2
Desenvolvimento de Linhas de Produtos de Softwares

Este capítulo apresenta uma visão geral do desenvolvimento de Linhas de Produtos de Software (LPS). Diversos conceitos relacionados ao desenvolvimento de LPS são, primeiramente, apresentados (Seção 2.1). A abordagem de Desenvolvimento Generativo (DG) também é descrita (Seção 2.2). DG endereça o estudo de métodos e ferramenta que habilitam a produção de membros de uma família de software a partir de especificações de alto nível. Posteriormente, algumas tecnologias frequentemente utilizadas na implementação de arquiteturas de linhas de produtos de software são brevemente detalhadas (Seção 2.3). Finalmente, as principais ferramentas para derivação de produtos de software, presentes no mercado, são apresentadas (Seção 2.4). Essas ferramentas fornecem uma solução automatizada para o processo de derivação baseada no modelo de características.

2.1.
Linhas de Produtos de Software

Uma Linha de Produtos de Software (LPS) (Clements et al. 2001) (Weiss et al. 1999) (Pohl et al. 2005) pode ser vista como uma família de sistemas (Parnas 1976) (Weiss et al. 1999) que endereça um segmento de mercado específico. Entende-se por família de sistemas um conjunto de sistemas especificados, modelados e implementados em termos de funcionalidades comuns e variáveis. Essas funcionalidades podem ser vistas como características de um domínio, que são relevantes para os interessados (do inglês, *stakeholders*) no desenvolvimento e no uso dos sistemas de software. A principal vantagem do uso de características, na modelagem e discriminação de funcionalidades, está no fato de que essas capturam e organizam a terminologia usada por especialistas e usuários de um domínio de aplicação, então, constituindo uma linguagem específica de domínio que pode ser utilizada para facilitar a comunicação entre eles. Um modelo de característica (Kang et al. 1990) pode ser utilizado para modelar as partes comuns e discriminar as diferenças entre os diversos sistemas de software que compõe uma família de

sistemas. Tal modelo foi inicialmente proposto como um componente do método *Feature Oriented Domain Analysis* (FODA) (Kang et al. 1990) desenvolvido pelo Instituto de Engenharia de Software (SEI). O modelo de características proposto por (Kang et al. 1990) é uma representação semântica que descreve o relacionamento estrutural entre as características do domínio. Kang et al. definem três tipos de relacionamentos entre características: composição, generalização e implementado-por. Além disso, características podem ser: opcionais, obrigatórias ou alternativas. Uma característica opcional descreve uma funcionalidade que pode estar ou não presente em produtos de uma família de sistemas. Da mesma forma, uma característica obrigatória representa uma funcionalidade que está presente em todos os produtos de uma família de sistemas. Características alternativas representam funcionalidades mutuamente exclusivas, ou seja, definem um grupo de funcionalidades onde uma, e somente uma, pode estar presente em um produto de uma família de sistemas.

Diversos trabalhos (Greenfield et al. 2005) (Weiss et al. 1999) (Clements et al. 2001) (Pohl et al. 2005) apresentam as vantagens do uso da abordagem de LPS no desenvolvimento de software: (i) redução no custo do desenvolvimento – o reuso de artefatos entre diferentes produtos promove redução no custo do desenvolvimento; (ii) maior evidência na qualidade – o reuso de artefatos testados e revisados entre os diferentes produtos que compõe a família de sistemas aumenta significativamente as chances de detecção e correção de defeitos; (iii) redução no tempo de entrega – inicialmente, o tempo de produção dos artefatos pode ser alto, mas após a produção o tempo de entrega é drasticamente reduzido, através do reuso dos diversos artefatos em cada nova produção; (iv) redução no esforço de manutenção – correções ou modificações em um artefato podem se propagar entre os diversos produtos que compõe a família de sistemas; (v) evoluções gerenciáveis – adaptações/extensões efetuadas em um núcleo de artefatos se refletem automaticamente em todos os produtos da família de sistemas; (vi) complexidade gerenciável – uma núcleo de artefatos bem projetado pode facilitar o gerenciamento de quais artefatos serão reusados e em quais lugares.

