Uma Arquitetura para Linhas de Produto de Software de E-Commerce usando o Framework Play em Scala

Yun Hu Lee, Patrícia Vilain, Leandro José Komosinski

Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) - Florianópolis, SC - Brasil

yeonhoo@gmail.com, {patricia.vilain, leandro.komosinski}@ufsc.br

Abstract. *A Software Product Line (SPL) is based on the reuse of artifacts, avoiding the need to redesign an entire application. It promotes an increase of the target public and improvement in the quality of the applications. This paper presents a SPL architecture using the Framework Play in the context of an E-Commerce system. The proposed approach consists of three development activities: requirements analysis, architecture and codification. This approach is exemplified and validated by three applications from the constructed SPL.*

Resumo. *Uma Linha de Produto de Software (LPS) baseia-se em um planejamento de reutilização de artefatos já construídos em novas aplicações, sem a necessidade de projetar novamente toda uma aplicação. Isso promove o aumento do público alvo e melhoria na qualidade das aplicações. Este artigo apresenta uma arquitetura de LPS utilizando o Framework Play no contexto de um sistema E-Commerce. A proposta é composta por três atividades de desenvolvimento: análise de requisitos, arquitetura e codificação. A abordagem é exemplificada e validada por meio três aplicações a partir da LPS construída.*

1. Introdução

Sistemas web evoluíram rapidamente de uma simples coleção de páginas estáticas para aplicações complexas e ricas de conteúdo. Sistemas E-Commerce (comércio eletrônico) podem ser considerados como aplicações web que se caracterizam basicamente por fornecer serviços relacionados a compras na internet. O aumento do comércio eletrônico vem motivando o desenvolvimento de tecnologias que facilitam o seu desenvolvimento.

Sistemas e-commerce podem ser derivados 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, entre outras [Albertin 2004]. De acordo com (Mathias and Oliveira Junior 2012), é possível gerenciar tais serviços, similaridades ou variabilidades, por meio da abordagem de Linha de Produto de Software (LPS), permitindo que várias aplicações sejam desenvolvidas por meio da instanciação de uma infraestrutura comum.

Na abordagem de LPS, artefatos são reutilizados de forma sistemática e consistente na construção de novas aplicações [Pohl, Böckle and Linden 2005]. Artefatos reutilizáveis abrangem todos os tipos de artefatos, tais como modelos de requisitos, modelos de arquitetura, componentes de software e planos de teste.

Atualmente a linguagem Scala é utilizada por empresas importantes e oferece a expressividade necessária para desenvolver uma LPS. Porém, ainda não é possível encontrar casos reais que utilizem esta linguagem para o desenvolvimento de LPS

[Thaker 2008], como pode ser constatado na revisão da literatura realizada neste trabalho. Nesta direção, este artigo propõe uma arquitetura de LPS de E-Commerce usando Framework Play em Scala.

Este artigo está organizado da seguinte maneira, a Seção 2 apresenta sucintamente o conceito de LPS e as ferramentas de suporte utilizadas. A Seção 3 apresenta a revisão do estado da arte. A Seção 4 descreve a proposta de desenvolvimento da LPS. A Seção 5 apresenta três exemplos de aplicações com base na LPS construída. A seção 6 conclui o trabalho.

2. Fundamentação Teórica

Nesta seção são explicados sucintamente os conceitos necessários para o entendimento do trabalho: LPS, Scala, SBT e o Framework Play.

2.1. Linha de Produto de Software

De acordo com (Clements and 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. Estes produtos satisfazem as necessidades de um mercado específico e são desenvolvidos a partir de recursos comuns de uma maneira pré-definida.

A metodologia de desenvolvimento da LPS requer duas atividades. 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 e é através desta que os produtos da plataforma estabelecida na engenharia de domínio são derivados [Pohl, Böckle and Linden 2005].

2.2. Linguagem Scala

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*. Ela incorpora recursos de linguagens orientadas a objetos e funcionais [Odersky, Spoon and Venners 2008]. Empresas importantes 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. SBT e Framework Play

O SBT (Simple Build Tool) é uma ferramenta de automação de *build* para projetos em Scala e em Java semelhante às ferramentas Maven e Gradle. Ela simplifica o processo de desenvolvimento de *build* e auxilia na manutenção do projeto. O 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 and Canedo 2013]. Desde a versão 2.0, a parte de *build* e *deployment* migrou para o SBT e passou a usar o sistema de apresentação (*template*) na linguagem Scala.

