ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Ligo: Uma linha de produtos de software para gerenciamento de igrejas cristãs

Trabalho de Conclusão de Curso

Engenharia da Computação

Leopoldo Motta Teixeira
Orientador: Prof. Tiago Lima Massoni

Recife, dezembro de 2007

UNIVERSIDADE
DE PERNAMBUCO

ESCOLA POLITÉCNICA
DE PERNAMBUCO

Departamento de
Sistemas
Computacionais

Ligo: Uma linha de produtos de software para gerenciamento de igrejas cristãs

Trabalho de Conclusão de Curso

Engenharia da Computação

Este Projeto é apresentado como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Leopoldo Motta Teixeira
Orientador: Prof. Tiago Lima Massoni

Recife, dezembro de 2007

UNIVERSIDADE
DE PERNAMBUCO

Leopoldo Motta Teixeira

Ligo: Uma linha de produtos de software para gerenciamento de igrejas cristãs

Resumo

A tendência atual de globalização pressiona a indústria de software a explorar maneiras de diversificar e entregar os produtos que desenvolve de maneira rápida e eficiente. Sistemas de software em um mesmo domínio de aplicação costumam possuir diversas características em comum, como é observado no domínio de sistemas de informação para igrejas cristãs. Linhas de produtos de software, que são definidas como conjuntos de sistemas de software que têm características em comum, mas são distintos entre si, podem ser aplicadas neste domínio para diminuir custos de desenvolvimento, maximizando reuso de software e aumentando a qualidade dos produtos desenvolvidos. Este trabalho apresenta o desenvolvimento da linha de produtos de software Ligo, destinada ao gerenciamento de igrejas cristãs. O texto descreve as fases de desenvolvimento da linha de produtos de software, assim como os artefatos gerados em cada fase. Esta linha de produtos foi utilizada para gerar um sistema de informação para uma igreja específica, através da instanciação da linha.

Abstract

The current trend of globalization pressures the software industry to explore means to diversify and deliver the products it develops in a timely and efficient way. Software systems in the same application domain usually share many common features, as observed in the domain of Christian church management systems. Software product lines, defined as sets of software-intensive systems that share common features, but are distinct from each other, can be applied in this domain to reduce development costs, maximizing software reuse and enhancing the quality of the developed products. This work presents the development of the Ligo software product line, targeted to the management of Christian churches. The text describes the development phases of the software product line, as well as the artefacts generated in each one of these phases. This product line was used to generate an information system for a specific church, by product instantiation.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

(Dispostos por ordem de aparição no texto)

LPS – Linha de Produtos de Software
PLUS – *Product Line UML-Based Software Engineering*
UML - *Unified Modeling Language*
ESPLEP - *Evolutionary Software Product Line Engineering*
PHP - *PHP: Hypertext Preprocessor*
HTML - *Hypertext Markup Language*
XML - *Extensible Markup Language*
XSL - *Extensible Stylesheet Language*
AOP - *Aspect Oriented Programming*
XSLT - XSL *Transformations*
ChMS - *Church Management System*
CMS - *Content Management System*
DLL - *Dynamic-link library*
CSS - *Cascading Style Sheets*
SQL - *Structured Query Language*
APDT - *Aspect PHP Development Tools*

Agradecimentos

Agradeço inicialmente a Deus, razão principal deste trabalho e da minha vida. Sem que eu merecesse ou fizesse coisa alguma, pela Sua infinita graça, Ele me adotou e me ajuda a perseverar diariamente.

À minha família, em especial aos meus pais, Aluizio e Ana, que em todos estes anos me apoiaram e procuraram me estimular, propiciando um ambiente sadio e agradável em casa, essencial para a formação do meu caráter. Às minhas avós Graça e Vilma, pelo exemplo dado. Às minhas irmãs, Ella e Raissa, pelas diversas brigas, elas fortaleceram o meu raciocínio argumentativo. =) Vocês são muito importantes para mim.

À minha maravilhosa namorada Raquel, que consegue a façanha de me agüentar, independente do meu humor. Obrigado por estar ao meu lado, me apoiando, pacientemente me acalmando, não me deixando nem cogitar a possibilidade de desistir. Agradeço à família Carneiro Leão também, em especial ao Dr. José Carneiro Leão, pela revisão do texto.

Aos amigos feitos durante estes anos de caminhada na POLI, o meu muito obrigado. Gostaria de poder citar nome a nome, cada um que me marcou durante este tempo. Fico apenas com a menção honrosa para "Os Caras", a turma que me acompanhou desde o início. Valeu galera!

Aos professores do DSC, todos vocês têm sido exemplo de profissionalismo e ética, e foram de fundamental importância na minha formação acadêmica. Em especial, agradeço ao professor Tiago Massoni, que me orientou de forma impecável durante o decorrer deste trabalho, com sugestões, idéias e conselhos que foram essenciais para que eu chegasse até a conclusão do mesmo.

Aos pastores e líderes que ofereceram sugestões para este trabalho. Especialmente ao Pr. Felipe, pela amizade e orientação espiritual nos últimos anos.

Aos meus demais amigos da igreja, açaí, Candelabro, ACBV e todos que, de alguma forma, contribuíram para o que sou hoje. Vocês conseqüentemente, contribuíram com este trabalho.

Finalmente, termino, assim como os cristãos antigos, louvando a Ele, que é o princípio e fim de todas as coisas:

"*Gloria Patri, et Filio, et Spiritui Sancto.*
Sicut erat in principio, et nunc, et in semper, et in sæcula sæculorum.
Amen."

Capítulo 1

Introdução

A tendência atual de globalização pressiona a indústria de software a explorar maneiras de diversificar e entregar os produtos que desenvolve de uma maneira rápida e eficiente. Durante a última década, a abordagem de linhas de produtos de software (LPS) tem surgido como um dos paradigmas de desenvolvimento de software mais promissores para aumentar a produtividade das organizações desenvolvedoras de software [1].

Uma idéia chave em LPS é a de que a maior parte dos sistemas de software não são únicos. Sistemas de software em um mesmo domínio de aplicação compartilham diversas características em comum. A maioria das organizações constrói sistemas de software em um domínio particular, repetidamente lançando variações de produtos, ao adicionar novas funcionalidades. Podemos tirar vantagem desta situação para melhorar este processo, utilizando uma abordagem sistemática de reuso de software.

Linhas de produtos de software são definidas como conjuntos de sistemas de software que têm características em comum, mas são suficientemente distintos entre si [1]. Os produtos que compõem uma LPS compartilham uma infra-estrutura de ativos base, normalmente formada por uma arquitetura de software e seus componentes. Esta infra-estrutura também deve suportar as variações entre os produtos, de modo a compreender ativos que estarão presentes apenas em alguns dos produtos.

Recentemente, temos observado a necessidade e dependência de sistemas de informação em todos os setores da indústria, comércio, e governo [2]. Não é diferente no âmbito das igrejas cristãs. Pastores e líderes vêm utilizando a tecnologia para propósitos espirituais, como aconselhamento, pesquisa e reflexão, e na preparação e apresentação de seus sermões [3].

As igrejas cristãs podem ser definidas como a reunião de pessoas que professam a mesma fé e se reúnem periodicamente, sob a liderança de oficiais. No entanto, existem diferentes fatores que influenciam a maneira como a igreja se organiza, e suas necessidades. Dentre estes fatores, podemos citar a forma de governo [4], tamanho e localização, por exemplo. O impacto destes fatores no projeto da LPS é detalhado no Capítulo 3. Portanto, concluímos que o domínio de gerenciamento de igrejas cristãs mostra-se adequado à idéia de linhas de produtos de software.

1.1 Objetivos

O principal objetivo do trabalho é o desenvolvimento de uma LPS voltada para o gerenciamento de igrejas cristãs.

Esta linha deve englobar todos os aspectos relacionados ao gerenciamento de uma igreja, procurando facilitar a integração e compartilhamento de dados entre os setores da igreja. Os produtos gerados por esta linha devem ser adequados à estrutura da organização alvo.

Como objetivos específicos do trabalho, podemos destacar:

- Instanciar um processo de desenvolvimento de LPS;
- Estudar técnicas de implementação de variabilidade em LPS, visando a utilização das técnicas mais adequadas, de acordo com o tipo de variabilidade;
- Desenvolver um gerador de produtos para a LPS;
- Gerar um produto a partir dos ativos da LPS, e realizar uma avaliação deste, implantando-o em uma igreja existente.

1.2 Estrutura do trabalho

Esta monografia está organizada em Capítulos e Apêndices. A seguir será detalhado o conteúdo de cada parte:

O Capítulo 2 discute a fundamentação teórica necessária para o entendimento do trabalho. Detalha o conceito de linhas de produtos de software, o método de desenvolvimento utilizado, e as tecnologias empregadas na construção da LPS.

O Capítulo 3 introduz a linha de produtos de software Ligo. O nome Ligo vem do latim e significa unir, ligar. Procura ilustrar o conceito de LPS, onde unimos diversos ativos para produzir produtos. Algumas considerações sobre o domínio da aplicação são feitas e depois é detalhada a fase de modelagem de requisitos da linha.

O Capítulo 4 descreve as fases de análise e projeto de desenvolvimento da linha. Também é descrito o processo de geração semi-automática de produtos da linha.

O Capítulo 5 conclui o trabalho, detalhando as principais contribuições referentes a este trabalho, algumas dificuldades e limitações ocorridas durante o desenvolvimento e possíveis trabalhos futuros.

O Apêndice A apresenta o caso de uso Adicionar Pessoa, escrito durante a fase de modelagem de requisitos.

O Apêndice B contém telas do processo de geração de produto, bem como do produto gerado.

Capítulo 2

Fundamentação Teórica

Neste capítulo são descritos os conceitos de linhas de produtos de software, a metodologia de desenvolvimento adotada, e as tecnologias utilizadas para a realização do projeto.

2.1 Linhas de Produtos de Software

Reuso de software tem sido motivo constante e recorrente de pesquisas em engenharia de software. O foco destas pesquisas tem se concentrado em métodos, técnicas e ferramentas que permitam melhorias em termos de estimativas de custo, tempo de desenvolvimento e qualidade. A idéia chave do reuso é: desenvolver algo uma vez, e reutilizar várias vezes [5]. Desta forma, procura-se evitar que as atividades de desenvolvimento de software se repitam. No entanto, para tornar esta prática efetiva, faz-se necessário o planejamento deste reuso.

Além da questão do reuso, que demonstra o lado da produção de software, há de se levar em conta o aspecto do cliente, o consumidor de software. Os clientes desejam adquirir softwares personalizados, mas sem que isso acarrete em um custo alto. Os custos de desenvolvimento de software personalizado são altos, em contraste com a produção de software em massa, que diminui os custos, porém limita a personalização do software de acordo com as necessidades do cliente.

A abordagem de linhas de produtos de software (LPS), também conhecida como famílias de produtos de software, procura estabelecer o reuso sistemático de software, assim como a customização em massa. Por reuso sistemático, entende-se o reuso planejado de componentes, reutilizando ativos base dentre os produtos [6]. A customização em massa diz respeito à produção em larga escala de produtos customizados às necessidades dos clientes [7]. A abordagem procura aproveitar-se da constatação de que poucos sistemas de software são de fato únicos. Muitas vezes, organizações desenvolvem produtos similares que pertencem a um domínio específico.

Uma LPS consiste em um conjunto de sistemas de software que compartilham funcionalidades em comum, e têm características individuais, ou seja, há variação entre produtos. O conceito é similar ao de linha de produção em outros domínios, por exemplo, nas indústrias automobilística e aérea. Uma definição dada por Clements e Northrop [1] é que uma LPS é "*um conjunto de sistemas de software que compartilham um conjunto gerenciável comum de características que satisfazem as necessidades específicas de um segmento de mercado particular e que são desenvolvidas de um conjunto de ativos base comum, de modo planejado*". Weiss e Lai [8] definem LPS como famílias de produtos projetadas para tirar vantagem de suas

características comuns e variações previstas. De acordo com Griss [9], uma LPS é um conjunto de produtos que compartilham um conjunto comum de requisitos, mas também exibem variações significativas nos requisitos.

O objetivo de uma LPS é minimizar o custo de desenvolvimento e manutenção de produtos de software que pertençam a um domínio comum. Para que seja possível o desenvolvimento customizado em massa a custo reduzido, é necessário utilizar uma base comum, chamada de plataforma, ou arquitetura da LPS. Esta arquitetura deve antecipar os possíveis produtos que podem ser gerados a partir de uma linha. Portanto, ela deve contemplar não apenas as características comuns aos produtos, mas também as possíveis variações e características opcionais. Esta é uma das principais diferenças entre a abordagem de LPS e as abordagens de desenvolvimento de um único sistema. O desenvolvimento baseado em LPS visa uma pluralidade de produtos que serão mantidos ao mesmo tempo, ao invés de um único produto que evolui no tempo. Outra diferença entre as abordagens é na aplicação de reuso. Enquanto em LPS, o reuso é planejado, no desenvolvimento de sistemas únicos, muitas vezes o reuso é feito de forma *ad hoc* ou oportunista. A prática de LPS encoraja escolhas e opções que procuram ser otimizadas, desde sua introdução, à aplicação em mais de um produto.

O desenvolvimento de uma LPS engloba duas etapas distintas [10]: Engenharia de Domínio (*Domain Engineering*) e Engenharia de Aplicação (*Application Engineering*), conforme observamos na Figura 1. Engenharia de Domínio, também citada como a etapa de desenvolvimento da linha de produtos, diz respeito ao desenvolvimento da arquitetura que servirá de base à LPS, e à definição das características comuns e variáveis da LPS. Esta arquitetura é composta pelos artefatos gerados, normalmente chamados de ativos base. Como exemplo destes ativos, podemos citar documentos de requisitos, bibliotecas de código, casos de testes, entre outros. Alguns destes ativos serão comuns a todos os produtos de uma LPS, enquanto outros serão opcionais ou alternativos.

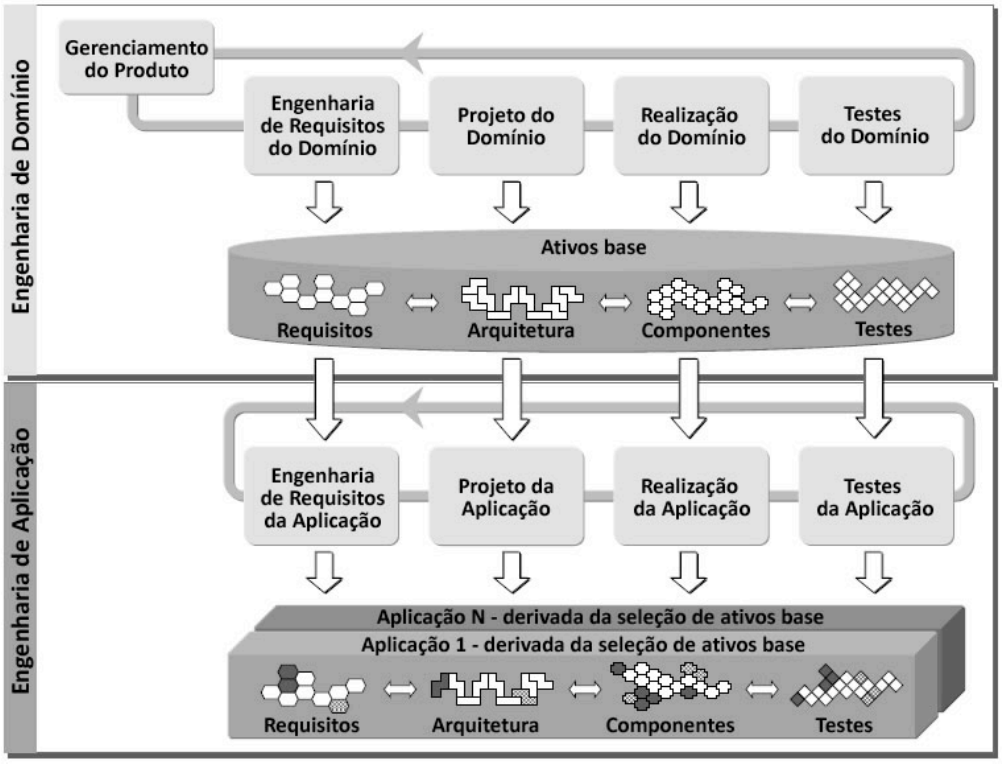

Figura 1. *Framework* de desenvolvimento de uma LPS. Adaptado de [10].

A etapa de Engenharia de Aplicação, também citada como desenvolvimento do produto, é responsável pela criação de produtos específicos a partir da LPS estabelecida na etapa anterior. O processo de geração de um produto de uma LPS é também referido como instanciação ou derivação de produto. Podemos observar um modelo geral de geração de produtos a partir dos ativos base de uma LPS na Figura 2. O processo de geração de produtos recebe como entradas os ativos base desenvolvidos, e uma configuração de produto. Esta configuração, também chamada de modelo de decisão, consiste na escolha dos ativos opcionais e variáveis, que irão compor o produto junto aos ativos comuns a todos os produtos. O mecanismo de geração então, ao receber estas entradas, faz a composição dos ativos, gerando como saída os produtos da LPS. O conjunto de todos os produtos possíveis de serem gerados a partir destas entradas determina o escopo da LPS. Os produtos de uma LPS também são referidos como membros da LPS.