A adoção da abordagem (Krueger 2001) de LPS não é uma tarefa fácil e em muitos casos obriga uma reorganização com o objetivo de estabelecer novas unidades modulares comprometidas com o gerenciamento da LPS ou até mesmo a redefinição de processos, fluxos de trabalho e tecnologias utilizadas. O processo de adoção da abordagem de LPS,pode ser feito tipicamente de três formas: (i) pró-ativa – a LPS é analisada, projetada e implementada por

completo, de forma que atenda, inicialmente, o maior número possível de produtos; (ii) reativa – a adoção é feita de maneira incremental, de forma que, a LPS cresça de acordo com a demanda por novos produtos ou novos requerimentos em produtos já existentes; ou (iii) extrativa – a construção da LPS é feita a partir da extração de características comuns e variáveis de um conjunto de softwares bases, previamente desenvolvidos. A abordagem pró-ativa é, em geral, bastante custosa e requer muitos recursos na sua execução. Isso se deve ao fato de que todo o esforço de construção da LPS é concentrado nas fases iniciais da adoção. Em contra partida, as abordagens reativas e extrativas distribuem a construção da LPS em ciclos incrementais, o que permite uma rápida adoção a um custo bem menor. O uso destas abordagens não é mutuamente exclusivo, por exemplo, podemos começar a construção da LPS tendo como base um conjunto de produtos já existentes e incrementalmente evoluir a arquitetura com acréscimos de novos produtos. Essa visão corresponde a adotar inicialmente a abordagem extrativa e, posteriormente, passar para uma abordagem reativa.

O processo de gerenciamento de um núcleo de artefatos de software é composto por dois estágios: engenharia de domínio e engenharia de aplicação. As abordagens de engenharia de domínio (Kang et al. 1990) (Prieto-Diaz et al. 1991) procuram sistematizar a coleta, organização e armazenamento de experiências na construção de software na forma de artefatos reutilizáveis, de maneira que características comuns possam ser exploradas enquanto a habilidade de construção de diferentes produtos é preservada. A engenharia de domínio engloba, basicamente, três atividades: (i) análise de domínio – atividade interessada na definição do escopo para uma determinada família de sistemas e na identificação de características comuns e variáveis presentes neste domínio; (ii) projeto do domínio – atividade que se concentra na especificação de uma arquitetura comum e de componentes para o domínio; e (iii) implementação do domínio – atividade responsável pela implementação da arquitetura comum e dos componentes previamente definidos na atividade de projeto do domínio. Ao final da execução dessas atividades é obtido um conjunto de artefatos prontos para serem reutilizados no desenvolvimento de produtos de software pertencentes à LPS em questão. O processo completo de desenvolvimento de produtos a partir de artefatos construídos pela engenharia de domínio é denominado *engenharia de aplicação* (Sybren et al. 2005) (Jansen et al. 2004). A construção de um produto de software, também chamada derivação (Sybren et al. 2005), é efetuada a partir de uma configuração do núcleo de artefatos. Uma

configuração corresponde a um arranjo dos artefatos e das opções associadas a esses, que implementa parcialmente ou completamente um produto de software.

Inicialmente, durante o processo de derivação (Sybren et al. 2005), o engenheiro de aplicação é responsável por definir, a partir dos artefatos que compõe a LPS, uma configuração base a partir: (i) da construção, geração ou composição de um conjunto de artefatos que compõe a arquitetura da LPS; ou (ii) da seleção dos artefatos que mais se aproximam dos requisitos elicitados para construção do produto a ser derivado, essa seleção pode ser baseada em configurações definidas anteriormente, em uma configuração de referência ou em uma configuração base. Ao final da fase inicial, a configuração construída pode ser validada de forma a determinar sua aderência aos requisitos elicitados. Se tal configuração satisfaz os requisitos, o processo de derivação é finalizado. Em casos onde a configuração inicial não implementa completamente o produto desejado, o processo de derivação pode entrar em uma segunda fase. Nessa segunda fase, o engenheiro de aplicação refina, iterativamente, a configuração criada na fase anterior até que esta represente por completo os requisitos elicitados. Normalmente, durante o processo de derivação, novos requisitos podem surgir por não serem contemplados nos artefatos que compõem a arquitetura da LPS. Para que a arquitetura das LPS passe a atender novos requisitos, certas adaptações ou novas implementações são necessárias e podem ser feitas em dois níveis: (i) adaptações específicas em um produto – corresponde a adaptações em artefatos específicos de um produto e que não são compartilhadas com outros membros da LPS; (ii) evolução reativa – envolve a adaptação do núcleo de artefatos de forma que ele passe a atender requisitos que emergem durante o processo de derivação e que também podem ser divididos com outros membros da LPS. Tais ajustes podem ser feitos através da adição de novas variabilidades ou partes comuns, mudanças em variabilidades ou partes comuns existentes, ou até mesmo remoção de variabilidades ou partes comuns.