3. Estado da Arte

Foi realizada uma revisão da literatura buscando identificar (i) os estudos existentes no desenvolvimento de uma LPS para sistemas e-commerce usando Scala; e (ii) as técnicas

específicas aplicadas a LPS para sistemas e-commerce usando Scala. Para tanto, foram pesquisados os seguintes termos:

Termo 1: (E-Commerce OR "Electronic Commerce") AND (SPL OR SPLE OR (software AND (product line OR product lines OR product family OR product families)))

Termo 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))))

Os termos 1 e 2 foram pesquisados nas seguintes bases de dados: IEEE Xplore[1], ACM Digital Library[2], Scopus[3] e CiteSeerX[4], onde foram pesquisados estudos publicados em língua inglesa entre janeiro de 2005 e abril de 2017. 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. É importante destacar que o termo 2, que é uma pesquisa mais específica procurando trabalhos em Scala, não retornou nenhum estudo. Após a remoção das redundâncias e a leitura dos títulos e dos resumos de cada estudo, foram selecionados 3 trabalhos apresentados a seguir.

Em [Laguna 2010] foi desenvolvida 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 pequenas e médias empresas não especialistas na área possam utilizar o paradigma de LPS para desenvolvimento de web.

O trabalho [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.

Em [Tawhid and Petriu 2008] é apresentada uma abordagem para a transformação de modelos UML (modelo de casos de uso, modelo de classes e modelo de implantação) em produtos específicos. 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

Nesta proposta o desenvolvimento da LPS começa com a análise de domínio, quando são coletadas as informações sobre sistemas que compartilham um conjunto comum de recursos e dados [Kang et al 1990]. Em seguida é definida a arquitetura da LPS e é feita a codificação da LPS.

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 et al 1990]. As *features* descritas aqui são relacionadas à interação com o usuário e muitas são diretamente

[1] http://ieeexplore.ieee.org

[2] http://portal.acm.org

[3] http://www.scopus.com

[4] http://citeseerx.ist.psu.edu

visíveis ao usuário. A Figura 1 ilustra o modelo de *features* usado para especificar um sistema E-Commerce configurável. A *feature* raiz (E-Commerce) identifica a LPS.

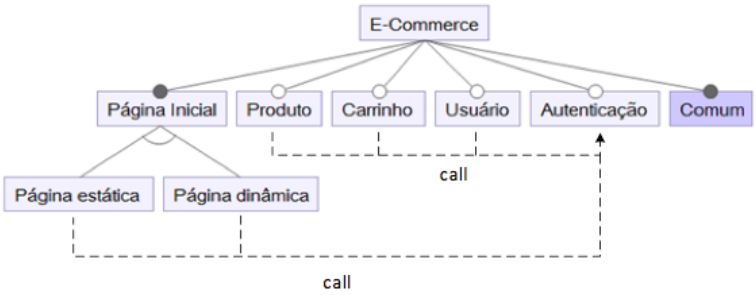

Figura 1. Modelo de features

As *features* opcionais são representadas com um círculo vazio, tal como "Produto", e elas podem ou não fazer parte de uma aplicação final. Por outro lado, as *features* obrigatórias, tal como "Página Inicial", são representadas por círculos preenchidos. *Features* alternativas podem ser exclusivas ou não. As *features* alternativas indicam que apenas uma sub-característica pode ser selecionada. Por exemplo, "Página estática" e "Página dinâmica" são *features* alternativas para a "Página inicial". As *features* não exclusivas indicam que uma ou mais características podem ser selecionadas, tais como "Produto", "Carrinho", "Usuário".

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 possuem esta relação. É bom salientar que esta notação é proposta por este trabalho pois as notações existentes não apresentavam a semântica necessária.

***Feature* Página Inicial.** Toda loja eletrônica possui uma página inicial. É a primeira página que um cliente encontra quando o site é acessado através do endereço URL. A loja eletrônica também pode ser configurada para redirecionar à página inicial se a URL 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 mostra a Figura 2, a página pode ser uma combinação de conteúdo estático e dinâmico.

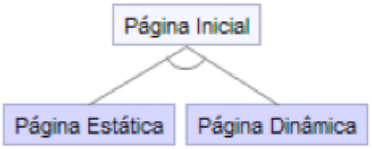

Figura 2. Mapeamento da feature "Página Inicial"

***Feature* Produto.** A *feature* "Produto" é uma *feature* opcional. A informação sobre produtos, exibida na Figura 3, mostra todos atributos que descrevem as características de um produto.

Figura 3. Mapeamento da feature "Produto"

***Feature* Carrinho de Compras.** O carrinho de compras mantém a informação de itens que os clientes desejam comprar durante a sessão. A *feature* possui uma página que mostra o conteúdo do carrinho e funções como a adição e remoção de itens (Figura 4).