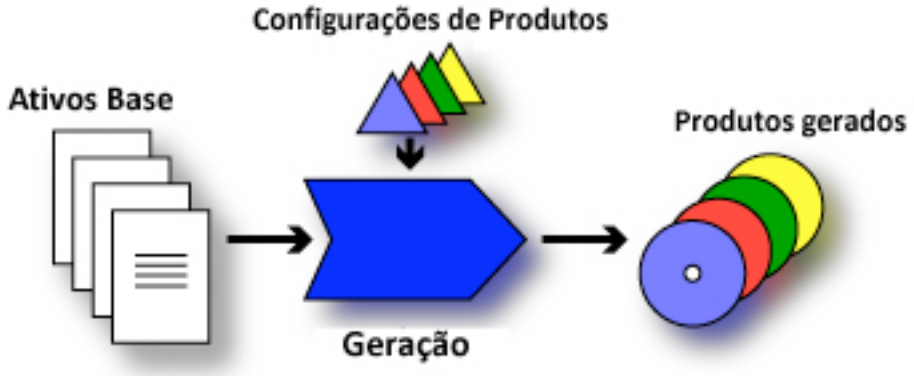

Figura 2. Modelo geral do processo de geração de produtos em uma LPS. Adaptado de [11].

Muitos são os benefícios trazidos por LPS [10]:

- Redução nos custos de desenvolvimento: Ao reutilizarmos os ativos-base nos produtos de uma LPS, há uma significativa redução de custos ao desenvolver produtos, em contraste com a abordagem de desenvolvimento de sistemas únicos. Investigações empíricas têm demonstrado que o investimento em uma LPS é compensado quando temos por volta de três produtos [8] em uma linha, pelo menos. É importante ressaltar que isto depende também da estratégia em que a LPS é adotada e iniciada em uma organização;
- Melhora na qualidade por conta do reuso: Os ativos de uma LPS são utilizados, analisados, e testados, em diversos produtos, o que gera melhores garantias de qualidade e confiabilidade;
- Redução de tempo para comercializar: Assim como os custos, o tempo para desenvolvimento de produtos é reduzido por conta do reuso;
- Especialização em um domínio particular: Aumenta a capacidade das organizações de atender a mudanças no mercado, permitindo a construção de novos produtos rapidamente, utilizando o conhecimento adquirido sobre o segmento de mercado em que atua;
- Redução no esforço de manutenção: A alteração de um artefato implica na propagação desta mudança para todos os produtos que o utilizam. A equipe de manutenção, em uma situação ideal, não modifica os produtos gerados, apenas os ativos que geram estes produtos. Portanto, caso haja algum erro em um ativo que é compartilhado por diversos produtos, a correção é feita no ativo, e os produtos são gerados novamente, já corrigidos.

Existem também alguns fatores a se considerar ao adotar a abordagem de LPS:

- Custos de desenvolvimento: O investimento para se iniciar uma LPS é superior ao de desenvolvimento convencional, por conta da necessidade de se criar ativos base para a formação da arquitetura. Existem abordagens que procuram minimizar este custo inicial, como a reativa e extrativa [11], em que ativos são desenvolvidos apenas quando necessários;
- Tempo para comercializar: Pelas mesmas razões do fator custos, o tempo para geração do primeiro produto geralmente é maior;
- Controle de versões: Deve haver cuidado para que as personalizações a um produto não ocorram fora da linha de produção, por exemplo, modificando o produto gerado ao invés dos ativos que o compõem;
- Apoio gerencial: As questões organizacionais são mais importantes do que o esperado [1]. O papel dos gestores é fundamental na prática de LPS, provendo treinamento, desenvolvendo a correta estrutura organizacional, criando e implementando um plano de adoção de LPS, lançando e institucionalizando a abordagem de maneira apropriada à organização, entre outros.

Alguns estudos têm sugerido que a abordagem de LPS estimula o reuso nas organizações [12], além de prover evidência empírica para a hipótese de que as organizações obtêm mais benefícios de reuso durante os estágios iniciais de desenvolvimento [13][14]. Diversas organizações, com diferentes características, têm relatado melhoras em produtividade e aumento dos ganhos com a adoção de LPS [1][10][15][16][17][18].

Por conta disso e dos benefícios citados, conclui-se que é válido a uma organização que desenvolve produtos similares a um domínio específico de mercado, investir tempo e dinheiro em métodos e processos de desenvolvimento de LPS. Existem diversas abordagens para o desenvolvimento de LPS. A abordagem adotada neste trabalho é detalhada na seção a seguir.

2.2 O método PLUS

PLUS (***P**roduct **L**ine **U**ML-Based **S**oftware Engineering*) é o método de desenvolvimento de LPS baseado em UML proposto por Hassan Gomaa [19]. O processo no qual o método se baseia é o *Evolutionary Software Product Line Engineering* (ESPLEP), um processo de desenvolvimento iterativo e orientado a objetos. O método é compatível com os modelos de desenvolvimento unificado [20] e espiral [21], por meio de adaptações ao processo geral descrito a seguir.

O ESPLEP é um processo de desenvolvimento que, baseado no *framework* apresentado na seção anterior, consiste em duas atividades principais, conforme ilustrado na Figura 3:

- Engenharia da Linha de Produtos (*Software product line engineering*): durante esta atividade, o objetivo é desenvolver os ativos da linha como um todo. Estes ativos são armazenados no repositório da linha de produtos de software;
- Engenharia da Aplicação (*Software application engineering*): nesta atividade, um membro da LPS é desenvolvido. Os ativos gerados pela atividade de engenharia de produto são utilizados como base. O desenvolvimento da aplicação torna-se uma derivação dos artefatos da linha, de acordo com os requisitos específicos do produto a ser gerado.

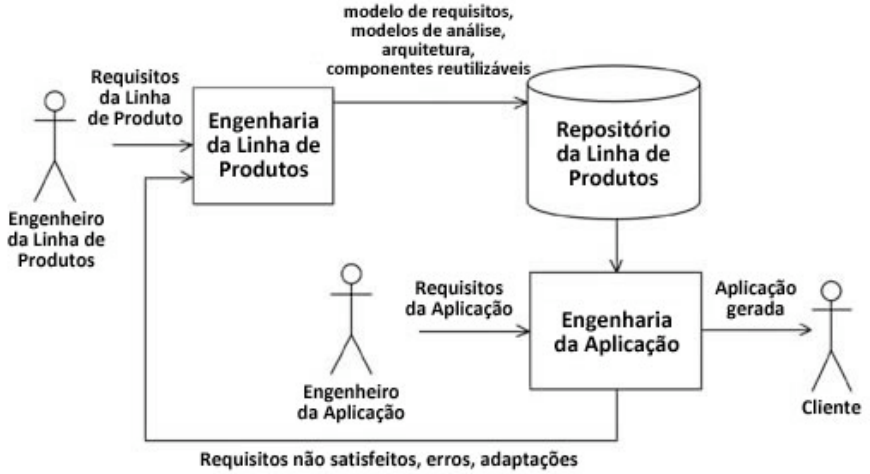

Figura 3. Processo de desenvolvimento ESPLEP. Adaptado de [19].

Esta abordagem elimina a distinção entre o desenvolvimento e manutenção de software, permitindo que o sistema evolua por meio de iterações. No entanto, o sistema deve ser projetado visando estas possíveis mudanças durante as iterações.

Existem duas estratégias de desenvolvimento de LPS. *Forward engineering* e *reverse engineering*. A estratégia *forward* é utilizada quando não há sistemas legado para guiar o desenvolvimento, em contraste à estratégia *reverse*, onde o desenvolvimento inicia com sistemas que já existem e são candidatos à inclusão na LPS.

A fase de Engenharia da Linha de Produtos pode ser dividida em três atividades principais: **(i)** Requisitos, onde definimos os requisitos funcionais da linha, e o que será comum e variável dentre os produtos. Caso os requisitos não sejam entendidos claramente, um protótipo descartável pode ser desenvolvido para esclarecimento; **(ii)** Análise, onde é feita a decomposição do problema, para melhor entendimento deste; **(iii)** Projeto e Desenvolvimento, que consiste na síntese da solução, contando com a implementação incremental de componentes. A cada iteração de desenvolvimento destes componentes, testes funcionais são realizados. Podemos visualizar as fases na Figura 4. Estas fases são adaptadas para se tornarem compatíveis com os processos unificado e espiral.

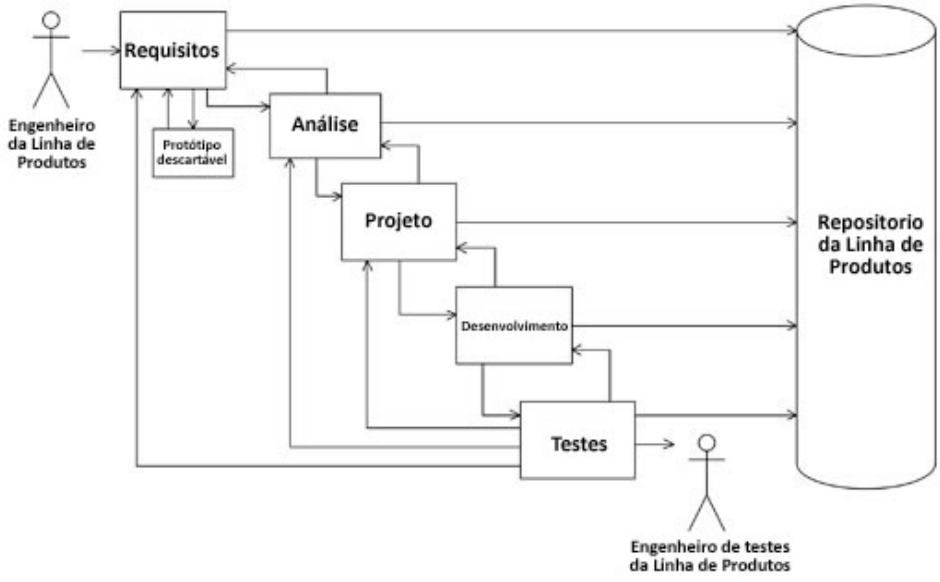

Figura 4. *Software Product Line Engineering* com ESPLEP. Adaptado de [19].

Nas seções a seguir, são detalhadas as atividades de desenvolvimento da fase de *Software Product Line Engineering* instanciadas para o desenvolvimento da LPS Ligo.

2.2.1 Requisitos

Durante esta fase, é desenvolvido o modelo de requisitos, que consiste do modelo de casos de uso e modelo de *features*. Esta fase é dividida em três atividades principais, detalhadas a seguir.

2.2.1.1 Análise de Escopo

O objetivo desta atividade é definir em alto nível, os possíveis sistemas que podem surgir a partir da linha. É recomendável que haja a participação de especialistas do domínio durante esta etapa, para melhor entendimento do problema.

Esta fase é importante para a decisão sobre a viabilidade de desenvolvimento de uma LPS. Durante a análise dos potenciais produtos que podem formar a linha, deve-se observar não apenas as semelhanças, como as diferenças entre os possíveis produtos.

É também a primeira tentativa de especificar os requisitos da linha, definindo o que vai pertencer à linha e o que estará fora.

2.2.1.2 Modelagem de *features*

A atividade de modelagem de *features* tem como finalidade descrever os requisitos da LPS da perspectiva do usuário final, por meio de um modelo de *features*.

Features podem ser descritas como conceitos [22], ou requisitos e características reutilizáveis de uma LPS [19]. O conceito de *feature* é utilizado para fazer distinção entre os possíveis produtos de uma LPS, definindo as funcionalidades comuns e variáveis de uma linha. Uma *feature* também pode se referir a requisitos não-funcionais.

Para que possamos modelar a variabilidade de uma linha, é preciso categorizar as *features*. *Features*, portanto, serão classificadas em **(i)** obrigatórias, o subconjunto de *features* compartilhados por todos os membros da linha; **(ii)** opcionais, *features* que são fornecidas por apenas alguns dos membros da linha; **(iii)** alternativas, grupo de *features* onde se deve fazer uma única escolha dentre as possíveis, as opções são mutuamente exclusivas; **(iv)** *or*, similar ao tipo alternativo, neste caso podemos fazer uma ou mais escolhas, dentre as possíveis.

Além destas categorias, também podemos adicionar regras de composição, para explicitar interdependência entre *features* [22]. Por exemplo, a regra *require* captura implicações entre *features*, definindo *features* como pré-requisito de outras. Outro exemplo é a regra *mutually-exclusive*, ou *conflict*, que coloca limitações nas combinações entre *features*. Esta regra é usada, em geral, quando desejamos excluir combinações de *features* que estejam em locais distintos da hierarquia do modelo.

Por exemplo [19], considere uma linha de produtos automobilística desenvolvida por uma empresa hipotética DSCars, que produz vários modelos de veículos. Suponha que é desenvolvido o modelo TCCarro, com versões sedã, esportiva e *station wagon*. Todas as versões compartilham o mesmo chassi, o que pode ser considerado uma *feature* comum, ou obrigatória. Existem *features* opcionais, como o pacote esportivo, câmbio automático e teto solar. A versão *station wagon* utiliza apenas a transmissão automática, um exemplo de regra *require*. Existem também *features* alternativas, como o tamanho do motor e a cor do veículo. É óbvio que todo veículo precisa de um motor. No entanto, o motor pode variar de tamanho. Portanto, a *feature* tamanho do motor tem as opções de 2 litros (padrão), 2,5 litros e 3 litros. No caso da *feature* cor do veículo, não há uma opção de cor padrão, portanto, uma cor deve sempre ser escolhida na

configuração de um produto da linha. Em certos casos, não escolher uma *feature* é também uma opção. Considere a possibilidade de o veículo ter bagageiro de teto. As opções alternativas da *feature* bagageiro de teto compreenderiam: bagageiro básico, bagageiro para pranchas, e bagageiro para bicicletas, embora seja possível não selecionar a *feature* bagageiro de teto.

A ferramenta utilizada para criar o modelo de *features* foi pure::variants *Community Edition* [23], desenvolvida pela empresa pure.systems na forma de um *plugin* do Eclipse [24]. A Tabela 1 explica a notação utilizada pela ferramenta na classificação de *features* e a Figura 11 mostra o modelo de *features* desenvolvido para a linha de produtos automobilística descrita anteriormente.

Tabela 1. Notação de símbolos utilizada na modelagem de *features* pela ferramenta pure::variants.

Símbolo	Explicação
	*Feature* obrigatória
	*Feature* opcional
	*Feature* alternativa
	Indicação de que uma *feature* requer outra *feature*

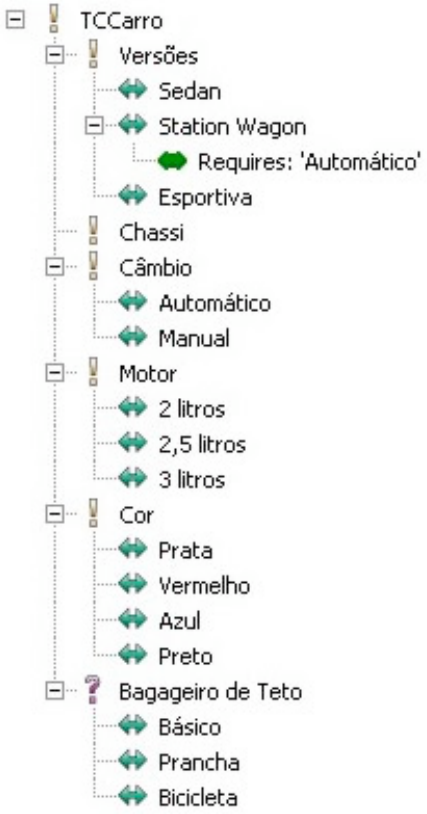

Figura 5. Modelo de *features* da linha de produtos TCCarro

Ao modelar as *features* de uma LPS, são levados em conta todos os produtos que podem ser gerados. Em contraste com o desenvolvimento de sistemas individuais, em que todas as *features* devem estar presentes, um produto da LPS vai compreender apenas um subconjunto das *features*.

2.2.1.3 Modelagem de casos de uso

A modelagem de casos de uso para LPS difere da abordagem tradicional, pois, nem todos os casos de uso serão utilizados por todos os produtos. Desta forma, é necessária a categorização dos casos de uso, de forma análoga ao que é feito com as *features*. Cada caso de uso terá um tipo definido de acordo com a sua característica.

A visualização destes diferentes tipos de casos de uso no modelo é feita por meio de estereótipos UML. Os estereótipos são um mecanismo padrão de extensão da linguagem UML, utilizados para distinguir entre os diferentes tipos de elementos da modelagem [25]. A representação visual do estereótipo é feita pelo seu nome entre os sinais "«" e "»", acima do nome do elemento a que este se aplica.