Abordagens recentes, como Desenvolvimento Generativo (Czarnecki et al. 2000) e Fábrica de Software (Greenfield et al. 2005) motivam a definição de mecanismos automáticos para suportar a derivação de produtos de software. Estes mecanismos podem melhorar a produtividade e qualidade do processo de derivação. Linguagens Específicas de Domínios (DSLs) e geração de código são as principais tecnologias adotadas por essas abordagens. Diversas ferramentas para derivação de produtos baseadas em modelo de características ou DSLs já estão sendo utilizadas pela industria.

2.2.
Desenvolvimento Generativo

Desenvolvimento Generativo (DG) (Czarnecki et al. 2000) envolve a definição de métodos e ferramentas para a geração automática de softwares a partir de linguagens de especificação de mais alto nível. DG promove a separação do espaço de problema do espaço de solução, através da proposta de uma abordagem baseada na engenharia de domínio (Arango 1993). Um conceito fundamental no DG é o de Modelo de Domínio Generativo. Tal modelo é composto por três elementos principais: (i) espaço do problema – contém os conceitos e características existentes em um dado domínio; (ii) espaço de solução – consiste na arquitetura e componentes usados no projeto e implementação de uma LPS; e (iii) conhecimento de configuração – que define como combinações específicas de características são mapeadas para um conjunto de artefatos de software no espaço de solução. A implementação do conhecimento de configuração deve idealmente ser feita por meio de geradores de código. A Figura 1 apresenta os principais elementos que compõe o modelo generativo.

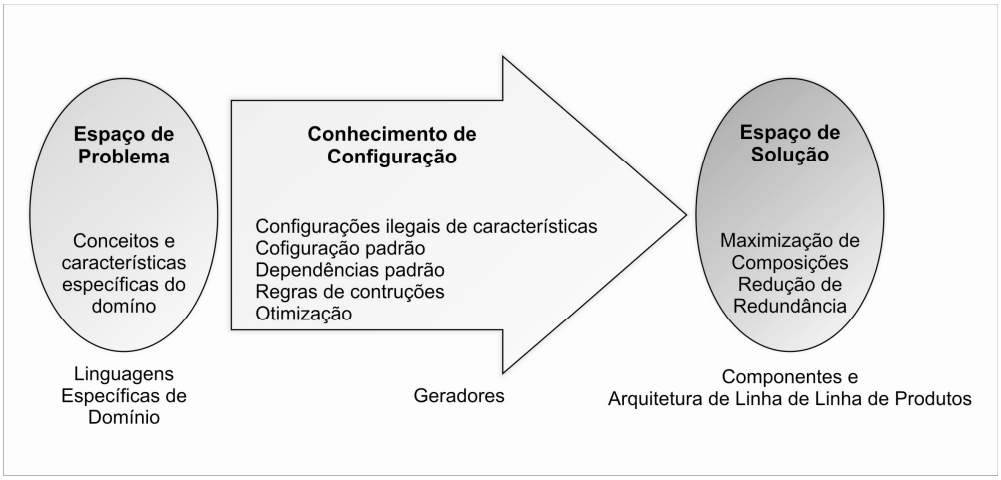

Figura 1. Modelo de Domínio Generativo

(Czarnecki et al. 2000) propõem a implementação do modelo de domínio generativo através dos seguintes passos: (i) definição do domínio; (ii) modelagem de conceitos e características do domínio; (iii) projeto de uma arquitetura comum e identificação do componentes de implementação; (iv) especificação de notações específicas de domínio; (v) especificação do conhecimento de configuração; (vi) implementação de componentes; (vii)

implementação das linguagens específicas de domínio; e (viii) implementação do conhecimento de configuração utilizando geradores. Basicamente, essa abordagem estende processos típicos de engenharia de domínio com duas novas atividades que visam permitir o desenvolvimento de mecanismos apropriados para especificar membros da família de sistemas e modelar detalhadamente o conhecimento de configuração de forma a automatizá-lo através do uso de geradores de código.

De acordo com (Czarnecki et al. 2000), o mapeamento entre o espaço de problema e o espaço de solução pode ser visto sobre duas perspectivas: (i) visão de configuração – nessa perspectiva o espaço de problema é visto como um conjunto de características a serem selecionadas e o espaço de solução consiste em um conjunto de componentes que podem ser combinados e customizados para derivar uma instância de uma LPS. O mapeamento entre tais espaços é definido por regras de construção que indicam como uma certa combinação de características deve ser traduzida para determinadas configurações de componentes; e (ii) visão transformacional – na qual o espaço de problema é representado por uma linguagem específica de domínio (DSL) e o espaço de solução por uma linguagem de programação, e mapeamento entre tais espaço é definido através de transformações que geram código-fonte em uma linguagem de programação a partir de um programa especificado na DSL.

