigit) != id

    }


    val newSession = Session(notEqual)


    val cartWithoutEmail = newSession.data.filter(_._1 != "email")

    val cart: List[SessionToCart] = cartWithoutEmail map { item => SessionToCart(item._1
filter (_.isDigit), item._2) } toList
```

```
    println(s"Cart content after : ${cart}")

    Ok(views.html.index(
      body = views.html.cart(cart)
    )).withSession(newSession)
  }

}

trait ProductFeature extends CommonFeature {

  def productList(page: Int, orderBy: Int, filter: String): EssentialAction = {

    super.index(
      Some(views.html.product_list(
        productService.list(page = page, orderBy = orderBy, filter = ("%" + filter +
"%")),
        orderBy, filter
      ))
    )

  }

  //deletar o metodo no Common
  def itemDetails(id: Long): EssentialAction = {

    productService.findById(id).map { product =>

      super.index(Some(
        views.html.item_details(product)
      ))

    }.getOrElse(super.index(None))
  }
}

trait AuthenticationFeature extends CommonFeature with Secured {

  override abstract def index(body: Option[Html] = None): EssentialAction = {

    IsAuth { super.index(None) }
  }

  def login: EssentialAction = {
    super.index(Some(
      views.html.login(loginForm)
    ))
  }

  def logout = Action { implicit request =>

    Redirect(controllers.features.routes.AuthenticationFeature.login).withNewSession
  }

  def authenticate: EssentialAction = Action { implicit request =>
    loginForm.bindFromRequest.fold(
      formWithErrors =>
Redirect(controllers.features.routes.AuthenticationFeature.login),
      user => Redirect(controllers.features.routes.ProductFeature.productList(0, 2,
"")).withSession("email" -> user._1)
    )
  }
```

Application.scala

```
@(changePassForm: Form[(String, String, String)])(implicit flash: Flash, messages:
Messages)
@import helper._

<div class="well">
    <p>Admin page!</p>
    <p>You probably should not be here!!</p>

    <p class="text-success">@flash.get("chgPassMsg")</p>
    @helper.form(action = controllers.common.routes.Application.changePass) {
        <legend>Change your password:</legend>

        @helper.inputPassword(changePassForm("currPass"), '_label -> "Current Password:
")
        @inputPassword(changePassForm("newPass"), '_label -> "New Password: ")
        @inputPassword(changePassForm("newPassRepeat"), '_label -> "New Password Again:
")

        <input type="submit" value="Enviar">
    }
    <p class="text-error">@changePassForm.error("").map(_.message)</p>
</div>
```

change_pw.scala.html

```
@import service.User
@(body : Html, user: Option[User] = None)
<!DOCTYPE html>

<html>
    <head>
        <title></title>

        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <meta name="description" content="E-commerce LPS in dev">

        <link rel="shortcut icon" type="image/png"
href='@routes.Assets.versioned("images/favicon.png")'>
        <link rel="stylesheet" type="text/css" media="screen"
href="@routes.Assets.versioned("lib/bootstrap/css/bootstrap.css")">
        <script data-main="@routes.Assets.versioned("javascripts/main.js")"
type="text/javascript"
src="@routes.Assets.versioned("lib/requirejs/require.js")"></script>
        <link rel="stylesheet" media="screen"
href="@routes.Assets.versioned("stylesheets/main.css")"/>
        <script
        src="https://code.jquery.com/jquery-3.1.1.min.js"
        integrity="sha256-hVVnYaiADRTO2PzUGmuLJr8BLUSjGIZsDYGmIJLv2b8="
        crossorigin="anonymous"></script>

        <script src="@routes.Assets.versioned("lib/bootstrap/js/bootstrap.js")"
type="text/javascript"></script>

    </head>
    <body>
        <div class="container">

            <nav class="navbar navbar-default">
                <div class="container-fluid">
                        <!-- Brand and toggle get grouped for better mobile display -->
                    <div class="navbar-header">
                        <button type="button" class="navbar-toggle collapsed" data-
toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
                            <span class="sr-only">Toggle navigation</span>
                            <span class="icon-bar"></span>
                            <span class="icon-bar"></span>
                            <span class="icon-bar"></span>
                        </button>
                        <a class="navbar-brand" href="#">LPS E-commerce</a>
                    </div>

                        <!-- Collect the nav links, forms, and other content for
toggling -->
                    <div class="collapse navbar-collapse" id="bs-example-navbar-
collapse-1">
```