Os documentos de casos de uso são documentados de acordo com as seguintes seções:

1. *Nome do caso de uso*;
2. *Categoria de reuso*: Esta seção especifica se o caso de uso é obrigatório, opcional ou alternativo;
3. *Sumário*: Esta seção descreve brevemente o caso de uso;
4. *Atores*: Lista os atores participantes do caso de uso;
5. *Dependência*: Esta seção opcional descreve se o caso de uso depende de outros casos – se inclui ou estende outro caso de uso;
6. *Pré-condições*: Esta seção especifica as condições que devem ser satisfeitas para que o caso de uso seja iniciado;
7. *Descrição*: O conteúdo desta seção é uma descrição narrativa da seqüência de interações entre o ator e o sistema. O foco é nas respostas do sistema às interações, e não em como o sistema processa estas respostas;
8. *Alternativas*: Esta seção prove uma descrição de alternativas à seqüência principal descrita anteriormente. É possível haver mais de uma possibilidade de alternativa;
9. *Pontos de variação*: Esta seção define os locais na descrição do caso de uso onde funcionalidades diferentes podem ser introduzidas para membros distintos da linha de produtos. No caso de pequenas variações, como a inserção de um campo em um formulário, podemos identificar a(s) linha(s) da descrição de caso de uso onde serão introduzidas as novas funcionalidades. No caso de variações complexas, onde começam a surgir demasiadas alternativas à seqüência principal, podemos modelar estes pontos por meio de relacionamentos *include* e *extend*;
10. *Pós-condição*: Esta seção identifica a condição que será sempre verdadeira ao fim do caso de uso, supondo que a seqüência principal tenha ocorrido.

2.2.2 Análise

Durante esta fase, é dada ênfase ao entendimento do problema. Um objetivo importante é identificar os objetos e a informação trocada entre eles.

As atividades que compõem esta etapa são:

- Modelagem estática: Nesta atividade, é desenvolvido um modelo estático que define o relacionamento estrutural entre as classes de domínio do problema. Isto é feito por meio de um diagrama de classes *entity*. Durante esta atividade, também é feita a análise da dependência entre *features* e classes, onde, similar ao que acontece com *features* e casos de uso, classificamos classes entre obrigatórias, opcionais ou alternativas. Nesta análise, definimos os pontos de variação das classes, que podem

ser modelados por meio de classes abstratas, ou parametrização, onde uma classe tem parâmetros de configuração, em que são assinalados diferentes valores de acordo com o produto da linha;

- Modelagem dinâmica: Durante esta atividade, os casos de uso são descritos por meio de diagramas de seqüência e comunicação. Com estes diagramas, é possível demonstrar a interação entre objetos durante a execução de casos de uso. A variabilidade é modelada por meio de diagramas alternativos, onde é dado destaque à mudança na seqüência de mensagens entre os objetos.

2.2.3 Projeto e desenvolvimento

Na fase de projeto e desenvolvimento, o foco principal é em como sintetizar os artefatos descritos nas fases anteriores em uma solução. O modelo de análise, que ilustra o problema, é mapeado para o projeto, que se concentra na solução.

Com base nos modelos criados, os componentes que compõem a linha de produto são desenvolvidos, de forma incremental. A cada iteração, um subconjunto da linha de produtos é selecionado para ser implementado. A implementação inicia com os casos de uso obrigatórios, seguidos pelos opcionais e alternativos, de acordo com a seqüência estabelecida durante a fase de análise. Esta implementação consiste do projeto, codificação e teste dos componentes.

Um dos pontos chave de uma LPS é a variabilidade entre produtos. A representação explícita de variabilidade torna possível a geração de produtos específicos de uma LPS. Podemos identificar e categorizar a variabilidade de uma LPS por meio de modelos de *features*. Porém, esta variabilidade precisa ser implementada em código fonte. Diversos tipos de variabilidade podem ocorrer em um programa, como a adição, remoção, substituição e mudança de funcionalidades.

A variabilidade pode ser interna, isto é, escondida do usuário final, ou externa, visível ao usuário final do produto gerado. Como exemplo de variabilidade interna, podemos citar a escolha entre a utilização um protocolo de comunicação ao invés de outro ou a possibilidade de escolha do sistema de gerenciamento de banco de dados. As variabilidades internas geralmente são questões técnicas que não precisam ser consideradas pelo usuário final, como a alteração do banco de dados utilizado pela aplicação.

Existem diversas técnicas para a implementação de variabilidade em LPS [26]. Algumas destas técnicas são:

- **Agregação/delegação:** Permite que objetos encaminhem (deleguem) requisições que eles não conseguem satisfazer a objetos delegados. A variabilidade é alcançada colocando a funcionalidade obrigatória no objeto que delega e a variação no objeto delegado. É aplicável a *features* opcionais, porém, não é satisfatória para *features* alternativas, por conta da indireção em vários pontos de variação. Tipicamente resolvida em tempo de compilação, porém, é possível resolver em tempo de linkagem e até mesmo em tempo de execução, utilizando carga dinâmica de classes ou bibliotecas de ligações dinâmicas (DLLs);
- **Herança:** Esta técnica pode ser utilizada ao colocarmos funções básicas nas superclasses e funções especializadas nas filhas. A técnica mostra-se problemática com o crescimento na quantidade e tipos de variações, gerando árvores complexas de herança. Isso pode ser exacerbado em linguagens que implementam herança múltipla, como C++ e AspectJ;
- **Compilação Condicional:** Possibilita o controle sobre os segmentos de código a serem incluídos ou excluídos da compilação de um programa. Diretivas marcam os

pontos de variação no código. A funcionalidade desejada é selecionada pela definição dos símbolos condicionais apropriados. A compilação condicional é resolvida antes da compilação;

- **Parametrização (Arquivos de configuração):** A idéia é representar software reutilizável como bibliotecas de componentes parametrizados. O comportamento do componente é determinado pelos valores escolhidos para os parâmetros. Isto evita duplicação de código, centralizando decisões de projeto em um conjunto de variáveis. Por exemplo, uma pilha, na qual o tipo dos elementos é definido por um parâmetro. A parametrização pode melhorar o reuso em LPS, assim como facilitar o rastreamento das decisões de projeto. No entanto, centralizar código apenas definindo parâmetros é uma tarefa difícil, se não impossível, e a tarefa torna-se mais complexa à medida que os sistemas crescem;
- **Reflexão:** É a capacidade de um programa manipular como dados, elementos que representam o próprio programa durante sua execução. Essa técnica está relacionada fortemente à meta-programação, onde objetos em altos níveis de abstração representam entidades, como sistemas operacionais, processadores e linguagens de programação. Porém, código escrito com esta técnica é difícil de entender, depurar, e manter;
- **Orientação a aspectos:** A técnica de programação orientada a aspectos é descrita na seção a seguir. A variabilidade pode ser alcançada com a implementação das funcionalidades obrigatórias de maneira padrão, enquanto as variações são encapsuladas em aspectos. Os benefícios são acumulados, pois combinações de aspectos, bem como diferentes interpretações para um aspecto são facilmente realizáveis.

Como detalhado acima, cada técnica tem vantagens e desvantagens, não existe uma técnica que seja ideal em todos os casos possíveis de variabilidade. Portanto, é interessante que se faça combinações de técnicas, se possível, para aproveitar os pontos fortes das técnicas utilizadas.

2.3 Tecnologias Empregadas

Nesta seção apresentamos as tecnologias utilizadas para desenvolvimento da linha de produtos Ligo. A linguagem de programação escolhida para o desenvolvimento foi PHP. Esta escolha foi baseada nos requisitos do programa, que definiam o seu uso em ambiente web. Para o gerenciamento de variabilidade, utilizamos arquivos de configuração, na forma de classes estáticas e descrições XML, além de orientação a aspectos, para introduzir comportamentos de acordo com a configuração de *features* de um produto da linha.

2.3.1 PHP

PHP, um acrônimo recursivo para "*PHP: Hypertext Preprocessor*", é uma linguagem de *script* de código aberto embutida no HTML (*Hypertext Markup Language*) [27]. A inclusão de código se dá por meio de *tags* específicas, conforme vemos na Figura 6. O exemplo da figura imprime a string "*Exemplo do TCC*" entre as *tags* `<h2>` e `</h2>`. O objetivo da linguagem é permitir o desenvolvimento de aplicações Web de forma rápida.

PHP é geralmente usado em conjunto com um servidor web, como o Apache [28]. As requisições de scripts PHP são recebidas pelo servidor e são interpretadas pelo interpretador PHP.

Os resultados desta execução são retornados ao servidor Web que, por sua vez, os inclui no texto da página HTML como substituição ao programa original PHP e transmite a resposta ao cliente.

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Strict//EN" "http://www.w3.org/TR/html4/strict.dtd">
2  <html>
3  <head>
4  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
5  <title>Exemplo TCC</title>
6  </head>
7  <body>
8  <h2><?php echo 'Exemplo do TCC'; ?></h2>
9  </body>
10 </html>
```

Código PHP vem entre as tags <?php e ?>

Figura 6. Exemplo de código PHP embutido em um arquivo HTML

A linguagem é fortemente baseada em C, Java e Perl, com algumas influências da linguagem de processamento de texto *awk*, além de características específicas da própria linguagem. A linguagem inclui variáveis de vários tipos, *arrays*, funções, e dá suporte a orientação a objetos, com a maior parte dos mecanismos comuns à este paradigma. As variáveis de PHP são fracamente tipadas e sempre começam com o símbolo $, além de não requererem declaração, simplesmente passam a existir ao serem inicializadas. Ao ser inicializada, a variável tem o valor "*null*" ou "zero", até ser assinalada a um valor específico. Também existem variáveis de ambiente, que representam itens de dados cujo valor pode ser determinado pelo servidor, como o endereço IP do cliente que fez a requisição ou da parte dinâmica da URL requisitada, como valores enviados por meio de formulários HTML.

Um dos grandes pontos fortes de PHP é o mecanismo de extensão, onde podemos incluir rotinas à biblioteca PHP. Existem diversas extensões escritas em C, embutidas na distribuição padrão, que podem ser ativadas ou não, de acordo com a configuração de sua instalação. Como exemplo, podemos citar o suporte nativo a diversos bancos de dados, por exemplo, MySQL [29], manipulação de XML [30], processamento XSL [31]. Também é possível desenvolver extensões para otimizar o desempenho de execução de certos algoritmos.

Atualmente, é a linguagem de script mais popular dentre os servidores web, sendo utilizada em mais de 20 milhões de domínios [32].

2.3.2 Programação orientada a aspectos

O conceito de programação orientada a aspectos surgiu da necessidade de melhorar a modularidade dos programas, promovendo uma melhor separação de interesses – do inglês *concerns*, é um termo geral que diz respeito a requisitos, funcionais ou não, que são úteis ou precisam estar presentes nos sistemas [33]. O conceito de separação de interesses envolve quebrar um programa em partes que se sobreponham em funcionalidade o mínimo possível. Os paradigmas de programação oferecem mecanismos que dão suporte à separação e encapsulamento de interesses. Como exemplo, citamos procedimentos, pacotes, classes, métodos.

Alguns interesses não se encaixam nas formas de encapsulamento providas por paradigmas tradicionais, como o funcional ou orientação a objetos. Estes são os chamados interesses transversais (*crosscutting concerns*), ou ortogonais, atravessam o programa, em pontos diversos. Como exemplo destes interesses podemos citar *logging*, persistência, *debugging*, gerenciamento de exceções. Em alguns casos, uma porção considerável de código de uma operação ou classe não está relacionada com o interesse da classe ou operação e sim com esses interesses transversais. O código então se torna **(i)** disperso, pois interesses estão espalhados por meio de diversas partes do programa; **(ii)** entrelaçado, pois uma parte do programa pode envolver diversos

interesses transversais e a modificação de código torna-se difícil, pois deve-se entender e levar em conta todos estes interesses.

O paradigma de programação orientada a aspectos [34] (AOP – *Aspect Oriented Programming*), assim como a primeira linguagem orientada a aspectos, AspectJ [35], surgiu para complementar o paradigma de orientação a objetos. AOP provê os meios para separação de código que contém interesses transversais, modularizando-o em aspectos, elementos principais da linguagem. O termo interesse é muitas vezes substituído por aspecto, na literatura da área. AOP pode ser entendida como o desejo de fazer declarações quantificadas sobre o comportamento dos programas e ter estas quantificações aplicadas a programas escritos por programadores alheios a estas [36].

Existem alguns elementos específicos do paradigma que é necessário entender, para compreender o seu funcionamento:

- *Join Points* (pontos de junção): Pontos específicos da execução de um programa. Por exemplo, chamada a um método, execução de método, ou construção de um objeto. São locais, no código, em que podemos alterar o comportamento do programa, por meio dos aspectos. Compõem o elemento chave da AOP, pois as modificações semânticas serão baseadas nestes pontos;
- *Pointcuts* (conjuntos de pontos de junção): Permitem a seleção de conjuntos de *join points* de interesse. É possível coletar o contexto destes pontos. A composição é dada por meio de operadores, como && (*And*), || (*or*) e ! (*not*). Também podemos utilizar *wildcards*, como *, que é similar a função * de expressões regulares, ou seja, vai casar com zero ou mais caracteres, e +, que casa com todos os subtipos possíveis de uma classe;
- *Advice* (adendos): São elementos em que especificamos o comportamento que um aspecto observará, ao encontrar um *join point*. Este adendo pode se dar antes (*before*), depois (*after*) ou ao redor de (*around*) um ponto de junção;
- *Inter-type declarations* (Declarações intertipos): Declarações de atributos e métodos a serem inseridos em classes, no código resultante.

Estes elementos combinados, na maioria das abordagens, compõem um aspecto, cujo objetivo é encapsular todas as instruções relativas a um interesse. Podemos observar um exemplo de aspecto na Figura 7, com o aspecto `PointAssertions`. Verificamos a presença de *inter-type declarations*, na introdução de novos métodos na classe `Point`, que fazem verificações (*assertions*) sobre valores passados como argumentos. Também é possível observar a presença de *advice*, interceptando o código nos *join points* de chamada dos métodos `setX()` e `setY()`. O código presente dentro da declaração do *advice*, neste caso, será executado antes do *join point* definido. Ao observar o código da função `main()` de `Point`, vemos que a execução de `p.setY(333)` resulta na impressão da mensagem de erro na tela.

Após a identificação e decomposição dos interesses e implementação dos aspectos, é necessário fazer a recomposição aspectual. De alguma maneira, o comportamento do programa final deve compreender o comportamento original combinado com o comportamento definido nos aspectos. Este processo, chamado *weaving*, faz a combinação de código e pode ocorrer tanto de forma estática, quanto dinâmica.

```
class Point  {
    int x, y;

    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    public static void main(String[] args) {
        Point p = new Point();
        p.setX(3); p.setY(333);
    }
}

aspect PointAssertions {   Inter-type declarations
    private boolean Point.assertX(int x) {
        return (x <= 100 && x >= 0);
    }
    private boolean Point.assertY(int y) {
        return (y <= 100 && y >= 0);
    }
                                                    Advices
    before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
        if (!p.assertX(x)) {
            System.out.println("Illegal value for x"); return;
        }
    }                          Pointcut                          Join point
    before(Point p, int y): target(p) && args(y) && call(void setY(int)) {
        if (!p.assertY(y)) {
            System.out.println("Illegal value for y"); return;
        }
    }
}
```

Figura 7. Implementação de uma classe Java e um aspecto em AspectJ [35]

2.3.3 *phpAspect*

phpAspect é uma extensão à linguagem PHP que implementa programação orientada a aspectos [37]. A sintaxe do interpretador PHP é estendida, de maneira a introduzir uma nova entidade chamada aspecto. Os aspectos devem ser escritos em arquivos com a extensão *.aspect.php*.

O compilador *phpAspect* insere o código especificado nos aspectos no código fonte PHP. O processo de *weaving* é estático, acontecendo antes da execução do código no servidor, e se baseia nos geradores de analisadores léxico (Lex) e sintático (Yacc) [38] para gerar árvores sintáticas em XML, para o código PHP e os aspectos. Transformações XSL (*Extensible Stylesheet Language*) [31] são usadas para realizar a transformação de código utilizando estas árvores em XML. O código PHP resultante pode ser executado com qualquer versão de PHP 5. O processo de *weaving* é ilustrado na Figura 8.

Figura 8. phpAspect *weaving chain* [37]

Os aspectos são entidades de primeira classe durante a execução do programa e são representados por classes com atributos e métodos. Todos os aspectos são instanciados, com um padrão *singleton* quando o programa é inicialmente executado.

A sintaxe de *phpAspect* é similar à de AspectJ. Contém os *join points* tradicionais, como chamada e execução de método, construção de classes. Um exemplo de aspecto em phpAspect é demonstrado na Figura 9. Este exemplo é equivalente ao demonstrado na Figura 7, mas em código *phpAspect*. O exemplo insere os métodos `assertX` e `assertY` à classe `Point`, além de executar verificações antes da chamada dos métodos `setX` e `setY`. Em um arquivo *.aspect.php*, qualquer código localizado fora da declaração do aspecto será ignorado.