Em uma instanciação particular do modelo de domínio generativo, o modelo de características (Kang et al. 1990) pode ser utilizado como uma DSL de uma LPS. Esse modelo trabalha como uma DSL de configuração onde é possível especificar instâncias concretas (produtos) de uma linha de produto de software ou família de sistemas (Czarnecki et al. 2000). Várias ferramentas (Seção 2.4) adotam essa estratégia para permitir a derivação automática de produtos no desenvolvimento de LPS.

Nesse trabalho é apresentada uma abordagem para derivação de produtos de software centrada nas idéias do modelo de domínio generativo. Na instância do modelo generativo proposta nesse trabalho, o mapeamento entre o espaço de problema e o espaço de solução é concebido, através de uma visão de configuração onde as características selecionadas fornecem informações para combinação e customização dos componentes que formarão o produto final. O modelo de características e o modelo de implementação da LPS são adotados para representar, respectivamente, o espaço de problema e espaço de solução de nosso modelo generativo.

2.3. Implementando Linhas de Produtos de Software

Essa seção apresenta uma visão geral das tecnologias frequentemente utilizadas na implementação de arquiteturas de LPS. A tecnologia de Frameworks Orientados a Objetos (OO), uma das mais populares para a implementação de arquiteturas de LPS, é comentada na seção 2.3.1. Posteriormente, a abordagem de Desenvolvimento de Software Orientado a Aspectos (DSOA), cujo objetivo é a modularizar interesses e características transversais, é apresentada na seção 2.3.2. E finalmente, a seção 2.3.3, apresenta o conceito de componentes de software e sua utilização no desenvolvimento de LPS é discutida.

2.3.1. Frameworks Orientados a Objetos

Frameworks Orientados a Objetos (OO) (Fayad et al. 1999) (Johnson et al. 1988) é uma técnica que é comumente utilizada na implementação de arquiteturas de LPS. Um framework OO permite a definição de uma arquitetura flexível que pode ser customizada, através de pontos de extensão, para a implementação de diferentes aplicações de um mesmo domínio. Um framework OO é implementado por um conjunto de classes, sendo que, parte dessas classes define comportamentos fixos, enquanto uma outra parte define os comportamentos extensíveis. Dessa forma, um framework OO promove a reutilização, através das partes fixas, e permite uma customização distinta para cada instância, através das partes extensíveis.

Frameworks OO são, tipicamente, classificados em três categorias: (i) caixa-branca (*white-box*); (ii) caixa-preta (*black-box*); e (iii) caixa-siza (gray-box). Em frameworks caixa-branca, os pontos de extensão (hot-spots) são especificados através da definição de classes abstratas ou interfaces, e geralmente, são implementados por padrões de projeto (Gamma et al. 1995). Durante o processo de instanciação devem ser criadas instâncias concretas dos pontos de extensão, através da criação de classes que estendem as classes abstratas ou implementam as interfaces do framework. Em frameworks caixa-preta todas as funcionalidades já se encontram presentes e são fornecidas por um conjunto de componentes. A construção de uma instância se dá através da composição de tais funcionalidades. Frameworks caixa-sinza é uma forma hibrida que possui características de frameworks caixa-branca e caixa-preta.

Durante o processo de instanciação deste tipo de framework, certas partes devem ser concretamente definidas através da extensão de classes abstratas ou implementação de interfaces e outras por composição, configuração ou seleção de funcionalidades já implementadas.

A tecnologia de framework OO traz, em geral, um impacto positivo na produtividade e qualidade de desenvolvimento de aplicações, porque possibilita não apenas a reutilização de código de implementação, mas também reuso de projeto de soluções arquiteturais para um dado domínio. Frameworks OO maduros contribuem para a melhoria da qualidade das aplicações finais gerada, por oferecerem uma implementação estável e bem testada, além de possibilitar a codificação de menos linhas.

A abordagem de derivação de produtos apresentada nesse trabalho pode ser utilizada na instanciação de frameworks. A instanciação dos pontos flexíveis é feita na abordagem proposta, através do uso da tecnologia de templates. Na instanciação de um framework caixa-branca (*white-box*) (Fayad et al. 1999), um template é responsável pela geração da estrutura das classes que implementam um ponto flexível. Posteriormente ao processo de instanciação, o código de tais classes deve ser complementado pelos desenvolvedores. No caso de frameworks caixa-cinza (*gray-box*) onde, geralmente, a customização é realizada por arquivos de configuração, a instanciação se dá a partir da seleção de partes que compõem o arquivo e variam de instância para instância. A codificação desses arquivos de configuração pode também ser feita usando a tecnologia de *templates*. Finalmente, no caso de frameworks caixa-preta (*black-box*) a instanciação se dá a partir da simples seleção dos elementos que implementarão a instância final.