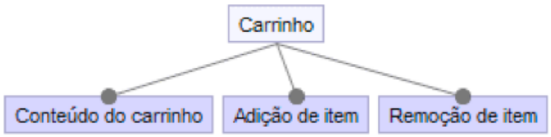

Figura 4. Mapeamento da feature "Carrinho de compras"

***Feature* Usuário.** Esta *feature* é referente ao cadastro de novos usuários (Figura 5). 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.

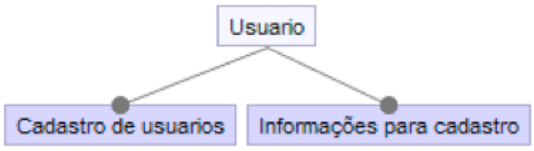

Figura 5. Mapeamento da feature "Usuário"

***Feature* Autenticação.** Com a autenticação habilitada, algumas ações do sistema são restritas para usuários não cadastrados. As funções referentes à autenticação são: login, logout, autenticar (Figura 6).

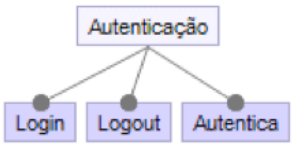

Figura 6. Mapeamento da feature "Autenticação"

***Feature* 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 7).

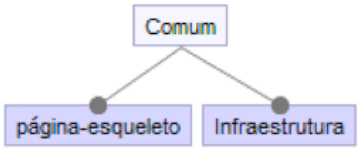

Figura 7. Mapeamento da feature "Comum"

4.2. Arquitetura da LPS de E-Commerce

O modelo de arquitetura proposto foi dividido em três partes: a construção de *features*, o mecanismo de variação estática, e a instanciação de produtos através da combinação de artefatos.

4.2.1 Construção de Features

As *features* capturadas na fase de análise de domínio foram implementadas usando um recurso da linguagem Scala 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 8.

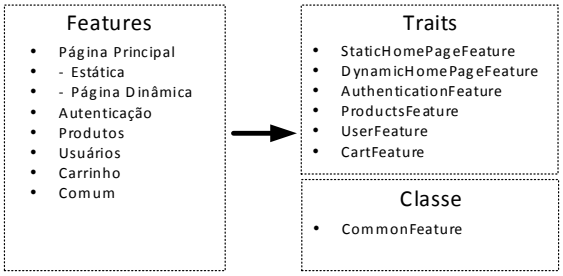

Figura 8. Relação entre features e traits

Um aspecto importante para projetar a aplicação no Framework 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 9.

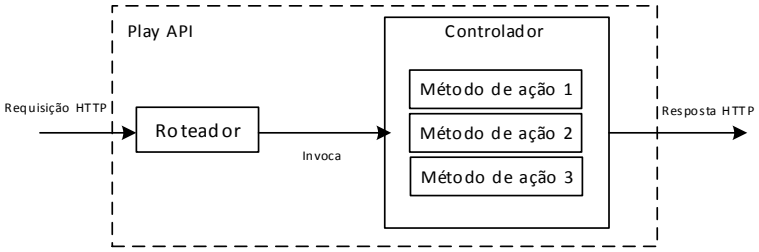

Figura 9. Arquitetura do Framework Play

Neste trabalho as classes de controladores foram implementadas usando *traits*. 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 de 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 definido um projeto no contexto da SBT, significando que cada projeto deve estar mapeado a um diretório onde está definida a classe que implementa as *features* daquela aplicação específica. A definição de um projeto "root" no diretório raiz é feita da seguinte maneira:

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. O código a seguir mostra o exemplo da definição do projeto product1 especificado no arquivo *build.sbt*.

```
// 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"))
)
```

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 a seguir 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 {
```

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 Framework 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 Framework Play irá instanciar um objeto desta classe e carregar o '*carregador de aplicação*'. Um exemplo simplificado é apresentado a seguir.

```
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 consideradas três aplicações a partir da LPS proposta. Para cada uma das aplicações são mostrados o arquivo de configuração e algumas telas. A execução do projeto é iniciada pelo comando run "nome_do_produto" na linha de comando da SBT.

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. O modelo de *features* é ilustrado na Figura 10.

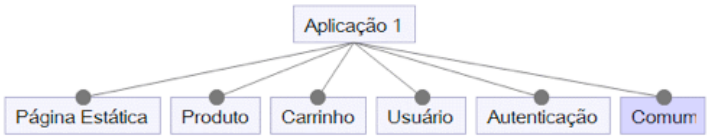

Figura 10. Modelo de features da Aplicação 1

Essa é uma aplicação que compõe as *features* "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 a seguir.

```
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 {
```

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 11 ilustra o acesso ao endereço raiz e o seu redirecionamento à URL /login.