```
<?php
aspect PointAssertions {
    private function Point::assertX($x) {
        return ($x <= 100 && $x >= 0);
    }

    private function Point::assertY($y) {
         return ($y <= 100 && $y >= 0);
    }

    before(): call(Point::setX(*)) {
        if (!$thisJoinPoint->getTarget()->assertX($thisJoinPoint->getArg(0))) {
            echo "Illegal value for x";
            exit;
        }
    }

    before(): call(Point::setY(*)) {
        if (!$thisJoinPoint->getTarget()->assertY($thisJoinPoint->getArg(0))) {
            echo "Illegal value for y";
            exit;
        }
    }
}
?>
```


Figura 9. Exemplo utilizado na Figura 7 em sintaxe *phpAspect*

2.3.3.1 Contribuições ao *phpAspect*

O projeto *phpAspect*, como a maior parte dos projetos em estágios iniciais de desenvolvimento, ainda não é estável, e apresentou diversos problemas durante o desenvolvimento deste trabalho. O elemento *inter-type declarations* protagonizou um dos problemas mais críticos encontrados. Especificamente, a inserção de atributos e constantes não funcionava.

Para a solução deste problema foi necessário estudar a linguagem XSL, para maior compreensão do funcionamento das transformações XSLT. Posteriormente, as folhas de estilo XSL referentes ao processamento dos elementos *inter-type declarations* foram alteradas, possibilitando então, a inserção de atributos e constantes em classes. O *weaver phpAspect* foi então alterado para acomodar estas modificações.

Outro problema encontrado foi com a utilização de acentos. A extensão Parse_Tree [39], responsável pela montagem da árvore sintática abstrata dos aspectos e códigos PHP, utiliza uma codificação de XML UTF8, que gerava erros ao utilizarmos acentos nos códigos fonte. Foi necessário utilizar funções de codificação e decodificação UTF8 no processo de *weaving* para contornar este problema. Outra solução possível seria alterar o código fonte da extensão e

recompilá-la, mas a solução acima mostrou-se funcional, portanto, para economizar tempo, foi utilizada.

2.3.4 XML

Extensible Markup Language (XML) é uma especificação de formato de texto derivada de SGML desenvolvida pelo *World Wide Web Consortium* (W3C) [30]. Originalmente, sua aplicação destinava-se a publicação eletrônica. Atualmente, XML tem um papel importante, sendo muitas vezes utilizado como padrão para a definição e comunicação de dados estruturados entre serviços Web, assim como em aplicações *offline*.

A sintaxe de XML, conforme podemos observar na Figura 10, é semelhante à de HTML (*Hypertext Markup Language*). Porém, HTML é uma linguagem de apresentação de documentos, enquanto XML é uma linguagem de descrição de dados. Outra diferença é relacionada ao uso de *tags*. Embora ambas as linguagens façam uso deste recurso, no HTML, só são relevantes as tags já existentes. Já nos arquivos XML, é possível definir as próprias tags, de acordo com a aplicação do documento, assim como podemos especificar regras como a ordem em que aparecem, como devem ser processadas e apresentadas. Essa flexibilidade é importante, pois torna XML uma meta-linguagem. Como exemplo, podemos citar aplicações como RSS [40], Atom [41], MathML [42], MusicXML [43]. É possível definir regras de validação (esquemas) em documentos no formato XML. Um exemplo de linguagem XML utilizada para tal fim é XML Schema [44].

O exemplo da Figura 10 descreve uma estrutura de dados para descrição de *features*. Podemos observar elementos como o nome e tipo da *feature*, assim como classes, aspectos e arquivos relativos à *feature* em questão.