2.3.2. Orientação a Aspectos

Orientação a Aspectos (OA) (Filman et al. 2005) (Kiczales 1997) é uma abordagem de engenharia de software que objetiva a modularização dos chamados interesses transversais. Interesses traversais são aqueles que entrecortam diversos módulos dentro de um sistema de software. Desenvolvimento de Software Orientado a Aspectos (DSOA) encoraja a descrição modular de sistemas de software complexo oferecendo mecanismos para separar claramente a funcionalidade básica do sistema, a qual pode ser endereçada por abordagens já propostas (tais como OO), dos interesses

traversais. DSOA permite a modularização dos interesses tranversais propondo uma nova abstração, denominada aspecto, e novos mecanismos que permitem compor aspectos e abstrações dos paradigmas vigentes (exs: classes, interfaces, métodos, construtores). A composição entre aspectos e abstrações OO são realizadas dentro de pontos específicos, denominados pontos de junção.

Diversos trabalhos de pesquisa apresentados pela comunidade demonstram os benefícios que programação orientada a aspectos pode trazer para a modularização de propriedades sistêmicas ou requisitos não funcionais, tais como, rastreamento (Colyer 2004), auditoria (Colyer 2004), delimitação de transações (Soares et al. 2002), persistência (Rashid et al. 2003) (Soares et al. 2002), segurança (Laddad 2003), tratamento de exceções (Filho et al. 2006), monitoramento e distribuição (Soares et al. 2002) .

Trabalhos recentes (Alves et al. 2005) (Apel et al. 2006) (Apel et al. 2006) (Kulesza et al. 2006) (Lougran et al. 2004) (Mezini et al. 2004) têm explorado o uso de técnicas orientadas a aspectos (OA) no projeto e implementação de arquiteturas de LPS. Nesses trabalhos, a abstração de aspectos é usada para melhorar a modularização de características transversais encontradas no domínio endereçado. Nesse sentido, aspectos contribuem para uma melhor modularização de características transversais opcionais e de integração em uma arquitetura de LPS. Como resultado, aspectos também permitem plugar e desplugar essas características do núcleo da arquitetura da LPS, simplificando, com isso, a complexidade do desenvolvimento de arquiteturas de LPS. Aspectos também podem definir variabilidades específicas através da definição de comportamentos extensíveis ou pontos de junção específicos que serão afetados. Vários mecanismos estão disponíveis nas linguagens/plataformas correntes para implementação dessas variabilidades, como, a definição de aspectos, pontos de junção e métodos abstratos usando AspectJ (Kiczales et al. 2001); e especificação de pontos de corte diretamente em arquivos de configuração do Spring (Johnson et al. 2005).

2.3.3. Tecnologia de Componentes

Um componente (Szyperski 2002) é uma unidade de composição e distribuição que implementa uma interface (contrato) responsável por estabelecer os serviços prestados pelo componente. A separação da especificação de um componente (interface) da sua implementação é o conceito

central da tecnologia de componentes. Essa separação permite um baixo acoplamento entre os diversos componentes que compõem uma aplicação possibilitando a composição, remoção e substituição desses facilmente.

Um componente geralmente é instalado em um *container*. Um *container* é visto como um conjunto de frameworks ou componentes que juntos fornecem várias funcionalidades de infra-estrutura como: (i) gerenciamento de sessão; (ii) persistência; (iii) segurança; (iv) transação; e (v) gerenciamento do ciclo de vida. Durante o processo de instalação de um componente, diversas informações devem ser fornecidas de forma que o *container* entenda como instalar e ativar o componente. Essas informações são, em geral, descritas em um arquivo. Para se descrever um componente (Roman et al. 2004) (Johnson et al. 2005) completamente, em geral, deve se especificar: (i) o modelo de componentes – que descreve como os componentes devem ser implementados e como se dá a interação entre eles; (ii) invariantes – modela regras que determinam o comportamento de cada interface em todas as suas instâncias; (iii) interfaces providas - descreve as operações que um componente implementa; e (iv) interfaces requeridas - descrevem quais operações um componente requer de outros.