Figura 11. 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 *features* é mostrado na Figura 12.

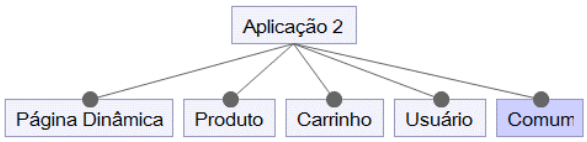

Figura 12. Modelo de features da Aplicação 2

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 13.

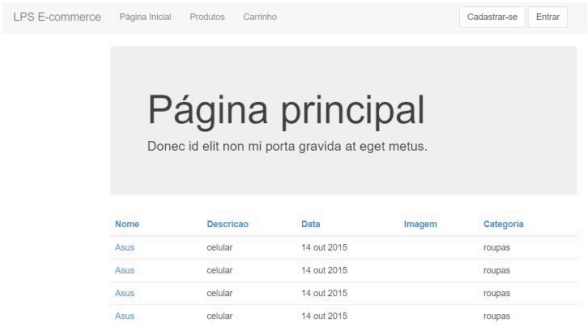

Figura 13. Página principal da Aplicação 2

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 14).

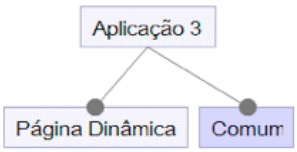

Figura 14. Modelo de features da Aplicação 3

A Figura 15 mostra o resultado do acesso à página raiz dessa aplicação que renderiza uma página estática.

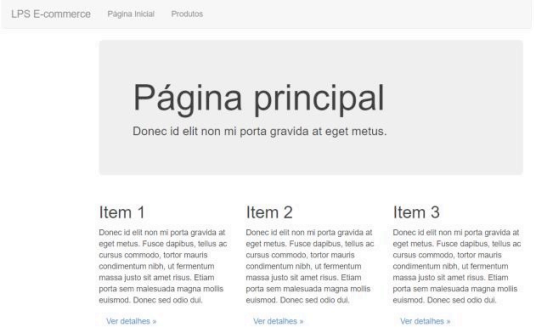

Figura 15. Página principal da Aplicação 3

6. Conclusões

Este trabalho propôs um protótipo de arquitetura de LPS utilizando o Framework Play no contexto de um sistema E-Commerce. A proposta é composta por três atividades de desenvolvimento: análise de requisitos, arquitetura e codificação.

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

Como já mencionado, durante a revisão da literatura não foram encontrados trabalhos que utilizam a abordagem LPS junto com a linguagem Scala. Portanto, esse trabalho pode ser considerado um passo na busca por novas formas de implementação de LPS, servindo também como base para uma avaliação comparativa em futuros estudos.

Referências

Albertin, A. L. Comércio Eletrônico: Modelo, Aspectos, e Contribuições de sua Aplicação. Editora Atlas, 5a. edição, 2004.

Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns. 3. ed. Addison-Wesley Prossional, 2001.

Hilton, P., Bakker, E. and Canedo, F. Play for Scala. Mannings Publications Co, 2013.

Kang, K.C., Cohen, S.G., Hess J.A., Novak W.E. and Peterson A.S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute: Carnegie Mellon University. 1990.

Laguna, Miguel A. “A Software Product Line Approach for E-Commerce Systems”. IEEE 7th International Conference on e-Business Engineering (ICEBE), 20102010.

Lau, S.Q. Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates. Master Thesis. University of Waterloo, Canada. 2006.

Mathias, J.M. and Oliveira Junior, E.A. “Métodos e Técnicas de Desenvolvimento de Linha de Produto de Software para Sistemas E-Commerce: uma Revisão Sistemática”. FITEM 2012 - X Fórum de Informática e Tecnologia de Maringá. 2012.

Odersky, M., Spoon, L. and Venners, B. Programming in Scala. PrePrint Edition Version 2, 2008.

Pohl, K., Böckle, G. and Linden, F. V. D. Software Product Line Engineering. Springer, 2005.

Tawhid, R. and Petriu, D.C. “Towards Automatic Derivation of a Product Performance Model from a UML Software Product Line Model”. Proceedings of the 7th international workshop on Software and performance (WOSP). 2008.

Thaker, S. et al. “Safe composition of product lines”. Generative Programming and Component Engineering. ACM. 2008. p. 95-104.