```
1<?xml version="1.0" encoding="UTF-8"?>
2<feature>
3<name>familia</name>
4<type>opcional</type>
5<classes>
6    <class>Familia.php</class>
7</classes>
8<aspects>
9    <aspect>familia.aspect.php</aspect>
10</aspects>
11<files>
12    <file>familias.php</file>
13    <file>adiciona_familia.php</file>
14    <file>edita_familia.php</file>
15    <file>pessoa_familia.php</file>
16</files>
17</feature>
```

Figura 10. Exemplo de código XML

Capítulo 3

A linha de produtos Ligo

Neste capítulo é apresentada a linha de produtos de software Ligo. São discutidos alguns conceitos relacionados ao domínio da aplicação e é descrita a fase de modelagem de requisitos da linha.

3.1 Considerações Iniciais

Conforme visto no capítulo 1, temos acompanhado a crescente utilização de Tecnologia da Informação, buscando eficiência na realização de tarefas, nos setores da indústria, comércio e governo. Isto também tem ocorrido no âmbito das igrejas cristãs. Observamos a utilização da tecnologia por líderes, para propósitos espirituais, onde podemos destacar aconselhamento via e-mail, pesquisa, e preparação e apresentação de mensagens [3].

Igrejas cristãs podem ser caracterizadas como a reunião de pessoas que professam a mesma fé e se reúnem periodicamente, sob a liderança de oficiais. No entanto, existem diferentes formas de governo dentre as igrejas, de acordo com a sua denominação [4]. Denominação é o nome que se dá a um subgrupo de uma religião que opera sob um nome comum, e um conjunto comum de doutrinas e tradições. Dentre as formas de governo possíveis, podemos destacar três principais:

- Episcopal – Neste sistema, as igrejas são governadas por bispos, que têm autoridade sobre dioceses. Os bispos podem estar sujeitos à autoridade de oficiais de maior escalão, como arcebispos, cardeais e patriarcas. Até a Reforma, era o sistema predominante entre as igrejas cristãs. Como exemplo de igrejas que utilizam esta forma de governo, podemos citar as igrejas católicas, anglicanas, episcopais, e algumas luteranas e metodistas.
- Congregacional – Também chamado de independente, neste sistema, cada igreja local, ou congregação, é eclesiasticamente soberana, ou autônoma. Dentre as principais igrejas protestantes que utilizam o sistema, podemos destacar as congregacionais e batistas. Recentemente, têm-se observado um crescimento de igrejas não-denominacionais, que não se alinham formalmente a uma denominação estabelecida. Estas igrejas geralmente adotam esta forma de governo, por prover maior independência.
- Presbiteriana – Este sistema é caracterizado pelo governo por meio de assembléias de presbíteros. Consiste numa ordem crescente de conselhos. O menor conselho é o

> da igreja local, consistindo dos pastores (ministros docentes) e presbíteros (ministros leigos) eleitos pelos membros da igreja. Acima dos conselhos locais, se encontra o Presbitério, formado por representantes dos conselhos locais. O Sínodo, instância superior, é formado por representantes dos Presbitérios, e finalmente, a última instância decisória sobre a igreja é o Supremo Concílio. Esta forma de governo é principalmente utilizada nas igrejas presbiterianas.

É certo afirmar que todas as formas de governo utilizadas pelas igrejas são variações destas três formas descritas acima. A forma de governo utilizada por uma igreja define a política da igreja e as regras, por exemplo, sobre quais serão os cargos que os membros da igreja podem assumir e os pré-requisitos para poder assumi-lo.

Além da forma de governo, existem outros fatores que influenciam a maneira como a igreja se organiza, qual sua ênfase ministerial principal e suas demais necessidades. Como exemplo, podemos citar tamanho e localização das igrejas. Portanto, verificamos que, apesar da semelhança na definição, há certa diversidade entre igrejas, por conta dos fatores acima explicitados. Por exemplo, uma igreja com 500 membros tem, normalmente, mais atividades durante a semana do que uma igreja com 100 membros.

Diversas aplicações têm surgido para solucionar problemas específicos do gerenciamento de uma igreja, como o gerenciamento de grupos familiares [45], controle financeiro [46], cadastro de sermões e reflexões [47], gestão de ministério de música [48], entre outros. Apesar de algumas destas aplicações resolverem bem o problema a que se propõem, o uso de diversos sistemas em separado gera novos problemas, como a inconsistência e dificuldade de compartilhamento de dados entre as aplicações.

Faz-se necessária, portanto, a utilização de um sistema que integre os dados e processos relativos ao gerenciamento da igreja. Tais sistemas são conhecidos como *Church Management Systems* (ChMS). No Brasil, existem poucas opções de sistemas ChMS, onde destacamos os produtos Church Tradicional [49] e SIGI [50] como principais expoentes do mercado.

A proposta da linha de produtos Ligo é atender a estas demandas, gerando produtos personalizados, visando satisfazer às necessidades individuais da igreja alvo, usuária final do produto gerado.

3.2 Modelagem de Requisitos

Conforme visto no capítulo anterior, a fase de modelagem de requisitos consiste de três atividades principais: **(i)** Análise de escopo; **(ii)** Modelagem de *features*; **(iii)** Modelagem de casos de uso. Esta etapa define os requisitos funcionais da linha, em particular, o que os produtos terão em comum, e o que será variável.

3.2.1 Análise de escopo

Durante esta atividade, foram realizadas as seguintes tarefas: **(i)** avaliações de ChMSs existentes, procurando identificar possíveis funcionalidades, além de observar como algumas funcionalidades foram implementadas; **(ii)** entrevistas com pastores e líderes de igrejas locais, visando maior entendimento das necessidades destes que seriam os principais usuários dos produtos gerados pela linha. Foram entrevistadas pessoas ligadas à liderança de igrejas pertencentes a três denominações que representam as formas de governo citadas na seção anterior (Batista, Episcopal e Presbiteriana) e pertencentes a igrejas de tamanho e localização distintas também; **(iii)** estudo sobre as formas de governo eclesiástico, procurando fazer a distinção de

organização entre as diversas igrejas estudadas, e observar como isso se reflete nos requisitos da linha de produto.

Esta atividade serviu para a solidificação do conhecimento sobre o domínio, criando um maior entendimento das necessidades reais de utilização do sistema. O resultado da atividade foi a definição de que a linha compreenderá produtos para cada uma das denominações citadas, assim como a possibilidade de personalização de configurações. Esta personalização faz-se necessária, de acordo com a observação de que igrejas constituem um domínio bastante variável, conforme explicado na seção anterior.

3.2.2 Modelagem de *features*

A atividade de modelagem de *features* tem como finalidade descrever os requisitos da LPS da perspectiva do usuário final, por meio de um *feature model*. No modelo de *features*, foram incluídos apenas os requisitos funcionais. No entanto, conforme visto no Capítulo 2, *features* também podem representar requisitos não-funcionais.

Uma breve explicação de cada *feature* segue abaixo:

- **Tipo Igreja** (*alternativa*): Especifica se o produto irá gerenciar apenas uma igreja (padrão), ou um conjunto de igrejas;
- **Denominação** (*alternativa*): Especifica a denominação da igreja utilizadora do produto. Esta escolha se refletirá na adaptação de alguns termos específicos a cada denominação, bem como as configurações de classificação de membros e cargos. Também é possível personalizar estas configurações. De forma similar ao exemplo da *feature* cor na linha de produtos de veículos no Capítulo 2, esta *feature* é alternativa, e não tem nenhuma seleção padrão, porém, uma denominação deve ser escolhida dentre as opções possíveis;
- **Pessoas** (*obrigatória*): Agrupa as funções relacionadas ao gerenciamento das informações relacionadas a uma pessoa ligada à igreja;
- **Mala Direta** (*obrigatória*): Possibilita o envio de e-mail em massa para a base de dados do produto. As configurações possíveis para envio de e-mails podem ser alteradas de acordo com a seleção de *features* opcionais. Por exemplo, ao incluirmos a *feature* famílias, podemos enviar e-mail para famílias em específico e assim por diante;
- **Relatórios** (*obrigatória*): Possibilita a geração de relatórios a partir da base de dados do produto. Assim como a *feature* Mailing, a configuração de relatórios pode ser alterada de acordo com a seleção de *features* opcionais.
- **Famílias** (*opcional*): Agrupa as funções de gerenciamento das informações de famílias ligadas à igreja;
- **Grupos** (*opcional*): Agrupa as funções de gerenciamento das informações de grupos pertencentes à igreja. Como exemplos de grupos podemos citar o grupo de música, pequenos grupos que se reúnem em casas, grupos de jovens, entre outros. A definição de um grupo envolve a escolha de um tipo e a criação de papéis possíveis dentro de um grupo, bem como a possibilidade de definição de propriedades específicas para um grupo, no formato de campos adicionais. Esta definição foi generalizada, mas podemos ter especializações de grupos, como pequenos grupos, e ministérios, com configurações específicas pré-definidas;
- **Finanças** (*opcional*): Agrupa as funções de gerenciamento das informações de movimentações financeiras ligadas à igreja. As movimentações podem ou não ser associadas a uma pessoa;

- ***Website*** (*opcional*): Transforma o ChMS em um *Content Management System* (CMS), onde torna-se possível também gerenciar um *website* da igreja;
- **Eventos** (*opcional*): Agrupa as funções de gerenciamento das informações de eventos da igreja. A *feature* filha opcional Registro *online*, possibilita o registro via site em eventos da igreja, o que torna necessária a presença da *feature website*;
- **Cursos** (*opcional*): Agrupa as funções de gerenciamento das informações de cursos ministrados por uma igreja. A *feature* filha opcional Aulas *online*, possibilita a disponibilização via site de material de aulas de um curso, o que torna necessária a presença da *feature website*;
- **Rede Ministerial** (*opcional*): É um programa desenvolvido pela *Network Ministries International* [51], que visa a descoberta dos dons e talentos dos participantes de uma igreja, por meio de exercícios e questionários.

O modelo de *features* gerado pode ser visto na Figura 11. Estas *features* compreendem uma lista não-exaustiva de possibilidades para um sistema de gerenciamento de igrejas cristãs. Conforme já citado acima, é possível, a partir destas *features*, gerar outras especializadas em um determinado foco. A evolução da LPS ocorre com a adição e remoção de *features*, alteração da lógica e reclassificação de *features* já existentes, possibilitando a inclusão de novos produtos à LPS, assim também como a eliminação de produtos que não atendam a demandas do mercado.

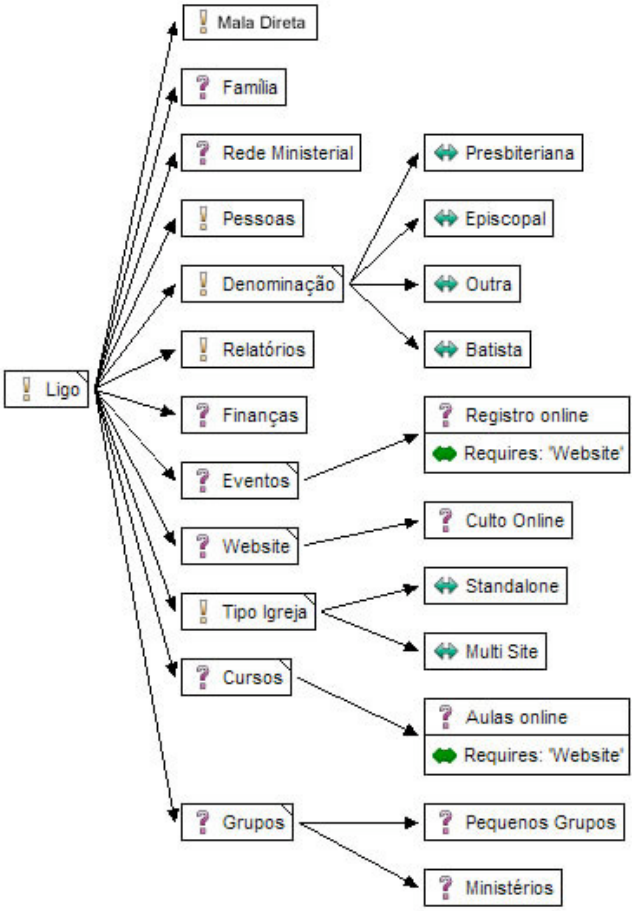

Figura 11. Modelo de *features* da LPS Ligo gerado com o auxílio da ferramenta pure::variants

3.2.3 Modelagem de casos de uso

A modelagem de casos de uso, descrita no capítulo 2, consiste na descrição e montagem dos diagramas de casos de uso. Os casos de uso são classificados e categorizados com o auxílio de estereótipos, e foram divididos de acordo com os atores. Foram identificados 4 atores, que são detalhados, com seus respectivos diagramas de casos de uso, a seguir.

O ator Secretária modela a pessoa que é responsável pelas atividades administrativas de uma igreja. Este ator tem acesso à maior parte dos casos de uso da LPS Ligo, conforme visualizamos na Figura 12. Os casos de uso que o ator executa são referentes às *features* Pessoa, Famílias, Grupos, Eventos, Finanças, Tipo de Igreja, Mala Direta, Relatórios.

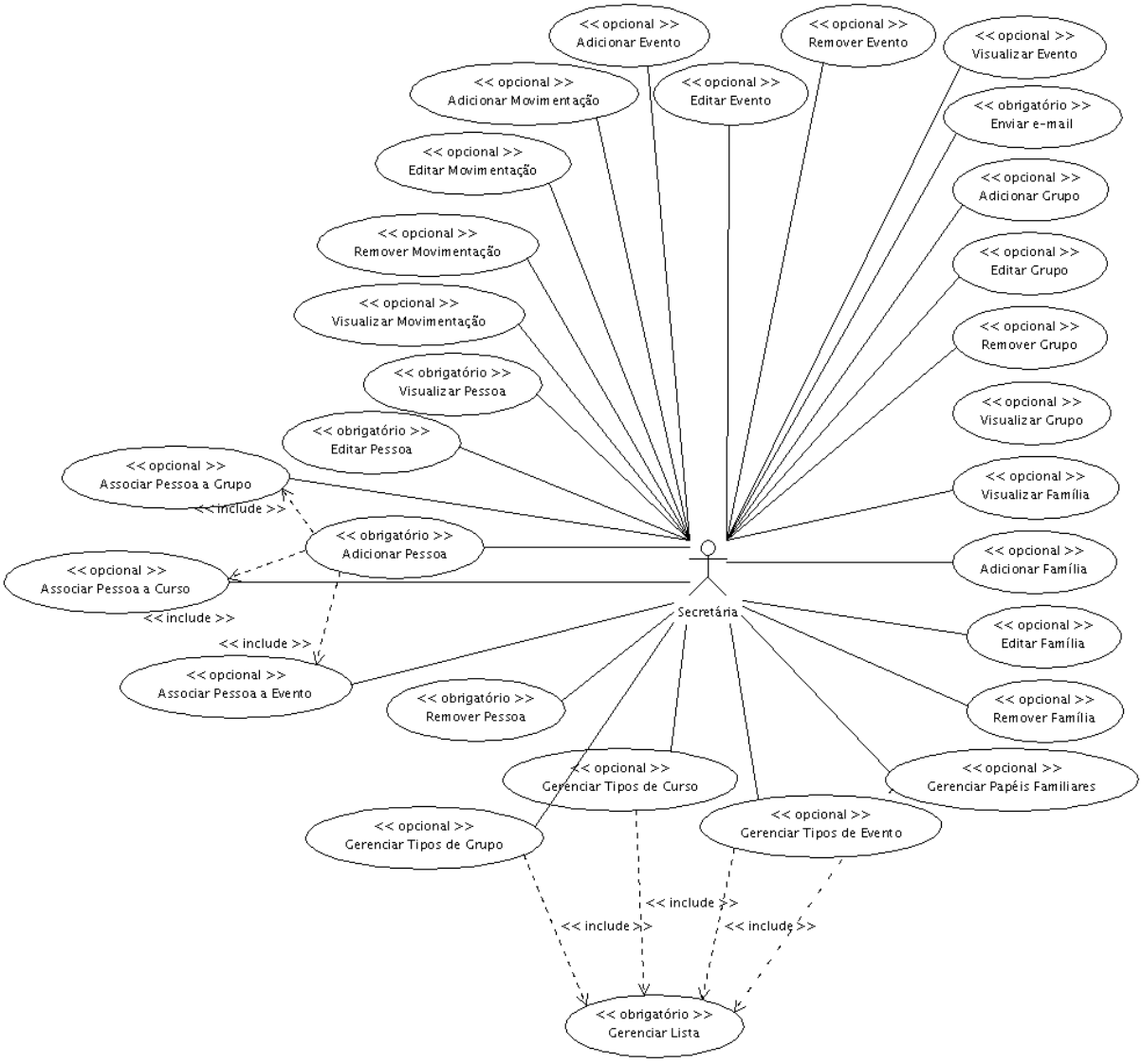

Figura 12. Diagrama de casos de uso da LPS Ligo referente ao ator Secretária

O ator Pastor modela a pessoa que é responsável pela liderança e supervisão de uma igreja. O papel de um pastor em grandes igrejas é até comparável ao de um CEO (*Chief Executive Officer*) [52], pois lidera equipes, faz o acompanhamento de membros da igreja, planejamentos anuais, direciona a visão da igreja, dentre outras atividades. Este ator, portanto, está associado aos casos de uso relacionados à visualização das informações e não necessita realizar os casos de uso

operacionais, como adicionar/editar/remover elementos. A exceção se faz aos casos relacionados à *feature* Cursos, pois normalmente, o Pastor é o responsável pela concepção e planejamento dos cursos de uma igreja. Os casos de uso que o ator executa são referentes às *features*: Pessoa, Famílias, Grupos, Eventos, Cursos, Tipo de Igreja, Relatórios, Mailing.

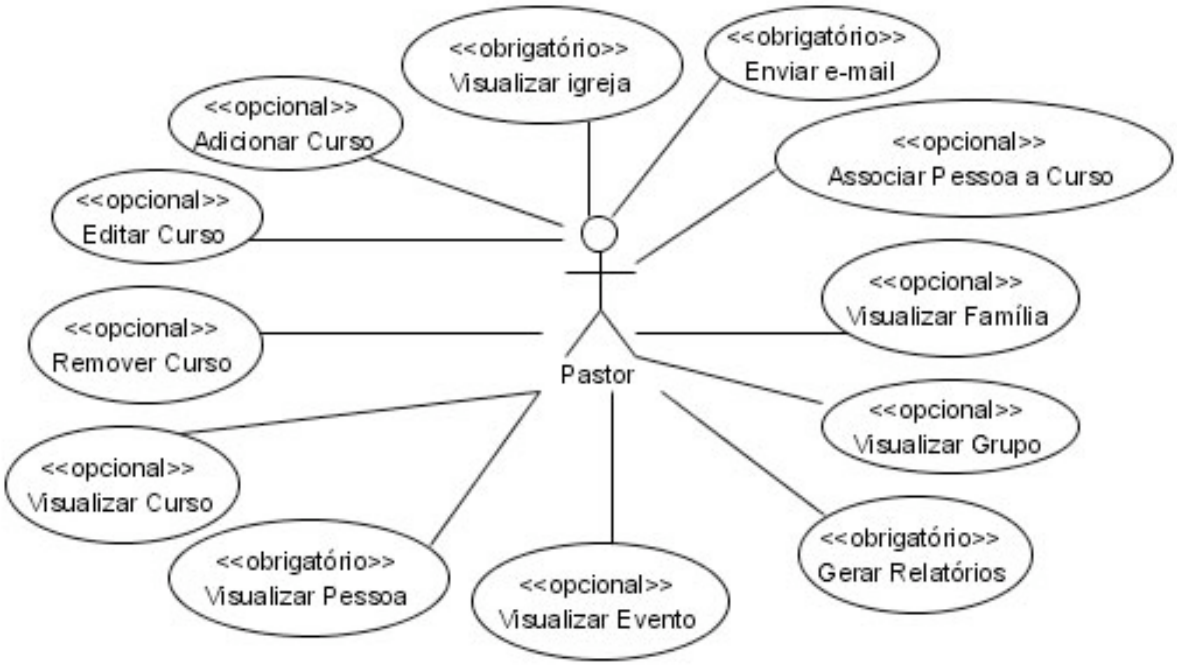

Figura 13. Diagrama de casos de uso da LPS Ligo referente ao ator Pastor

O ator Tesoureiro modela a pessoa responsável pelo gerenciamento de finanças de uma igreja. O papel desta pessoa é supervisionar e documentar as movimentações financeiras da igreja. Este ator, portanto, está associado apenas aos caso de usos relacionados à *feature* Finanças e pode gerar relatórios, mas apenas do tipo financeiro.

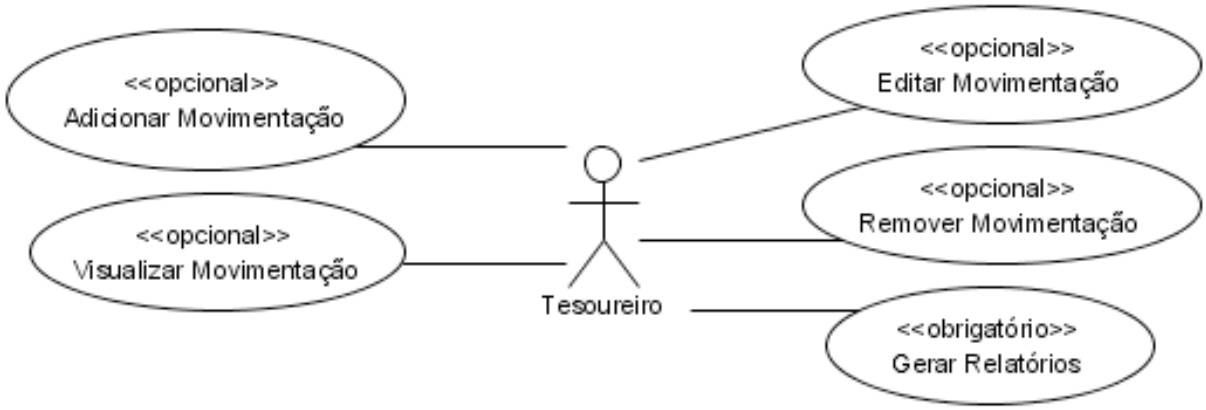

Figura 14. Diagrama de casos de uso da LPS Ligo referente ao ator Tesoureiro

O ator Membro modela a pessoa que é envolvida em uma igreja e não se encaixa em uma das definições anteriores. Inicialmente, este ator pode apenas editar as informações relativas à sua

pessoa e realizar o questionário Rede Ministerial, caso esteja presente na configuração do produto. No entanto, a este ator podem ser concedidas permissões para executar outros casos de uso. Por exemplo, um Membro pode ser líder de um grupo, portanto poderá editar informações relacionadas àquele grupo específico.

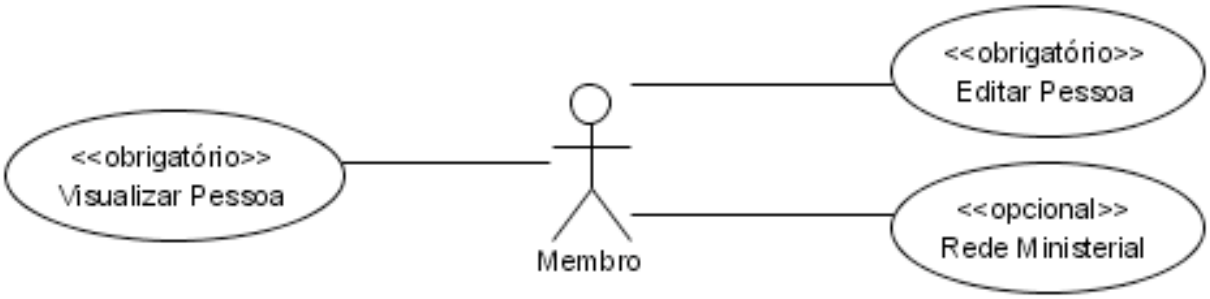

Figura 15. Diagrama de casos de uso da LPS Ligo referente ao ator Membro

A descrição do caso de uso Adicionar Pessoa encontra-se no Apêndice A. Na descrição, podemos observar o ponto de variação introduzido com a adição da *feature* Família.

3.2.4 Relação entre *features* e casos de uso

Podemos representar os relacionamentos entre *features* e casos de uso por meio de tabelas. As Tabelas 2 e 3 informam a notação de símbolos utilizada para identificar *features* e casos de uso na Tabela 4. Esta representação é importante para visualização dos casos de uso que a seleção de uma *feature* introduz em um produto, bem como o impacto que esta *feature* terá nos casos de uso.

Como exemplo, podemos analisar a *feature* Grupos. Ao observarmos a tabela, verificamos que esta *feature* impacta não apenas os casos de uso relacionados ao gerenciamento de grupos. A adição da *feature* tem impacto também nos casos de uso relacionados a pessoas, pois grupos são formados por pessoas, assim como os casos de uso de enviar e-mail e gerar relatórios. Podemos analisar também o caso de uso Gerar Relatórios. Este caso de uso sofre impacto de diversas *features*. Por exemplo, caso a *feature* Finanças esteja presente em um produto, poderemos gerar relatórios financeiros.

Tabela 2. Notação de símbolos de *features* utilizada na Tabela 4.

Símbolo	***Feature***
F_2	Mala Direta
F_3	Pessoas
F_4	Relatórios
F_5	Tipo Igreja
F_8	Denominação
F_13	Eventos
F_14	Famílias
F_15	Cursos
F_16	Grupos
F_17	Finanças
F_19	Rede Ministerial

Tabela 3. Notação de símbolos de *casos de uso* utilizada na Tabela 4.

Símbolo	Caso de Uso
UC_1	Adicionar Pessoa
UC_2	Editar Pessoa
UC_3	Remover Pessoa
UC_4	Visualizar Pessoa
UC_5	Adicionar Grupo
UC_6	Editar Grupo
UC_7	Remover Grupo
UC_8	Visualizar Grupo
UC_9	Adicionar Família
UC_10	Editar Família
UC_11	Remover Família
UC_12	Visualizar Família
UC_13	Adicionar Curso
UC_14	Editar Curso
UC_15	Remover Curso
UC_16	Visualizar Curso
UC_17	Adicionar Evento
UC_18	Editar Evento
UC_19	Remover Evento
UC_20	Visualizar Evento
UC_21	Adicionar Movimentação
UC_22	Editar Movimentação
UC_23	Remover Movimentação
UC_24	Visualizar Movimentação
UC_25	Enviar e-mail
UC_26	Associar Pessoa a Curso
UC_27	Associar Pessoa a Grupo
UC_28	Associar Pessoa a Evento