Atualmente, diversas infra-estruturas tecnológicas baseadas em componentes foram propostas. O principal objetivo dessas tecnologias é oferecer um modelo unificado que permita o gerenciamento adequado (especificação, adaptação, composição, instalação e execução) de componentes e de suas diferentes configurações. Duas tecnologias importantes vêm sendo adotadas no desenvolvimento de aplicações Java: o framework Spring (Johnson et al. 2005) e a plataforma OSGi (OSGi 2003). Spring é um framework mundialmente adotado na implementação de aplicações corporativas. Ele oferece um modelo para construção de aplicações como uma coleção de componentes simples (chamados *beans*) que podem ser conectados ou customizados usando as tecnologias de injeção de dependência e programação orientada a aspectos. A especificação OSGi define uma arquitetura comum e aberta para provedores de serviços, desenvolvedores, operadores de gateway, instalarem e gerenciarem serviços de forma coordenada. A plataforma OSGi oferece uma ambiente de execução dinâmico onde componentes (*bundles* na terminologia OSGi) podem ser instalados, atualizados ou removidos em tempo de execução. De acordo com a especificação OSGi, aplicações Java são estruturadas em um conjunto de *bundles*, onde cada *bundle* fornece serviços para um usuário final ou para outros *bundles*. Essas tecnologias de

componentes podem ser utilizadas na implementação de arquiteturas de LPS, para permitir uma melhor modularização e gerenciamento das características sendo endereçadas.

Esse trabalho explora o uso destas tecnologias de componentes na implementação de LPS oferecendo suporte para a instanciação automática dos mecanismos oferecidos pelas mesmas. Uma extensão das funcionalidades base da ferramenta GenArch é proposta (Capítulo 5) para incorporar novos modelos e endereçar a derivação de arquiteturas de LPS desenvolvidas com os modelos de componentes Spring e OSGi.

2.4. Ferramentas para Derivação de Produtos de Software Baseadas em Modelo de Características

O processo de derivação de produtos (Sybren et al. 2005) deve, preferencialmente, ser realizado com a ajuda de uma ferramenta de instanciação automática que facilite a seleção, composição e configuração dos artefatos e suas respectivas variabilidades. Gears (Gears 2007) e pure::variants (pure::variants 2007) são exemplos de ferramentas desenvolvidas nesse contexto. Essas ferramentas visam fornecer uma solução automatizada para o processo de derivação de produtos de software baseada na configuração e seleção de características. Nas próximas seções, uma visão geral de tais ferramentas é apresentada.

2.4.1. Gears

Gears (Gears 2007) fornece uma infra-estrutura, um ambiente de desenvolvimento e um configurador que juntos, permitem a criação de linhas de produtos de software a partir da definição de um modelo generativo focado na derivação automática de produtos. A ferramenta Gears permite a adoção/construção de LPS seguindo a abordagem incremental. Dessa forma, é possível iniciar a construção de productos a partir de um ou dois subsistemas, módulos ou artefatos, e então transferir facilmente, através de incrementos gerenciáveis, para uma engenharia de LPS.

A Figura 2 exibe os três componentes que formam a ferramenta Gears. O ambiente de desenvolvimento é utilizado para criar, organizar e manter as LPS. A infra-estrutura corresponde aos arquivos e diretórios adicionados aos artefatos

de software que implementam a LPS. Esse conjunto de arquivos e diretórios permite a ferramenta Gears gerenciar as variabilidades da LPS. O configurador é a parte responsável por derivar instâncias de uma LPS. O processo de derivação é executado a partir de uma descrição do produto que é realizada a partir da configuração de características, atributos e propriedades.