```
                        <ul class="nav navbar-nav">
                            <li><a href="#">Produtos</a></li>
                            <li><a href="#">Categoria</a></li>
                            <li><a href="#">Carrinho</a></li>
                        </ul>

                        <form class="navbar-form navbar-right" action="/login">
                            <button type="submit" class="btn btn-default">
                                <a href="#">
                                    Cadastrar-se
                                </a>
                            </button>

                            <button type="submit" class="btn btn-default">
                                <a href="#">
                                    Entrar
                                </a>
                            </button>
                        </form>
                    </div><!-- /.navbar-collapse -->
                </div><!-- /.container-fluid -->
            </nav>


            <div class="container-fluid">
                <div class="row">
                    <div class="col-md-2"></div>
                    <div class="col-md-10">@body</div>
                </div>

            </div>
        </div>
    </body>
</html>
```

Index.scala.html

```
@(product: service.Product)(implicit request: Request[AnyContent], messages: Messages)

@import helper._
@import b3.vertical.fieldConstructor

    <div class="well">
        <table style="width: 100%">
            <tr>
                <td>Nome</td>
                <td>@product.name</td>
            </tr>
            <tr>
                <td>Descri&ccedil;&atilde;o</td>
                <td><pre>@product.description</pre></td>
            </tr>
        </table>
        <div>
            <a href="#">
                Adicionar ao carrinho
            </a>
        </div>
        <div>
            <img src="">
        </div>
        <hr />
    </div>
```

Item_details.scala.html

```
@import service._
@(currentPage: Page[Category], currentOrderBy: Int, currentFilter: String)(implicit
flash: play.api.mvc.Flash, messages: Messages)

@import b3.vertical.fieldConstructor
@header(orderBy: Int, title: String) = {
    <th class="col@orderBy header @if(scala.math.abs(currentOrderBy) == orderBy)
```

```
@{if(currentOrderBy < 0) "headerSortDown" else "headerSortUp"}">
        <a href="@link(0, Some(orderBy))">@title</a>
    </th>
}

<h1>@Messages("Lista de categorias", currentPage.total)</h1>

@Option(currentPage.items).filterNot(_.isEmpty).map { category =>

<table class="table">
    <thead>
        <tr>
            @header(1, "#")
            @header(2, "Nome")
            @header(3, "Descricao")
        </tr>
    </thead>
    <tbody>

    @category.map {
        case (category) => {
            <tr>
                <td>
                    @category.id
                </td>
                <td>
                   @category.name</a>
                </td>
                <td>
                @category.description.getOrElse("No Description")
                </td>
            </tr>
        }
    }

    </tbody>
</table>

<div id="pagination" class="pagination">
    <ul>
        @currentPage.prev.map { page =>
        <li class="prev">
            <a href="@link(page)">&larr; Previous</a>
        </li>
        }.getOrElse {
            <li class="prev disabled">
                <a>&larr; Previous</a>
            </li>
        }
    <li class="current">
        <a>Displaying @(currentPage.offset + 1) to @(currentPage.offset + category.size)
of @currentPage.total</a>
    </li>
        @currentPage.next.map { page =>
        <li class="next">
            <a href="@link(page)">Next &rarr;</a>
        </li>
        }.getOrElse {
            <li class="next disabled">
                <a>Next &rarr;</a>
            </li>
        }
    </ul>
</div>

}.getOrElse {

    <div class="well">
        <em>Nothing to display</em>
    </div>

}
```

List_category.scala_change.html

```
@()

<div class="row">
    <div class="col-md-4">
        <h2>Item 1</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac
cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.
Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
        <p><a class="btn btn-secondary" href="#" role="button">Ver detalhes
&raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Item 2</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac
cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.
Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
        <p><a class="btn btn-secondary" href="#" role="button">Ver detalhes
&raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Item 3</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac
cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.
Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
        <p><a class="btn btn-secondary" href="#" role="button">Ver detalhes
&raquo;</a></p>
    </div>
</div>

<div class="row">
    <div class="col-md-4">
        <h2>Item 4</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac
cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.
Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
        <p><a class="btn btn-secondary" href="#" role="button">Ver detalhes
&raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Item 5</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac
cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.
Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
        <p><a class="btn btn-secondary" href="#" role="button">Ver detalhes
&raquo;</a></p>
    </div>
    <div class="col-md-4">
        <h2>Item 6</h2>
        <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac
cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.
Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui. </p>
        <p><a class="btn btn-secondary" href="#" role="button">Ver detalhes
&raquo;</a></p>
    </div>
</div>
```

Static_product_list.scala.html