UC_29	Gerenciar Papéis Familiares
UC_30	Gerenciar Lista
UC_31	Gerenciar Tipos de Curso
UC_32	Gerenciar Tipos de Grupo
UC_33	Gerenciar Tipos de Evento
UC_34	Gerenciar Tipos de Movimentação
UC_35	Realizar Rede Ministerial
UC_36	Editar Configurações Denominação
UC_37	Gerar Relatórios

Tabela 4. Representação tabular de relacionamentos entre *features* e casos de uso

	F_2	F_3	F_4	F_5	F_8	F_13	F_14	F_15	F_16	F_17	F_19
UC_1		X					X				
UC_2		X					X		X		
UC_3		X					X		X		
UC_4		X					X		X		
UC_5									X		
UC_6									X		
UC_7									X		
UC_8									X		
UC_9							X				
UC_10							X				
UC_11							X				
UC_12							X				
UC_13								X			
UC_14								X			
UC_15								X			
UC_16								X			
UC_17						X					
UC_18						X					
UC_19						X					
UC_20						X					
UC_21										X	
UC_22										X	
UC_23										X	
UC_24										X	
UC_25	X	X		X		X	X	X	X		
UC_26		X						X			
UC_27		X							X		
UC_28		X				X					
UC_29							X				
UC_30											
UC_31								X			
UC_32									X		
UC_33						X					
UC_34										X	
UC_35											X
UC_36					X						
UC_37		X	X	X		X	X	X	X	X	

Capítulo 4

Análise, Projeto e Desenvolvimento

Neste capítulo são descritas as fases de Análise, Projeto e Desenvolvimento da linha de produtos de software Ligo, bem como os artefatos resultantes. Também é descrito o processo de geração de produto da LPS Ligo.

4.1 Análise

A etapa de análise, conforme visto no Capítulo 2, enfatiza o entendimento do problema. Esta etapa foi dividida em duas atividades, que são descritas nas seções a seguir.

4.1.1 Modelagem estática

A modelagem estática descreve a estrutura estática da LPS em desenvolvimento. O papel desta atividade é expressivo na modelagem de LPS, pois esta é uma notação poderosa para capturar as características comuns e variáveis de uma LPS. O modelo estático pode ser utilizado para modelar as associações entre classes *entity*, que modelam entidades centradas em dados, similar ao desenvolvimento de sistemas únicos, assim como pode modelar as hierarquias utilizadas em modelos de LPS para famílias de sistemas. Esta modelagem de hierarquias utiliza classes *physical*, que modelam dispositivos físicos, e é útil para contextualizar o problema, principalmente em casos de desenvolvimento de software embarcado. Por este não ser o caso da LPS Ligo, o foco foi dado na modelagem de classes *entity* e suas dependência em relação às *features*.

Assim como aconteceu com as *features* e casos de uso, as classes também são classificadas entre obrigatórias, opcionais e alternativas. A estratégia utilizada para o desenvolvimento foi similar à do desenvolvimento da LPS, *forward evolutionary engineering*. O modelo estático das classes obrigatórias à LPS foi desenvolvido e depois evoluído com a inclusão de variações.

O modelo estático de classes *entity* da LPS Ligo pode ser visto na Figura 16. Foram utilizados estereótipos, assim como na modelagem de casos de uso, para classificação de reuso das classes. O estereótipo *entity* foi removido das classes para eliminar redundância pois todas as classes são *entity*. Pela aplicação ser centrada em dados, as classes demonstradas no modelo representam os dados que estão armazenados no banco de dados.

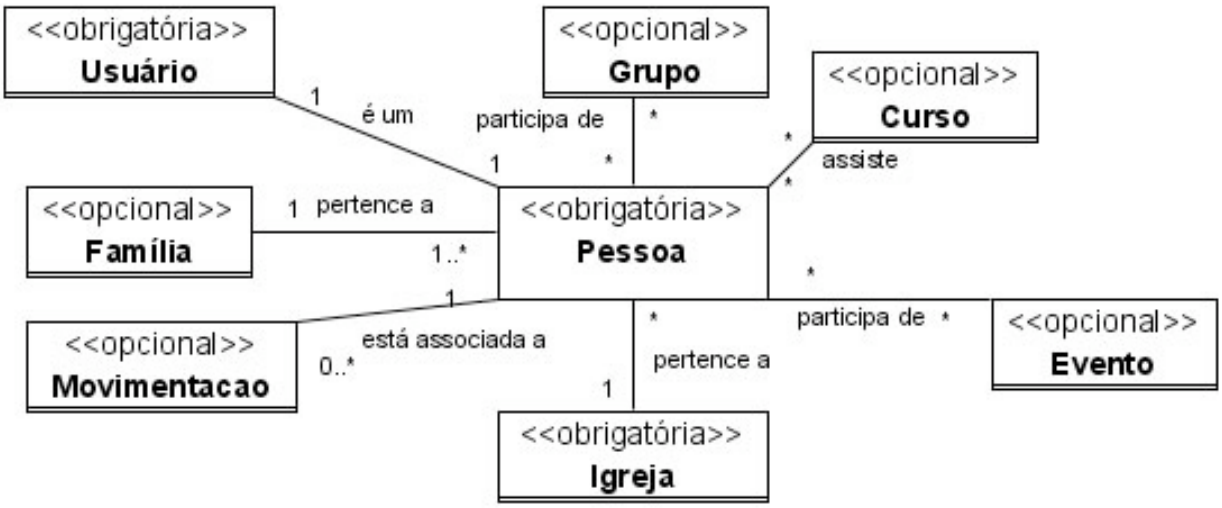

Figura 16. Diagrama de classes *entity* da LPS Ligo

É possível observar que as classes obrigatórias estarão presentes em todos os produtos gerados pela LPS, porém a presença de alguns de seus atributos ou métodos dependerá da escolha das *features*. A modelagem destes casos é feita através de parametrização de classes. Esta parametrização pode ser implementada de diversas maneiras, e é detalhada na seção de implementação de variabilidade. Com classes parametrizadas, uma classe da LPS terá parâmetros de configuração, que têm diferentes valores, entre membros da LPS. A vantagem desta abordagem é a simplificação, pois ao invés de gerarmos inúmeras classes adicionais, temos apenas uma classe parametrizada. Estes atributos e métodos serão incluídos ou modificados durante o processo de geração de um produto a partir dos ativos da LPS. Podemos observar os atributos das classes obrigatórias na Figura 17 e das classes opcionais na Figura 18. Ao lado dos atributos parametrizados, é colocada a restrição para que estes atributos estejam presentes. Neste caso, a inclusão da *feature* citada entre chaves.

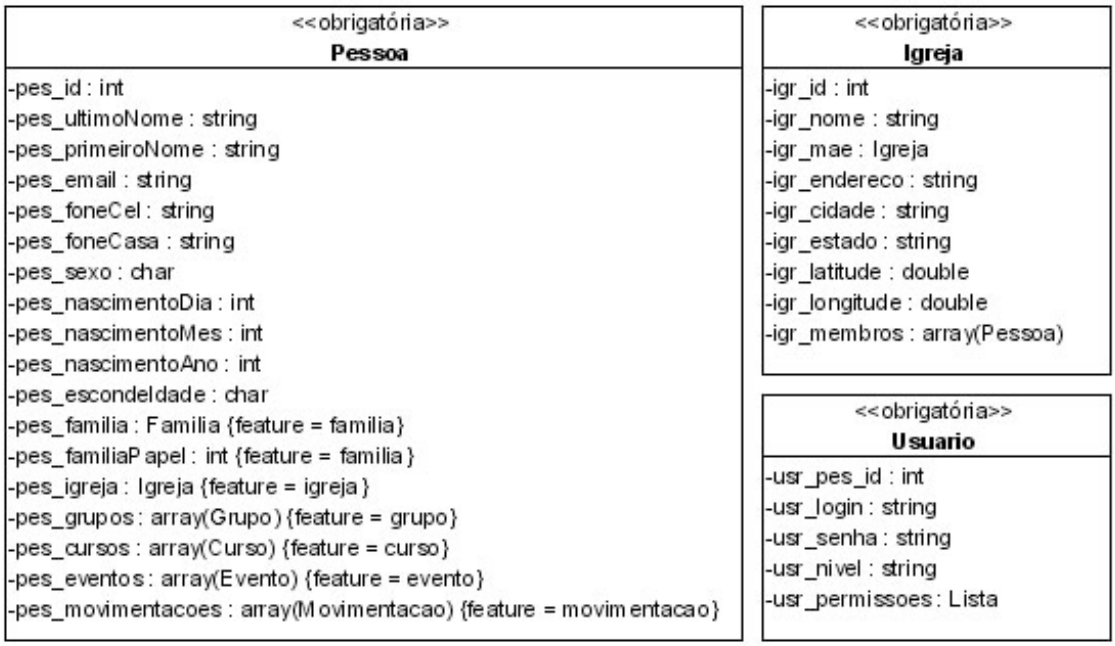

Figura 17. Classes *entity* obrigatórias da LPS Ligo

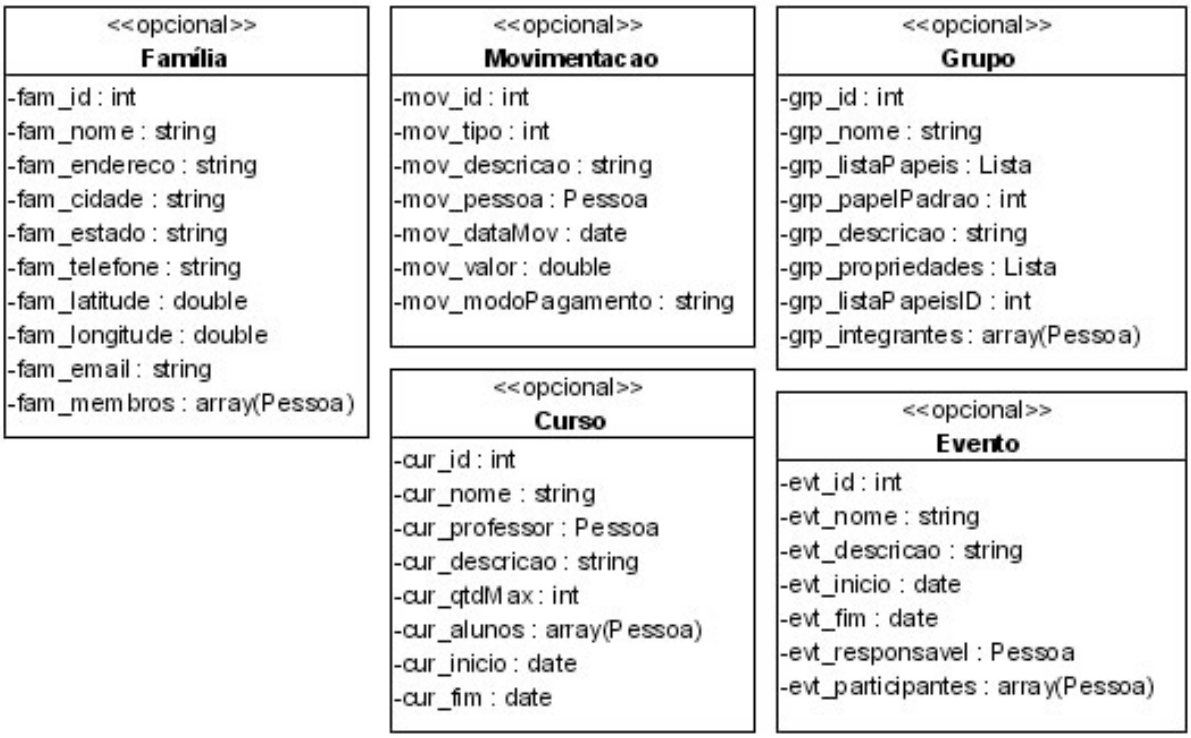

Figura 18. Classes *entity* opcionais da LPS Ligo

É importante ressaltar que as classes *entity* são apenas um dos tipos de classes da LPS. A modelagem estática utiliza este tipo de classe pois são elementos centrados em informação. Para podermos realizar a atividade seguinte, modelagem dinâmica, precisamos determinar que classes e objetos serão necessários para realizar os casos de uso. A identificação de objetos de software pode ser assistida por critérios de estruturação de objetos, que provêem direção na estruturação da aplicação em objetos. A idéia é categorizar classes e objetos pelos papéis que desempenham na aplicação. Os tipos de objetos que estarão presentes na LPS Ligo, foram divididos em:

- *Interfaces*: Objetos que se comunicam com elementos externos ao sistema, como interfaces com o usuário, sistemas externos e dispositivos externos;
- *Entities*: Objetos persistentes que armazenam informação. Estes objetos são instanciados de classes que seriam modeladas como entidades em modelos entidade-relacionamento;
- *Controle*: Objetos que provêem coordenação de coleção de objetos em casos de uso;
- *Lógica de negócio*: Objetos que contém detalhes da lógica da aplicação. Estes objetos são necessários quando é desejável esconder a lógica da aplicação separadamente dos dados sendo manipulados.

Estereótipos serão utilizados para descrever o tipo de cada classe nos diagramas que mostram a interação entre objetos.

4.1.2 Modelagem dinâmica

A modelagem dinâmica provê uma visão da LPS em que a seqüência e controle de ações são considerados, seja em um objeto, por meio de máquinas de estado finito, seja entre objetos, pela análise de interações entre os mesmos. Neste trabalho o foco foi mantido na interação entre objetos. A modelagem é baseada nos casos de uso desenvolvidos durante a modelagem de casos de uso e pode ser feita por meio de diagramas de comunicação ou diagramas de seqüência. Uma descrição narrativa da interação entre objetos acompanha o modelo. Por serem baseados nos casos de uso, os diagramas de interação são classificados em obrigatórios, opcionais ou alternativos.

A estratégia de desenvolvimento dos diagramas de interação é chamada evolucionária e inicia com o padrão *kernel first approach*. Os diagramas referentes aos casos de uso obrigatórios serão inicialmente desenvolvidos, seguidos dos opcionais e alternativos. Adicionalmente, com base na análise da dependência entre *features* e casos de uso (ver Tabela 4), as possíveis variações em um caso de uso são adicionadas ao diagrama.

A Figura 19 mostra o diagrama de seqüência para o caso de uso Editar Pessoa. Podemos observar a interação entre os diversos tipos de objetos descritos na seção anterior. O objeto `interfaceUsuario` representa a tela disponibilizada no navegador para o usuário, neste caso, com uma lista de pessoas. Ao clicarmos em editar pessoa, para uma pessoa hipotética, o objeto controlador `handlerPessoa` interage com os objetos `form`, responsável pela montagem do formulário HTML, e `pessoa`, objeto que representa a entidade Pessoa. Os dados da pessoa são obtidos do banco de dados utilizando o objeto `dbHandler`, que retorna as informações relativas à pessoa em questão. Com estas informações, é possível montar o formulário, via mensagem `setForm`, e gerar o código HTML do mesmo, com a mensagem `toHTML`. O formulário então é apresentado ao usuário no navegador.

O usuário pode então adicionar e editar os dados que acha necessário e após esta ação, clicar no botão salvar para efetivar as mudanças no banco de dados. O *handler* `handlerPessoa` então faz a validação dos dados, de acordo com as regras estabelecidas quando da geração do formulário, e caso não haja nenhum problema, o objeto `pessoa` é atualizado com os novos valores. Após a atualização do objeto, os valores são armazenados no banco de dados, e, na ausência de erros, é retornada a mensagem de sucesso ao usuário.

Com a adição da *feature* Famílias a um produto, o fluxo de mensagens é alterado e é necessário demonstrar esta mudança no diagrama. As mudanças ocorrem **(Figura 19.i)** no momento da obtenção dos dados de uma pessoa, em que também obtemos as informações que dizem respeito à família da pessoa; **(Figura 19.ii)** no momento de configuração do formulário, quando os campos relacionados com a *feature* Família são adicionados – família a que uma pessoa pertence, e o papel desempenhado nesta; **(Figura 19.iii)** ao atualizarmos as informações no objeto `Pessoa`, onde atualizamos os atributos da família de uma pessoa; **(Figura 19.iv)** finalmente, ao realizarmos a consulta de atualização das informações no banco de dados, atualizaremos também os campos da tabela de pessoas adicionados por conta da presença da *feature* Família.

O diagrama de seqüência alternativo do caso de uso Adicionar Pessoa, com a adição da *feature* Família, é demonstrado na Figura 20. As interações adicionais são destacadas com o seu respectivo número assinalado. A maneira como estas alterações são implementadas é discutida na seção a seguir.

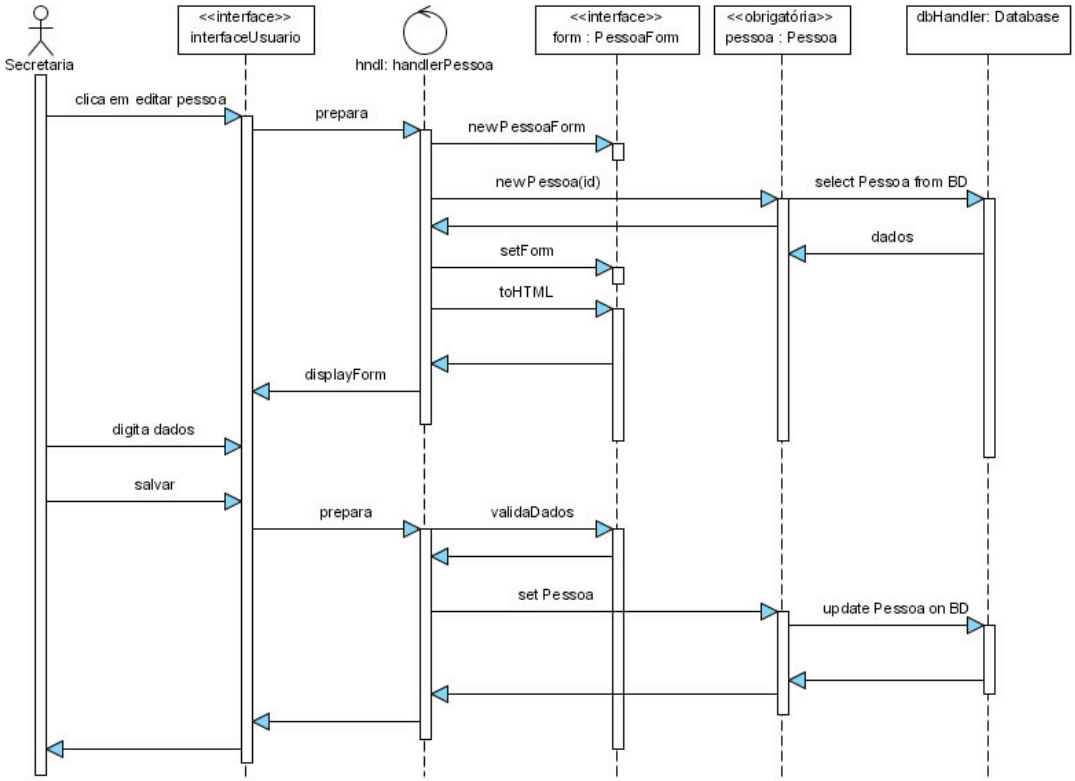

Figura 19. Diagrama de seqüência do caso de uso Editar Pessoa

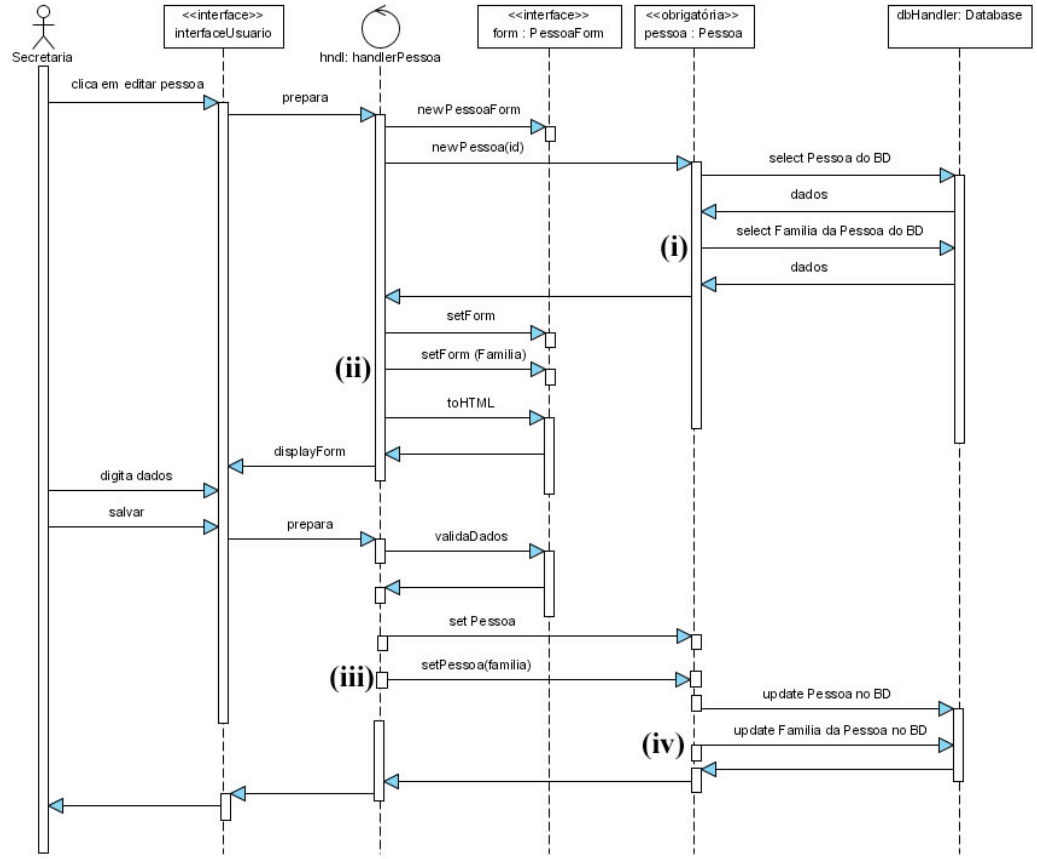

Figura 20. Diagrama de seqüência do caso de uso Editar Pessoa com a adição da *feature* Família

4.2 Projeto e Desenvolvimento

O objetivo desta fase é, baseado nos modelos gerados nas etapas anteriores, sintetizar a solução em código. A linguagem utilizada para desenvolvimento da solução foi PHP, conforme citado no Capítulo 2.

A estratégia utilizada foi, assim como em situações anteriores, desenvolver os casos de uso obrigatórios inicialmente. Adicionalmente, para dar suporte à decisão de como implementar a variabilidade nos casos de uso, foram desenvolvidos protótipos destes casos de uso, com as variações implementadas manualmente. O objetivo destes protótipos foi a identificação dos possíveis pontos de variação, assim como a maneira como se dá esta variação. A abordagem de gerenciamento de variabilidade utilizada neste trabalho é descrita na seção a seguir.

4.2.1 Gerenciamento e implementação de variabilidade

Conforme visto no Capítulo 2, um dos pontos chave de uma LPS é a variabilidade entre produtos. Visualizamos na Figura 11 uma maneira de identificar e categorizar a variabilidade de uma LPS, por meio de modelos de *features*. Porém, esta variabilidade precisa ser implementada em código fonte. As *features* descritas no modelo de *features* da Figura 11 são exemplos de variabilidades externas, pois a seleção destas resultará em mudanças visíveis ao usuário de um produto gerado.

Na implementação de variabilidade da LPS Ligo, foram utilizadas as técnicas de herança, arquivos de configuração e orientação a aspectos. A escolha foi baseada em estudos sobre técnicas que se mostram mais adequadas, e demonstram maior modularidade, de acordo com o tipo da variabilidade [26][53]. A técnica de herança foi utilizada em variações onde era necessária a alteração completa dos métodos (*whole method*). Arquivos de configuração foram utilizados para substituição de valores constantes, e mudanças na estrutura organizacional de uma igreja. Programação orientada a aspectos foi utilizada para as variações acontecidas antes e depois de métodos, de modo a separar os códigos das *features*. A aplicação das técnicas é detalhada a seguir.

Herança

O uso de herança se deu principalmente na implementação das classes relacionadas ao gerenciamento de banco de dados. Foram desenvolvidas classes abstratas de conexão ao banco de dados, bem como classes abstratas de geração de consultas SQL *Update*, *Insert*, *Delete* e *Select*.

A especialização para um banco de dados específico se deu herdando destas classes abstratas. Na classe filha, são especificados os métodos de acesso ao banco de dados, bem como a geração de consultas na sintaxe do banco de dados escolhido. Desta forma, é necessário apenas especializar a classe abstrata para possibilitar o uso de um banco de dados específico. Este é um caso de variabilidade interna, pois a escolha do banco de dados é feita pela organização.

Por exemplo, conforme visualizamos na Figura 21, a classe abstrata *Database* declara atributos e métodos comuns a qualquer implementação de banco de dados. Para tornar possível a utilização do banco de dados MySQL [29], por exemplo, a classe que implementará as funções relacionadas ao banco é declarada como filha de *Database*. Os métodos específicos de acesso ao banco de dados são definidos na classe. A utilização desta técnica em conjunto com o padrão de projeto *Factory*, que lida com o problema de criação de objetos sem a especificação da classe exata, provê a modularidade necessária para alterarmos o banco de dados utilizado sem grande prejuízo ao restante do código dos ativos da LPS. Caso seja necessário alterar o banco de dados utilizado, é necessário apenas modificar o *factory* e criar uma nova classe com os métodos de acesso específicos ao banco de dados desejado.

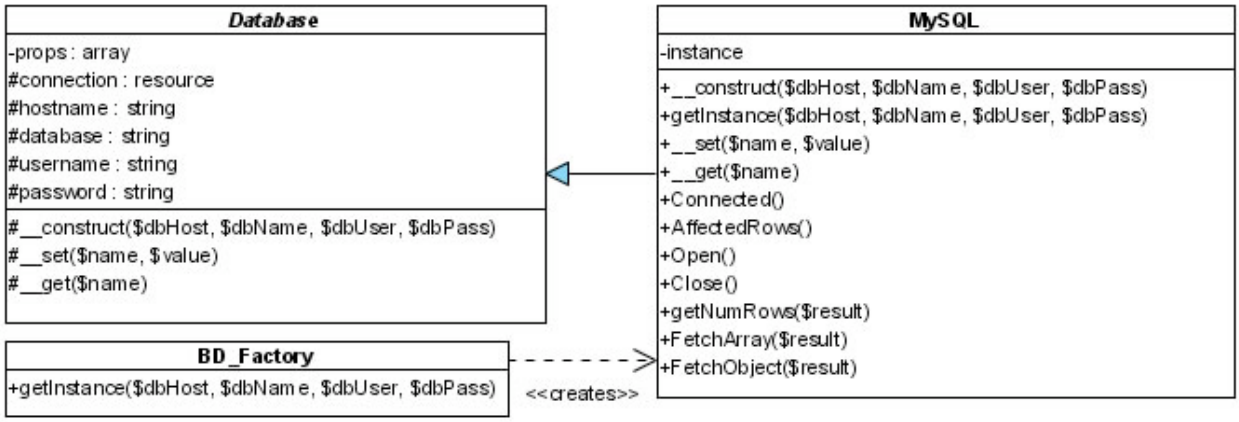

Figura 21. Implementação de variabilidade de banco de dados utilizando herança.

Arquivos de Configuração

A técnica de arquivos de configuração foi utilizada para gerenciar a variabilidade trazida pela *feature* Denominação. Conforme foi visto no Capítulo 2, a denominação de uma igreja reflete-se em mudanças não apenas na forma como a igreja se organiza, assim como na nomenclatura de termos específicos do domínio.

Para cada denominação foram criados arquivos de configuração contendo os mapeamentos de termos específicos relacionados a ela, assim como arquivos XML descrevendo a hierarquia de organização. Durante o processo de geração de produto, que será descrito na próxima seção, de acordo com a escolha da denominação, são carregados os arquivo de configuração referentes à denominação. No caso da escolha de personalização de uma denominação, é possível editar estes termos, assim como configurar a hierarquia.

Orientação a Aspectos

As variabilidades das demais *features* foram implementadas utilizando programação orientada a aspectos (AOP). AOP mostra-se adequada à implementação das variações que ocorrem antes e depois da chamadas de métodos, inserindo comportamento com os *advice before* e *after* [53]. A análise dos artefatos gerados nas etapas anteriores, permitiu a identificação dos possíveis pontos de variação nos casos de uso, de acordo com cada *feature*. Esta análise também identifica *features* como interesses transversais, visto que a adição de *features* impacta diversos casos de uso do sistema.

Inicialmente, foram desenvolvidos protótipos dos casos de uso Adicionar Pessoa e Editar Pessoa, para visualização de como estas variações ocorrem no código. Posteriormente, foi estudado como estas variações poderiam ser separadas em aspectos. Com base neste estudo preliminar, foram desenvolvidos aspectos para cada *feature*. O desenvolvimento dos casos de uso também levou em conta os possíveis *join points* em que o aspecto deve interceptar o programa, inserindo comportamento.

Um exemplo do aspecto relacionado à *feature* Famílias pode ser visto na Figura 22. Este aspecto insere atributos e métodos de gerenciamento da Família à classe Pessoa. Intercepta as inserções, atualizações e consultas da classe Pessoa ao banco de dados, inserindo, atualizando e obtendo os valores relacionados à família. Também adiciona campos na geração do formulário HTML de adição e edição de Pessoas. Esta abordagem permitiu a manter o código das *features* separado, evitando problemas de código entrelaçado e disperso, além de permitir a fácil combinação de aspectos, ao selecionarmos diversas *features* opcionais.