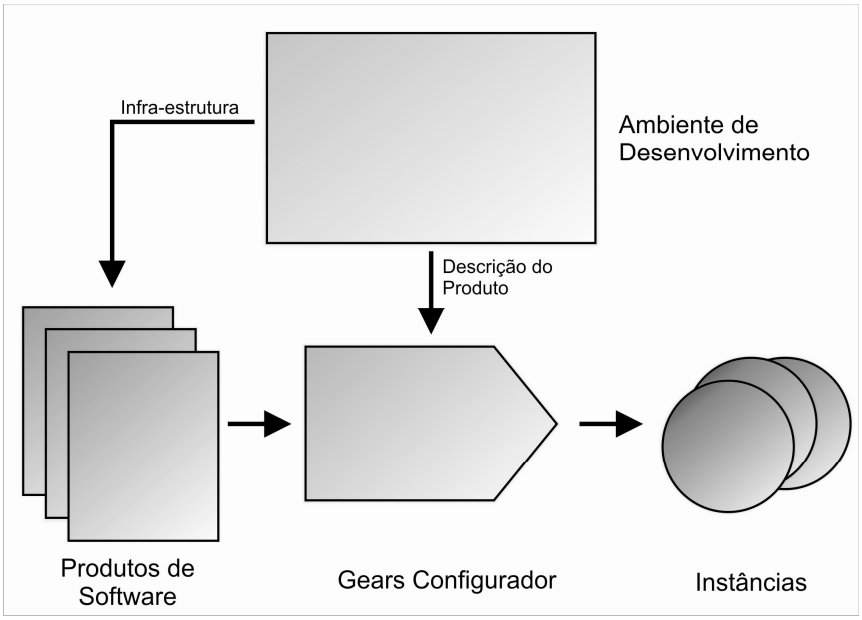

Figura 2. Elementos da ferramenta Gears

Para a ferramenta Gears, um artefato de software é qualquer arquivo configurável (código fonte, requerimentos, casos de teste) construído para ser reusado através da LPS. Cada LPS, na ferramenta, é composta por diversos artefatos e modelada pelas suas características e pontos de variação. Características são parâmetros que representam as variabilidades da LPS. Gears fornece uma linguagem própria para a declaração das características de um produto. Tal linguagem permite a declaração de características de vários tipos, tais como: Boolean, Integer, Float, String, Character, Atom, Record, Enumeration e Set. Regras de associação e dependências entre características podem ser expressas na ferramenta através de um mecanismo de assertivas em lógica proposicional. Valores padrões também podem ser associados a qualquer tipo de característica e utilizados na configuração inicial de um produto. Para se configurar (descrever) um produto, valores devem ser associados aos parâmetros das características. Os valores associados devem estar de acordo com as regras de associações e dependências declaradas por cada característica. Para determinar o mapeamento entre elementos de

implementação e características, Gears utiliza a abstração de pontos de variação. Um ponto de variação encapsula as variações presentes em um arquivo ou diretório. Pontos de variação são especificados utilizando uma linguagem lógica que fornece construções análogas a *switch* ou *case* de linguagens de programação convencionais. Cada cláusula de declaração possui uma expressão booleana e um operador. No momento da derivação, a expressão booleana é primeiramente processada e somente no caso dessa ser verdadeira é que o operador é processado. Um operador tem a função de selecionar um arquivo ou diretório que irá compor o produto derivado. No caso de arquivos texto, diversos operadores de casamento de padrões podem ser aplicados no conteúdo do arquivo após a seleção.

Gears permite o particionamento das características declaradas e suas definições em módulos. Cada módulo pode ter suas próprias características e definições de forma que este pode ser derivado separadamente. Para um melhor gerenciamento das características que podem se espalhar entre diversos módulos, Gears fornece o mecanismo de *mixins* que permite a declaração de uma característica em um lugar central e a mistura dessa entre vários módulos. Para permitir a combinação de vários módulos na criação de uma LPS, Gears utiliza matrizes. Em uma matriz, cada linha representa um produto da LPS e cada coluna um módulo, um *mixin* ou até mesmo outra LPS. A partir dessa estrutura, Gears permite a composição hierárquica da LPS.

Guiado pelas configurações das características, módulos, *mixins* e matrizes, o configurador Gears automaticamente monta e configura os artefatos de software para produzir um novo produto. Esse processo de derivação pode ser feito em vários níveis: (i) no nível de módulo, onde um único módulo é separadamente derivado; (ii) no nível de uma LPS, onde os vários módulos que compõe uma LPS são derivados; e (iii) no nível de LPS hierárquicas, onde a derivação é feita recursivamente entre as diversas LPS componentes.

2.4.2. pure::variants

pure::variants (pure::variants 2007) é uma ferramenta de derivação de LPS que utiliza modelos para descrever o domínio do problema, os elementos de implementação e os produtos que compõem a LPS. A ferramenta pure::variants procura fornecer uma tecnologia que permita o gerenciamento completo de variabilidades, suportando todas as fases do processo de desenvolvimento de

software, desde a análise e projeto, passando pela implementação e teste até o uso e manutenção.

A abordagem de derivação implementada pela ferramenta pure::variants é composta por dois modelos: modelo de características e modelo de família. O modelo de características contém as variabilidades da LPS. Uma característica representa uma propriedade ou funcionalidade do sistema que é visível para o usuário. O modelo de característica implementado pela ferramenta permite a modelagem de características com propriedades e restrições, além de permitir a especificação de relacionamentos entre características. pure::variants permite o uso de um número arbitrário de modelos de característica e de conexões hierárquicas entre estes. O modelo de família descreve a estrutura interna de cada componente e suas dependências em relação às características. Ele é estruturado em vários níveis. O nível mais alto é formado por componentes. Cada componente representa uma ou mais características funcionais e consiste de partes lógicas do software (classes, objetos, funções, variáveis, documentação). A parte física pode ser arquivos que já existem, arquivos que serão criados e ações que serão executadas baseadas no conhecimento de configuração.