```
#Home
GET     /                           controllers.products.Product1.homePage

#Product
GET     /productlist                controllers.products.Product1.productList(p:Int ?=0,
s:Int ?= 2, f ?= "")
GET     /productlist/:id            controllers.products.Product1.itemDetails(id: Long)

#Cart
GET     /cart                       controllers.products.Product1.showCart
GET     /cart/remove/:id
controllers.products.Product1.removeItemFromCart(id:String)
GET     /cart/add/:id               controllers.products.Product1.addItemToCart(id:
Long)
```

```
#Authentication
GET     /login                      controllers.products.Product1.login
GET     /logout                     controllers.products.Product1.logout
GET     /authenticate               controllers.products.Product1.authenticate

#User
GET     /signup                     controllers.products.Product1.signUp
POST    /signup                     controllers.products.Product1.addUser
GET     /signup/success             controllers.products.Product1.signUpSuccess


GET     /assets/*file               controllers.Assets.versioned(path="/public", file:
Asset)
```

Product1.routes

```
#Home
GET     /                           controllers.products.Product2.homePage

#Product
GET     /productlist                controllers.products.Product2.productList(p:Int ?=0,
s:Int ?= 2, f ?= "")
GET     /productlist/:id            controllers.products.Product2.itemDetails(id: Long)

#Cart
GET     /cart                       controllers.products.Product2.showCart
GET     /cart/remove/:id
controllers.products.Product2.removeItemFromCart(id:String)
GET     /cart/add/:id               controllers.products.Product2.addItemToCart(id:
Long)

#Authentication
GET     /login                      controllers.products.Product2.login
GET     /logout                     controllers.products.Product2.logout
GET     /authenticate               controllers.products.Product2.authenticate

#User
GET     /signup                     controllers.products.Product2.signUp
POST    /signup                     controllers.products.Product2.addUser
GET     /signup/success             controllers.products.Product2.signUpSuccess


GET     /assets/*file               controllers.Assets.versioned(path="/public", file:
Asset)
```

Product2.routes

```
#Home
GET     /                           controllers.products.Product3.homePage

#Product
GET     /productlist                controllers.products.Product3.productList(p:Int ?=0,
s:Int ?= 2, f ?= "")
GET     /productlist/:id            controllers.products.Product3.itemDetails(id: Long)




GET     /assets/*file               controllers.Assets.versioned(path="/public", file:
Asset)
```

Product3.routes

```
 /*
  * Features
  */

lazy val features = (project in
file("features/")).enablePlugins(PlayScala).aggregate(common).dependsOn(common).settings
```

```
(
  scalaVersion := "2.11.8",
  name := "features",
  PlayKeys.devSettings ++=
    Seq(("play.http.router", "features.Routes"),
      ("db.default.driver", "org.h2.Driver"),
      ("db.default.url", "jdbc:h2:mem:play"),
      ("play.evolutions.enabled", "true"))
)

/*
 * Products
 */

// StaticHomePage, Product, Cart, Authentication, User
lazy val product1 = (project in file("prod/product1")).enablePlugins(PlayScala)
  .aggregate(features)
  .dependsOn(features).settings(
  scalaVersion := "2.11.8",
  name := "product1",
  PlayKeys.devSettings ++=
    Seq(("play.http.router", "product1.Routes"),
      ("db.default.driver", "org.h2.Driver"),
      ("db.default.url", "jdbc:h2:mem:play"),
      ("play.evolutions.enabled", "true"))
)

// DynamicHomePage, Product, Cart, Authentication, User
lazy val product2 = (project in file("prod/product2")).enablePlugins(PlayScala)
  .aggregate(features)
  .dependsOn(features).settings(
  scalaVersion := "2.11.8",
  name := "product1",
  PlayKeys.devSettings ++=
    Seq(("play.http.router", "product2.Routes"),
      ("db.default.driver", "org.h2.Driver"),
      ("db.default.url", "jdbc:h2:mem:play"),
      ("play.evolutions.enabled", "true"))
)

// StaticHomePage, Product
lazy val product3 = (project in file("prod/product3")).enablePlugins(PlayScala)
  .aggregate(features)
  .dependsOn(features).settings(
  scalaVersion := "2.11.8",
  name := "product3",
  PlayKeys.devSettings ++=
    Seq(("play.http.router", "product3.Routes"),
      ("db.default.driver", "org.h2.Driver"),
      ("db.default.url", "jdbc:h2:mem:play"),
      ("play.evolutions.enabled", "true"))
)
```

build.sbt