```
<?php
aspect Familia{
    private Pessoa::$pes_familia;
    private Pessoa::$pes_familiaPapel;

    public function Pessoa::get_familia() {
        return $this->pes_familia;
    }

    public function Pessoa::get_familiaPapel() {
        return $this->pes_familiaPapel;
    }

    public function Pessoa::set_familia($val) {
        $this->pes_familia =  $val;
    }

    public function Pessoa::set_familiaPapel($val) {
        $this->pes_familiaPapel =  $val;
    }

    pointcut updateANDinsert:exec(public Pessoa::update(*)) || exec(public Pessoa::insert(*));
    pointcut formInterception:exec(public PessoaForm::getAllData(*)) || exec(public PessoaForm::toHTML(*));

    before(): formInterception {
        ...     intercepta geração de formulário e a captura dos dados digitados
    }

    after(): updateANDinsert {
        ...     após a inserção ou atualização no banco de dados, atualiza os campos de família na tabela
    }

    after(): exec(public Pessoa::select(*)) {
        ...     após a consulta de dados de uma pessoa no banco de dados, obtém os campos de família
    }
}
?>
```

Inter-type declarations

Figura 22. Implementação do aspecto relacionado à *feature* Família.

4.3 Engenharia da Aplicação

A atividade de Engenharia da Aplicação consiste em desenvolver os produtos membros da linha. O desenvolvimento de um produto não é feito do zero, mas, é baseado nos ativos gerados durante o desenvolvimento da linha. A arquitetura da LPS é utilizada como base e é adaptada e personalizada de acordo com as necessidades específicas do produto. Por estas razões, o desenvolvimento de um produto em uma LPS é também chamado de geração ou derivação de produtos.

Durante esta atividade, novos requisitos para um produto podem surgir. Neste caso, é necessário fazer uma nova iteração de desenvolvimento da linha, para que estes novos requisitos sejam inclusos nos modelos da linha. Esta nova iteração inicia com a inclusão das novas *features*, adição de novos casos de uso ou adaptação dos já existentes. Os modelos estáticos e dinâmicos são adaptados, o que se reflete na alteração da arquitetura da LPS, se necessário, e em seguida os novos requisitos são implementados. Desta forma, os novos requisitos são incorporados à LPS, possibilitando inclusive a utilização destes por outros produtos da linha. É importante ressaltar que a abordagem utilizada para evolução da linha varia de acordo com a estratégia utilizada pela organização que a utiliza.

Para a LPS Ligo, foi desenvolvido um gerador de produtos, que guia o processo de derivação de membros da LPS. As atividades do processo de geração são descritas a seguir, e podem ser visualizadas na Figura 23.

1. Escolher denominação: Esta atividade é definida como a atividade inicial de geração de produtos, pois, desde o processo de geração, já utilizaremos termos específicos de acordo com a seleção feita;
2. Personalização: Caso seja escolhida
3. Escolher Tipo de Igreja: De acordo com a *feature* Tipo de Igreja, é feita a escolha do tipo da igreja utilizadora;
4. Configuração da igreja: Nesta etapa, são adicionadas as informações iniciais relacionadas à igreja, como nome, endereço e, no caso da seleção do tipo como *multi-site*, esta etapa é repetida para cada uma das igrejas que será gerenciada;
5. Seleção de *features*: As demais *features* são selecionadas e a seleção passa por um processo de validação pois, conforme vimos no Capítulo 2, existem *features* que podem ser mutuamente inclusivas ou exclusivas;
6. Escolher diretório de destino: O diretório alvo, onde serão colocados os arquivos gerados referentes ao produto é escolhido;
7. Gerar produto: A atividade final consiste em, baseado nas escolhas anteriores, realizar o processo de *weaving* dos aspectos com o código fonte.

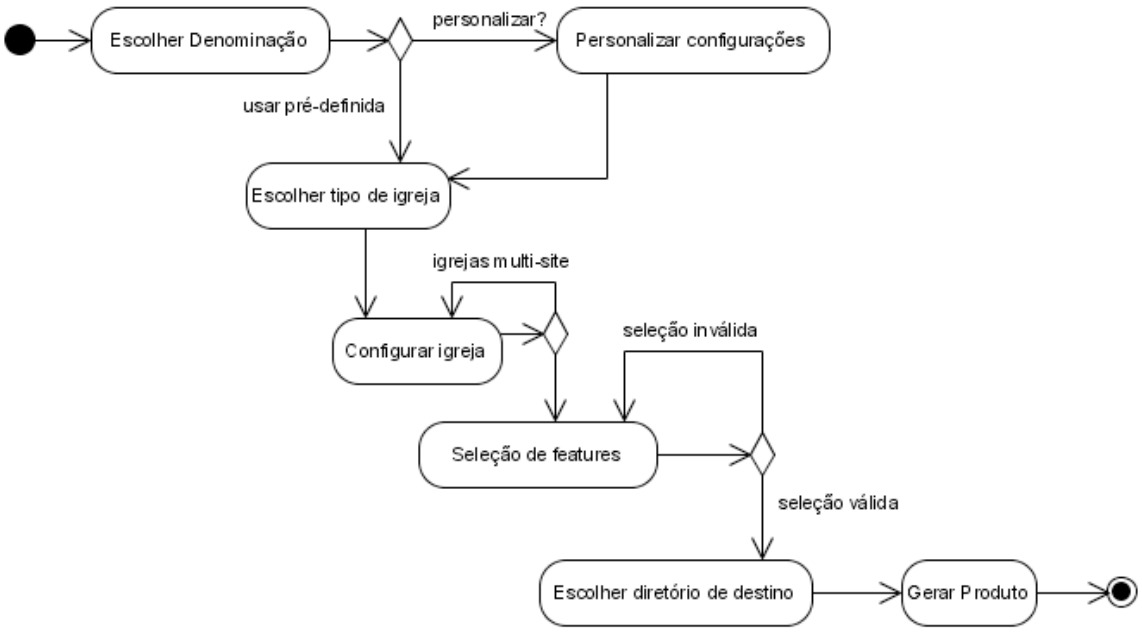

Figura 23. Diagrama de atividades que ilustra o processo de geração de um produto.

As *features* são descritas por meio de arquivos XML, que contém uma estrutura descritiva dos componentes de código participantes em uma *feature*, como classes, aspectos, folhas de estilo CSS (*Cascading Style Sheets*) [54], bibliotecas *javascript* [55], *templates* de páginas. Estas informações são utilizadas pelo gerador durante o processo de geração de produto. Um exemplo de descrição de *features* é visto na Figura 24. Neste exemplo, é possível observar a estrutura de descrição na própria figura, por meio das *tags*. O exemplo da figura ilustra a descrição da *feature* opcional Família. São descritas as classes e aspectos participantes da *feature*, neste caso, apenas *Familia.php* e *familia.aspect.php*, respectivamente. Arquivos auxiliares também são descritos para inclusão durante o processo de geração do produto.