A Figura 3 mostra uma visão do processo de derivação suportado pelo pure::variants. A derivação de um produto é feita a partir das informações contidas no modelo de características, no modelo de famílias e em uma seleção das características desejadas. O processo de derivação começa com a criação da instância do modelo de características. A ferramenta verifica a criação da instância do modelo de característica interativamente e tenta resolver automaticamente possíveis problemas que ocorrem ou relatar os que não tiveram solução automática. As regras de validação são expressas usando lógica de primeira ordem descritas em Prolog em uma sintaxe próxima da notação OCL. Após a avaliação da instância do modelo de características e a garantia da sua corretude, uma descrição das transformações é produzida em uma linguagem baseada em XML. pure::variants provê uma linguagem, chamada XMLTrans, que permite descrever ações que devem ser executadas durante o processo de derivação. Esse recurso permite a extensão da ferramenta com novas transformações.

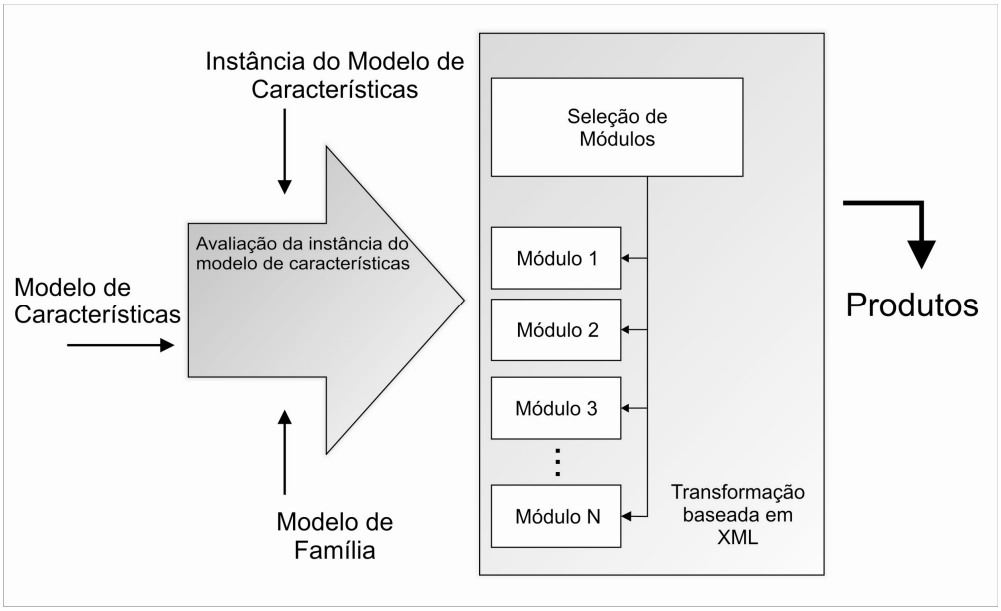

Figura 3. Processo de derivação do pure::variants

pure::variants não requer nenhuma técnica específica de implementação e provê uma interface de integração poderosa, baseada em XML, com outras ferramentas de: engenharia de requisitos, gerenciamento de testes e geração de código. Essa característica permite que o processo de desenvolvimento comece com uma produção parcial, automática, do modelo de família, baseado em um software já existente e continue com uma construção passo-a-passo, executada pelo usuário, do modelo de característica.

2.5. Conclusão

Esse capítulo apresentou os principais conceitos que permeiam a abordagem de LPS, assim como ofereceu uma visão geral das diversas tecnologias adotadas na implementação de arquiteturas de LPS: frameworks OO; desenvolvimento de software orientado a aspectos; e a tecnologia de componentes de software. O capítulo foi concluído apresentando duas das principais ferramentas de derivação de LPS disponibilizadas pela indústria: Gears e pure::variants. Essas ferramentas, embora, ofereçam um conjunto de funcionalidades úteis para a derivação de produtos de software, são, em geral, complexas para serem utilizadas pela comunidade de desenvolvedores, porque incorporam vários conceitos e funcionalidades novas centradas nos novos conceitos de LPS já propostos. Como resultado, elas sofrem das seguintes

deficiências: (i) pode trazer dificuldade para a preparação de arquiteturas de LPS já existentes para serem automaticamente instanciadas; (ii) definição de vários modelos e/ou funcionalidades complexas; e (iii) elas são, em geral, mais adequadas para trabalharem com abordagens pró-ativas.