```
<?xml version="1.0" encoding="UTF-8"?>
<feature>
<name>familia</name>
<type>opcional</type>
<classes>
    <class>Familia.php</class>
</classes>
<aspects>
    <aspect>familia.aspect.php</aspect>
</aspects>
<files>
    <file>familias.php</file>
    <file>adiciona_familia.php</file>
    <file>edita_familia.php</file>
    <file>pessoa_familia.php</file>
</files>
</feature>
```

Figura 24. Exemplo de descrição XML da *feature* Família

O processo de geração de produtos é ilustrado com telas no Apêndice B.

4.3.1 Avaliação

Durante a fase de desenvolvimento da LPS, diversas igrejas foram consultadas para avaliação das *features* providas pela linha. Essa interação foi importante, pois já durante a fase de desenvolvimento foram identificadas novas *features*, como Rede Ministerial, assim como outras *features* foram rearranjadas, como se vê no agrupamento das *features* Pequenos Grupos e Ministérios sob a feature Grupos e na divisão de tipos de igrejas.

Como forma de validação, um produto gerado está sendo atualmente implantado na igreja presbiteriana Comunidade Memorial [56], situada na Lagoa do Araçá, Recife – PE. O produto gerado consiste da seleção de *features* opcionais Família, Finanças, Cursos e Grupos, que refletem a necessidade atual da igreja, não foram necessárias alterações ou desenvolvimento de novas *features*.

Um outro produto foi gerado de acordo com as necessidades da Igreja Episcopal Carismática Paróquia da Reconciliação [57], situada em Boa Viagem, Recife – PE. Este produto consiste da seleção das features Família, Finanças, Grupos, Website e Eventos. Foi observado que a *feature* Eventos precisa ser especializada para lidar com eventos específicos. Um exemplo é a realização de Cursilhos, pequenos retiros durante o final de semana, onde homens ou mulheres passam este tempo convivendo e estudando a Bíblia. Este tipo de evento, característico da igreja Episcopal, leva em conta não apenas o agendamento do evento, mas também o registro das pessoas participantes, registro das pessoas que irão trabalhar, controle das finanças, entre outras necessidades específicas.

Capítulo 5

Conclusões e Trabalhos Futuros

O desenvolvimento de software é uma atividade que muitas vezes torna-se repetitiva, pois poucos sistemas são de fato únicos. Reuso de software é uma preocupação constante dos desenvolvedores e vem mostrando-se como algo essencial a organizações desenvolvedoras de software. Diversas organizações desenvolvem produtos similares em domínios específicos. Faz-se necessário adotar metodologias para tirar proveito das características comuns a estes produtos, e procurar otimizar o reuso de software nestas situações. Linhas de produtos de software é uma abordagem que se propõe a tirar vantagem desta situação e melhorar a produtividade do desenvolvimento de software.

Este trabalho propôs o desenvolvimento de uma linha de produtos de software para gerenciamento de igrejas cristãs. O domínio de aplicação mostrou-se adequado ao desenvolvimento de uma LPS. Igrejas cristãs compartilham muitas semelhanças, porém vimos que aspectos como denominação, tamanho e localização afetam a organização de uma igreja.

Este capítulo apresenta as principais contribuições dadas por este trabalho, dificuldades encontradas, bem como possíveis trabalhos futuros.

5.1 Contribuições

A principal contribuição do trabalho é a LPS Ligo, que agrupa aspectos essenciais do gerenciamento de igrejas cristãs. Algumas outras contribuições estão listadas a seguir:

- Instanciação de um processo de desenvolvimento de LPS;
- Estudo e implementação de técnicas de variabilidade de acordo com o tipo de variabilidade. As técnicas utilizadas foram: Herança, Arquivos de Configuração e Programação Orientada a Aspectos;
- Modificação do weaver do *phpAspect*, visando a correção de *inter-type declarations* para inserção de atributos e constantes e a possibilidade de poder trabalhar com acentos;
- Desenvolvimento de um gerador de produtos de LPS parametrizável, que pode ser utilizado em outras LPS;
- Geração de um produto específico para a igreja Comunidade Memorial [56], situada na Lagoa do Araçá, Recife – PE. O produto se mostrou apropriado às necessidades desta igreja, e está atualmente sendo implantado.

- Geração de um produto para a igreja Paróquia da Reconciliação [57], situada em Boa Viagem, Recife – PE. Este produto ainda precisa de algumas modificações para se adequar às necessidades desta igreja.

5.2 Dificuldades encontradas

Por ser uma técnica com que não tinha experiência prévia, houve dificuldade em passar da etapa de requisitos e projeto para o código fonte. Existem poucas ferramentas disponíveis à comunidade acadêmica, que dão suporte ao desenvolvimento de LPS como um todo. Por exemplo, a ferramenta pure::variants auxilia no processo de desenvolvimento do modelo de *features*, porém não é útil para modelagem de casos de uso, modelagem estática, entre outros. Algumas ferramentas comerciais não permitem o seu uso em ambiente acadêmico. Na literatura da área, existe bastante material relacionado aos aspectos de requisitos, análise e projeto, porém, pouca informação relacionada a implementação de LPS.

O *phpAspect*, como toda ferramenta em desenvolvimento, ainda possui alguns problemas, e não está completa. A comunicação com o principal desenvolvedor do projeto foi escassa, só foi possível obter um retorno do mesmo ao fim do projeto. A base de usuários também ainda não é vasta. Desta forma, alguns destes problemas foram solucionados durante o decorrer deste trabalho.

5.3 Trabalhos futuros

Como trabalhos futuros, podemos citar a evolução da LPS Ligo, incluindo novas *features*, refinando e especializando algumas já existentes, procurando atender às necessidades dos usuários finais, como visto no caso da igreja Episcopal, com a *feature* Eventos. Outra possível adição ao trabalho é a integração da LPS a uma distribuição Linux, gerando uma distribuição direcionada à igrejas, visando minimizar o problema de uso de software ilegal.

Para melhorar o processo de desenvolvimento da LPS, sugere-se que, ao invés de descrições XML escritas manualmente pelos desenvolvedores da LPS, sejam incluídas anotações, na forma de comentários, detalhando a relação do componente de código com uma *feature*. O gerador então ficaria responsável por ler estas anotações e gerar as descrições a partir delas.

Conforme visto na seção anterior, há uma falta de ferramentas que dêem suporte a LPS como um todo, auxiliando em todas as fases do desenvolvimento. O desenvolvimento de uma ferramenta que, baseada em uma metodologia de desenvolvimento de LPS, dê suporte às fases desta metodologia, é um potencial trabalho futuro. A própria ferramenta pode ser desenvolvida como uma LPS, em que cada produto seria a instância de uma determinada metodologia de desenvolvimento de LPS.

Assim como visto na seção anterior, apenas próximo à conclusão do trabalho, foi possível estabelecer contato com o principal desenvolvedor do *phpAspect*. Este contato no entanto, se mostrou importante para possíveis trabalhos futuros, neste caso, específicos para o *phpAspect*. A inclusão de *join points* específicos relacionados à linguagem PHP e ambiente web é um dos principais. Também podemos citar o auxílio no desenvolvimento do plugin APDT (*Aspect PHP Development Tools*) [58], que visa o suporte ao desenvolvimento orientado a aspectos com a linguagem *phpAspect* no ambiente Eclipse [24].

Bibliografia

[1] CLEMENTS, Paul, NORTHROP, Linda. *Software Product Lines: Practices and Patterns*. 3. ed. Boston: Addison-Wesley, 2002. 608 p.

[2] LAUDON, Kenneth, LAUDON, Jane. *Management Information Systems: Managing the Digital Firm*. 10. ed. New Jersey: Prentice Hall, 2006. 736 p.

[3] WYCHE, Susan, et al. *Technology in Spiritual Formation: An Exploratory Study of Computer Mediated Religious Communications*. In: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work CSCW '06, 2006, Alberta. p.199-208.

[4] AKIN, Daniel, GARRETT Jr., James, REYMOND, Robert, WHITE, James, ZAHL, Paul. *Perspectives on Church Government: Five Views of Church Polity*. 1. ed. Nashville: B&H Publishing Group, 2004. 353 p.

[5] MILI, Hafedh, MILI, Ali, YACOUB, Sherif, ADDY, Edward. *Reuse-Based Software Engineering: Techniques, Organizations, and Controls*. 1. ed. New York: Wiley-Interscience, 2001. 650 p.

[6] GREENFIELD, Jack, SHORT, Keith, COOK, Steve, KENT, Stuart. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. 1 ed. New York: Wiley, 2004. 500 p.

[7] DAVIS, Stanley. *Future Perfect*. 1. ed. New York: Perseus Book Group, 1997. 255 p.

[8] WEISS, David. LAI, Robert. *Software Product-Line Engineering. A Family-Based Software Development Process*. 1. Ed. Boston: Addison-Wesley Professional, 1999. 448 p.

[9] GRISS, Martin. *Product-line architectures*. In G. T. Heineman and W. T. Councill, editors, Component-Based Software Engineering, cap.22 p. 405-420. Addison Wesley, 2001

[10] POHL, Klaus, BÖCKLE, Günter, VAN DER LINDEN, Frank. *Software Product Line Engineering: Foundations, Principles, and Techniques*. 1. ed. New York: Springer, 2005. 468 p.

[11] KRUEGER, Charles. "*Introduction to the Emerging Practice of Software Product Line Development*", In: Methods and Tools, vol 14, nr. 3, pp 3-15, Fall 2006.

[12] RINE, D.C., SONNEMANN, R.M. "*Investments in reusable software. A study of software reuse investment success factors*", In: The journal of systems and software, nr. 41, pp 17-32, Elsevier, 1998.

[13] RINE, D.C., NADA, N. *An empirical study of a software reuse reference model*. in Information and Software Technology, nr 42, pp. 47-65, Elsevier, 2000.

[14] BASS, Len, CLEMENTS, Paul, KAZMAN, Rick. *Software Architecture in Practice*. 2. Ed. Boston: Addison-Wesley Professional, 2003. 560 p.

[15] DAGER, J.C. *Cummin's Experience in Developing a Software Product Line Architecture for Real-Time Embedded Diesel Engine Controls*. In: Proc. 1st Software Product Line Conf. (SPLC1), Kluwer, Dordrecht, Netherlands, 2000, pp. 23–46.

[16] BUHRDORF, Ross, CHURCHETT, Dale, KRUEGER, Charles. *Salion's Experience with a Reactive Software Product Line Approach*. In: Proceeding of the 5th International

Workshop on Product Family Engineering. Nov 2003. Siena, Italy. Springer-Verlag LNCS 3014, p 315.

[17] HETRICK, William, KRUEGER, Charles, MOORE, Joseph. *Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice*. In: OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA. ACM 1-59593-491-X/06/0010.

[18] VERLAGE, Martin, KIESGEN, Thomas. *Five Years of Product Line Engineering in a Small Company*. In: ICSE'05, May 15-21, 2005, St. Louis, Missouri, USA. Copyright 2005 ACM 1-58 113-963-2/05/0005

[19] GOMAA, Hassan. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. 1. ed. Boston: Addison-Wesley Professional, 2004. 736 p.

[20] JACOBSON, Ivar, BOOCH, Grady, RUMBAUGH, James. *The Unified Software Development Process*. 1. ed. Reading, MA: Addison-Wesley Professional, 1999. 463 p.

[21] BOEHM, Barry. *A Spiral Model of Software Development and Enhancement*. IEEE Computer v.21, Issue 5: p.61–72, 1998.

[22] CZARNECKI, Krzysztof, EISENECKER, Ulrich. *Generative Programming: Methods, Tools, and Applications*. 1. ed. Boston: Addison-Wesley Professional, 2000. 864 p.

[23] pure-systems: pure::variants. Disponível em: <http://www.pure-systems.com/Community_Edition.55.0.html> Acesso em: 17 de Novembro de 2007.

[24] Eclipse Foundation: Eclipse. Disponível em: <http://www.eclipse.org> Acesso em: 17 de Novembro de 2007.

[25] RUMBAUGH, James, BOOCH, Grady, JACOBSON, Ivar. *The Unified Modeling Language Reference Manual*, 2 ed. Boston: Addison-Wesley, 2005. 576p.

[26] ANASTASOPOULOS, Michalis, GACEK, Cristina. *Implementing Product Line Variabilities*. In: SSR'01, May 18-20, 2001, Toronto, Ontario, Canada. Copyright 2001 ACM 1-58113-358-8/01/0005.

[27] PHP. Disponível em: <http://www.php.net/> Acesso em: 17 de Novembro de 2007.

[28] Apache Foundation: *HTTP Server*. Disponível em: <http://httpd.apache.org/> Acesso em: 17 de Novembro de 2007.

[29] MySQL AB: *MySQL*. Disponível em: <http://www.mysql.org/> Acesso em: 17 de Novembro de 2007.

[30] W3C: *Extensible Markup Language*. Disponível em: <http://www.w3.org/XML/> Acesso em: 19 de Novembro de 2007.

[31] W3C: *XSL Transformations*. Disponível em: <http://www.w3.org/TR/xslt> Acesso em: 17 de Novembro de 2007.

[32] Netcraft: *Web Server Survey*. Disponível em: <http://www.netcraft.com/Survey/> Acesso em: 17 de Novembro de 2007.

[33] ELRAD, T., AKSIT, M, KICZALES, G, LIEBERHERR, K, OSSHER, H. *Discussing Aspects of AOP*. In: Communications of the ACM 44(10),pp.33-38, October 2001b.

[34] KICZALES, Gregor, LAMPING, John, MENDHEKAR, Anurag, MAEDA, Chris, LOPES, Cristina, Jean-Marc, IRWIN, John. *Aspect-Oriented Programming*. In: European Conference on Object-Oriented Programming, ECOOP'97, LNCS 1241, p. 220–242, Finland, June 1997. Springer-Verlag.

[35] AspectJ. Disponível em: <http://www.eclipse.org/aspectj/> Acesso em: 17 de Novembro de 2007.

[36] FILMAN, Robert, FRIEDMAN, Daniel. *Aspect-Oriented Programming is Quantification and Obliviousness*. In: Workshop on Advanced Separation of Concerns, OOPSLA'2000, p. 220–242, Minnesota, October 2000.

[37] phpAspect. Disponível em: <http://phpaspect.org/> Acesso em: 17 de Novembro de 2007.
[38] The Lex & Yacc Page. Disponível em: <http://dinosaur.compilertools.net/> Acesso em: 17 de Novembro de 2007.
[39] PECL: *Parse_Tree*. Disponível em: <http://pecl.php.net/package/Parse_Tree> Acesso em: 17 de Novembro de 2007.
[40] RSS Advisory Board: *RSS 2.0 Specification*. Disponível em: <http://www.rssboard.org/rss-specification> Acesso em: 19 de Novembro de 2007.
[41] AtomEnabled: *Atom*. Disponível em: <http://www.atomenabled.org/> Acesso em: 19 de Novembro de 2007.
[42] W3C: *MathML*. Disponível em: <http://www.w3.org/Math/> Acesso em: 19 de Novembro de 2007.
[43] Recordare: MusicXML. Disponível em: <http://www.musicxml.org/xml.html> Acesso em: 19 de Novembro de 2007.
[44] W3C: XML Schema. Disponível em: <http://www.w3.org/XML/Schema> Acesso em: 19 de Novembro de 2007.
[45] G-Cell 2.0. Disponível em: <http://www.celulas.com.br/ferra1.htm> Acesso em: 17 de Novembro de 2007.
[46] Ativo Sistemas: abcFinance. Disponível em: <http://www.ativosistemas.com.br/abcFinance.htm> Acesso em: 17 de Novembro de 2007.
[47] Ativo Sistemas: Reflex. Disponível em: <http://www.ativosistemas.com.br/Reflex.htm> Acesso em: 17 de Novembro de 2007.
[48] Wisys: LOUVADEUS. Disponível em: <http://www.wisys.com.br/Louvadeus/> Acesso em: 17 de Novembro de 2007.
[49] DM10: Church Tradicional. Disponível em: <http://www.dm10.com.br/> Acesso em: 17 de Novembro de 2007.
[50] SN Systems: SIGI. Disponível em: <http://www.soareseneves.com.br/> Acesso em: 17 de Novembro de 2007.
[51] BUGBEE, Bruce, COUSINS, Don, SEIDMAN, Wendy. *Network Participant's Guide: The Right People, in the Right Places, for the Right Reasons, at the Right Time*. 2 ed. Grand Rapids: Zondervan, 2005. 192 p.
[52] The Economist: *Jesus, CEO: Churches as Businesses*. The Economist (Dec 20, 2005). 2005. http://www.economist.com/world/na/displaystory.cfm?story_id=5323597&no_jw_tran=5323591&no_na_tran=5323591.
[53] RIBEIRO, Márcio, MATOS JR., Pedro, BORBA, Paulo, CARDIM, Ivan. *On the Modularity of Aspect-Oriented and Other Techniques for Implementing Product Lines Variabilities*. In: I Latin American Workshop on Aspect-Oriented Software Development - LA-WASP'2007, affiliated with SBES'07, João Pessoa-PB, Brazil, October 2007.
[54] W3C: Cascading Style Sheets. Disponível em: <http://www.w3.org/Style/CSS/> Acesso em: 17 de Novembro de 2007.
[55] Mozilla Developer Center: Javascript. Disponível em: <http://developer.mozilla.org/en/docs/JavaScript/> Acesso em: 17 de Novembro de 2007.
[56] Comunidade Memorial. Disponível em: <http://www.comunidadememorial.com/> Acesso em: 17 de Novembro de 2007.
[57] Paróquia da Reconciliação. Disponível em: <http://www.reconciliacao.org/> Acesso em: 17 de Novembro de 2007.
[58] Google Code: APDT: Aspect PHP Development Tools. Disponível em: <http://code.google.com/p/apdt/> Acesso em: 17 de Novembro de 2007.

Apêndice A

Caso de uso Adicionar Pessoa (UC_1)

Categoria Reuso
Obrigatório
Sumário
Este caso de uso permite a adição de pessoas ao banco de dados
Atores
Secretária
Pré-Condições
Usuário precisa estar logado no sistema.
Descrição
1. Usuário clica em Adicionar Pessoa 2. O formulário de cadastro de pessoa é apresentado ao usuário 3. O usuário digita os dados da pessoa a ser adicionada e clica em salvar 4. O sistema faz a validação dos dados 5. O sistema adiciona a nova pessoa ao banco de dados 6. A relação de pessoas atualizada é apresentada ao usuário, encerra caso de uso
Alternativas
A01 – Cancelar cadastro 1. O usuário pode, a qualquer momento, clicar em cancelar e encerrar o caso de uso.
Pontos de variação
V01 – Famílias • **Tipo:** opcional • **Linhas:** 2,4,5 • **Descrição:** Campos relacionados à *feature* família são adicionados, na apresentação do formulário. As informações destes campos são também validadas e adicionadas ao banco de dados.
Pós-Condição
Cadastro de pessoas atualizado

Apêndice B

Telas da LPS Ligo

Apresentaremos aqui as telas referentes ao processo de geração do produto para implantação na igreja Comunidade Memorial, conforme descrito no texto da monografia. Ao final, é apresentada uma tela do sistema em execução.


LIGO
Passo 3 :: Configurar Igreja
* Nome: Comunidade Memorial
Endereço: Av. Pinheiros, 1110
Cidade: Recife
Estado: Pernambuco
CEP: 55000-000
* E-mail: info@comunidadememorial.com
Telefone: 3428-3666
xxxx-xxxx
Logomarca: /Users/ibook/Desktop/ Browse...
Próximo


LIGO
Passo 4 :: Selecionar Features
Features comuns: Mailing,Pessoas,Relatórios,Tipo Igreja,Denominação
estas features estão presentes em todos os produtos
Selecione as Features:
Eventos
Famílias
Cursos
Grupos
Finanças
Website
Rede Ministerial
Próximo

Tela do produto em execução, caso de uso Editar Pessoa: