Uma abordagem orientada a aspectos para desenvolvimento de linhas de produtos de software

Stanley Fabrizio Pacios

Uma abordagem orientada a aspectos para desenvolvimento de linhas de produtos de software

Stanley Fabrizio Pacios

Orientadora: ***Profa. Dra. Rosana Teresinha Vaccare Braga***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências - Ciências de Computação e Matemática Computacional.

USP – São Carlos
Dezembro/2006

“Conhece a ti mesmo”
(Sócrates)

Aos meus pais Carmen e Waldir

Agradecimentos

A Deus por me ajudar muitas vezes.

A minha orientadora, Profa. Dra. Rosana Teresinha Vaccare Braga, duas vezes: pela dedicação e pelo horário extra concedido aos sábados.

Ao Prof. Dr. Paulo César Masiero pelo auxílio na orientação.

Ao Prof. Dr. Fernão Stella R. Germano pela revisão da dissertação.

Aos meus amigos Johnny, Mario, Bruno e Luiz, os quais foram os causadores de tudo isto.

Aos meus amigos Fabiano, Jão, Delane, Diogo, Kazu, Marcella, Alessandra, Paulo e Rafael.

Aos amigos de laboratório Érica, Bira, Camila, André, Débora, Otávio, Paula, Percy, Sandro, Marcelo, Antoniely, Marco, Carlos, André, Darley e Harú (praticamente LABES).

A galera da PGCOMPUSP-04 e demais agregados da pós anteriores e posteriores por serem tão gente boa e divertidos e pelos churrascos, jogos de futebol, jam sessions, baladas, viagens, reuniões, aniversários e outras coisas.

A todos os professores do ICMC que me deram aulas ou que me tiraram dúvidas e me ajudaram.

Aos funcionários do ICMC pelo bom trabalho prestado.

A CAPES pelo apoio financeiro.

Índice

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

APOA Análise e Projeto Orientado a Aspectos

BMPL Linguagem de Modelagem de Processos de Negócios (do inglês *Business Process Modeling Language*)

DED Documento de Estudo de Domínio

DSOA Desenvolvimento de Software Orientado a Aspectos

FODA Análise de Domínio Orientada a Features (do inglês Feature Oriented Domain Analysis)

ESPLEP do inglês Evolutionary Software Product Line Engineering Process

LP Linhas de Produtos (do inglês: *Product Line*)

OA Orientado a Aspectos ou Orientação a Aspectos

OO Orientado a Objetos ou Orientação a Objetos

PLP Prática de Linha de Produtos (do inglês: *Product Line Practice*)

PLUS do inglês *Product Line UML Base Software Engineering*

POA Programação Orientada a Aspectos (do inglês: Aspect-Oriented Programming)

POO Programação Orientada a Objetos (do inglês: *Object-Oriented Programming*)

PU Processo Unificado (do inglês: *Unified Process*)

SEI Instituto de Engenharia de Software da CMU (do inglês: *Software Engineering Institute*)

UML do inglês *Unified Modeling Language*

Resumo

Este trabalho investiga como o desenvolvimento de linhas de produtos de software pode ser beneficiado pela utilização da programação orientada a aspectos para reduzir o acoplamento e aumentar a coesão das *features* da linha de produtos. Como resultado dessa investigação, uma abordagem para desenvolvimento incremental de linhas de produtos baseado em aspectos é proposta. São apresentadas as etapas, atividades e artefatos dessa abordagem. Por ser uma abordagem incremental, reduz-se a carga de trabalho necessária no início da produção da linha de produtos. Isso é conseguido graças à utilização de aspectos. Com isso, tem-se as vantagens de linhas de produtos ao mesmo tempo amenizando a desvantagem do risco do alto investimento inicial não ter o retorno esperado. A abordagem foi proposta com base em práticas estabelecidas de desenvolvimento de linhas de produtos de software e no estudo das práticas atuais para análise e projeto orientado a aspectos. Foi dada ênfase à abordagem Tema, que é utilizada neste trabalho como parte do ferramental para análise e projeto. A abordagem desenvolvida especifica práticas desde a análise de domínio até a implementação. Os aspectos são tratados desde os estágios iniciais do desenvolvimento. Técnicas para implementação com orientação a aspectos são propostas. Um estudo de caso utilizando as linguagens Java e AspectJ é apresentado para ilustrar as idéias propostas.

Abstract

This work investigates how the development of software product lines can benefit from the use of the aspect-oriented programming to reduce coupling and increase cohesion of the product line features. As a result of this investigation, an approach to the incremental development of software product lines based in aspects is proposed. The phasesstages, activities, and artifacts of this approach are presented. As it is an incremental approach, the amount of work needed in the beginning of the product line development is reduced. This is accomplished thanks to the use of aspects. This way, the advantages of product lines is obtained, attenuating, at the same time, of the risk of the high initial investment not having the expected return. The approach has been proposed based on the established practices of software product lines development and on the study of the actual practices for aspectoriented analysis and design. Emphasys has been given to the Theme Approach, which is used in this work as part of the analysis and design tools. The proposed approach specifies practices from the domain analysis up to the implementation. The aspects are treated since the early development stages. Aspect-oriented implementation techniques are proposed. A case study using Java and AspectJ languages is presented to illustrate the proposed ideas.

1. Introdução

Processos de desenvolvimento de software são necessários, pois tornam o desenvolvimento previsível e reprodutível, ajudando a reduzir a complexidade do desenvolvimento (Alves, 2005). Isso significa qualidade no processo de software, o que é fundamental para o desenvolvimento em larga escala e traz qualidade também para o produto de software.

A programação orientada a objetos (POO), por ser um paradigma de programação consolidado, possui processos de desenvolvimento bem definidos, como o Processo Unificado (PU) (Jacobson et al., 1999). Esses processos foram criados por uma sucessão de refinamentos dos processos ad-hoc que surgiram da experiência dos desenvolvedores.

A Programação Orientada a Aspectos (POA) (Kiczales et al., 1997) é um paradigma de programação relativamente novo e interessante para a comunidade de software, que surgiu para complementar o paradigma da POO, que dificulta a separação de interesses em certas situações, como para alguns requisitos não funcionais.

A POA não se consolidou como a solução definitiva para os problemas da complexidade do desenvolvimento de software. Porém, já há indicativos de ser boa solução para alguns problemas, como por exemplo, implementar requisitos não funcionais.

Assim, a comunidade de software em pesquisas recentes (Pearce e Noble, 2006; Apel et al., 2006; Griswold et al., 2006), busca desenvolver processos para análise e projeto de software orientado a aspectos. Atualmente existem muitas pesquisas para solucionar problemas específicos das várias fases do desenvolvimento.

Uma abordagem completa para APOA é a abordagem Tema (Clarke e Baniassad, 2005), porém até o momento da elaboração deste trabalho a abordagem Tema ainda é recente e pouco utilizada, portanto não se tem uma definição da sua eficiência.

Quanto à definição dos conceitos e linguagens da orientação a aspectos (OA), as pesquisas já se encontram em estágios maduros (Baniassad et al., 2006). Porém, as pesquisas com relação a processos ainda precisam ser mais aperfeiçoadas. Acompanhando as pesquisas atuais, nota-se que elas estão mais avançadas para tratar dos estágios mais próximos do final do processo de software. Uma das grandes dificuldades atuais é pensar em termos aspectuais nos estágios iniciais do desenvolvimento, como por exemplo, de requisitos (Baniassad et al., 2006).

Técnicas de implementação (como padrões de projetos) adequadas para POA também são necessárias para complementar o bom desenvolvimento orientado a aspectos. A necessidade de técnicas que auxiliem o projeto e o desenvolvimento de software de maior qualidade e em menor tempo é uma das preocupações da engenharia de software. Muitos produtos de software são desenvolvidos em função de artefatos já especificados e implementados, utilizando técnicas de reutilização de software. Nesse contexto, a técnica Linha de Produtos de Software (LPS, ou somente LP no caso do presente trabalho) (Griss, 2000; Bass et al., 1998; Weiss e Lai, 1999) surge como uma proposta de construção e reutilização sistemática de software baseada em um domínio específico (Aragon, 2004). Essa técnica já mostrou seu valor no desenvolvimento OO e pode beneficiar o desenvolvimento OA, também dele se beneficiando.

1.1. Motivação

As dificuldades no desenvolvimento de software orientado a aspectos são inerentes em grande parte à falta de um processo completo e integrado para o desenvolvimento. Assim, uma das motivações deste trabalho é desenvolver uma abordagem integrada para APOA, a qual deve servir, no mínimo como um ponto de partida para a evolução rumo a um processo de desenvolvimento.

A maioria das pesquisas desenvolvidas atualmente nessa área aborda partes isoladas do desenvolvimento, resolvendo problemas específicos, mas não resolvendo o problema principal de desenvolver software orientado a aspectos.

Em tese, várias dessas técnicas e tecnologias podem ser combinadas para resolver melhor o problema. Este trabalho pretende investigar como algumas destas técnicas podem ser combinadas e utilizadas em conjunto.

Linhas de produtos de software, por sua vez, possuem processos de desenvolvimento apropriados, mas existem outras dificuldades. Mesmo com processos, o desenvolvimento não deixa de ser trabalhoso. O custo/esforço de desenvolvimento de linhas de produtos é alto no início do desenvolvimento. É necessária uma análise de domínio e extensivo projeto no início. Isso ocasiona risco para os interessados, que podem ter prejuízos se depois a linha de produtos não trouxer o retorno esperado do investimento.

Assim, os processos para LPs ainda permitem aperfeiçoamentos. Uma possibilidade a ser investigada é a de que a criação de LPs pode ser beneficiada pelo uso de aspectos. Devido a capacidade dos aspectos modularizarem interesses, eles poderiam modularizar as *features* de LPs e acrescentar *features* sem a necessidade de projetar extensivamente e previamente. Isso faria com que a linha de produtos pudesse ser incremental e fosse desenvolvida em ciclos iterativos.

Outra possibilidade a ser investigada é o uso da abordagem Tema para análise e projeto das *features* orientadas a aspectos da linha de produtos. Com isso, também seria investigada a eficiência da abordagem Tema.

Assim, os motivos para realizar este trabalho são vários, dentre os quais podemos citar:

- Carência de processos de software para APOA;
- Carência de técnicas de APOA;
- Crescente interesse da comunidade de software em aspectos nos estágios iniciais do desenvolvimento de software; e
- Falta da uma abordagem completa e integrada para desenvolvimento de LPs orientadas a aspectos.

1.2. Objetivos

A abordagem aqui proposta tem a intenção de melhorar o desenvolvimento de linhas de produtos, utilizando as vantagens inerentes da POA. Assim, é investigado como os aspectos podem melhorar a modularização de partes da linha de produtos, isolando interesses e beneficiando as linhas de produtos, permitindo a criação de *features* mais conectáveis e intercambiáveis.

Como os aspectos têm a propriedade de permitir a alteração do funcionamento do software de maneira não invasiva, espera-se conseguir uma abordagem que faça com que a linha de produtos de software possa crescer incrementalmente com o tempo. Essa seria uma grande vantagem para as linhas de produtos, já que uma das preocupações e desvantagens no seu desenvolvimento é o alto esforço inicial de trabalho investido. Ressalte-se que as linhas de produtos têm o risco de não trazerem o retorno do investimento. É bom lembrar que as técnicas de evolução e manutenção não invasivas do código são desejáveis também para o desenvolvimento de software tradicional.

Resumindo, os objetivos deste trabalho são:

- Apresentar um processo para desenvolvimento de linhas de produtos orientadas a aspectos;
- Reduzir o esforço para criação e manutenção de LPs fazendo com que a linha de produtos possa ser desenvolvida de maneira incremental;
- Conciliar técnicas atuais para APOA em uma abordagem operacional de desenvolvimento.

Além disso, são objetivos deste trabalho:

- Investigar o uso da abordagem Tema (Clarke e Baniassad, 2005) para análise e projeto OA;
- Fazer com que a abordagem apresentada possa servir de marco de partida para evolução de um verdadeiro processo de desenvolvimento de software orientado a aspectos (DSOA);
- Investigar o quanto os aspectos beneficiam a modularização de *features* de LPs e facilitam sua composição; e
- Investigar como os aspectos podem beneficiar a evolução e manutenção de software.

1.3. Organização

O restante deste documento está organizado como segue. No Capítulo 2 revisam-se os conceitos, técnicas e tecnologias envolvidas no trabalho apresentado. No Capítulo 3 apresenta-se a abordagem para criação de linhas de produtos incremental, que é a proposta deste trabalho, explicando detalhadamente as fases do processo e as atividades e artefatos de cada fase. No Capítulo 4 é apresentado um estudo de caso para ilustrar a abordagem de desenvolvimento proposta. Finalmente, no Capítulo 5, são apresentadas as conclusões deste trabalho.

2. Revisão Bibliográfica

2.1. Considerações Iniciais

Este capítulo descreve as tecnologias envolvidas no desenvolvimento deste trabalho. Na Seção 2.2 são descritas técnicas para análise de domínio, bem como explicações sobre os conceitos envolvidos. Na Seção 2.3 é apresentado o modelo de *features*. Na Seção 2.4 é apresentada a abordagem FODA, que integra análise de domínio na criação de linhas de produtos. Na seção 2.5 é apresentada a tecnologia de linhas de produtos de software, com algumas das técnicas e abordagens para aplicá-la. Na Seção 2.6, abordam-se os conceitos referentes ao paradigma de programação orientada a aspectos. Na Seção 2.7 é apresentada a abordagem Tema para análise e projeto orientado a aspectos. Por fim, na Seção 2.8 são feitas as considerações finais deste capítulo.

2.2. Análise de Domínio

Prieto-Díaz (1990) define a análise de domínio como o processo pelo qual a informação usada no desenvolvimento de sistemas de software é identificada, captada e organizada com o propósito de fazer com que ela seja reutilizada na criação de novos sistemas.

O termo análise de domínio foi definido por Neighbors (1981) como "a atividade de identificar os objetos e as operações de uma classe de sistemas similares no domínio de um problema em particular". Segundo ele, a diferença entre análise de domínio e análise de sistemas é que a análise de sistemas preocupa-se com ações específicas de um determinado sistema, enquanto que a análise de domínio preocupa-se com ações e objetos em uma área de aplicação.

Temos a seguir uma lista de termos básicos da análise de domínio definidos por Kang et al. (1990):

- Aplicação: Um sistema que fornece um conjunto de serviços genéricos para resolver algum tipo de problema do usuário.

- Contexto: As circunstâncias, situações ou ambiente no qual um particular sistema existe.
- Domínio (também chamado Domínio de Aplicação): Um conjunto de aplicações existentes ou que virão a existir que compartilham um conjunto de capacidades e dados em comum.
- Análise de Domínio: O processo de identificar, coletar, organizar e representar as informações relevantes em um domínio baseado no estudo de sistemas existentes, em suas histórias de desenvolvimento, no conhecimento obtido dos especialistas, na teoria por trás do domínio e na tecnologia envolvida no domínio.
- Engenharia de Domínio: Um processo que inclui a análise de domínio e a subseqüente construção de componentes, métodos e ferramentas que visam resolver os problemas do desenvolvimento de sistemas e subsistemas utilizando a aplicação dos artefatos da análise de domínio.
- *Feature*: Uma característica proeminente ou diferenciada, um aspecto perceptível pelo usuário ou característica de um ou mais sistemas de software.
- Arquitetura de Software: A estrutura de alto nível que agrupa dados, funções e suas interfaces e controles para apoiar a implementação de aplicações em um domínio.
- Reúso de Software: O processo de implementar novos sistemas de software usando informações de software existentes.
- Componente Reutilizável: Um componente de software (inclui requisitos, projetos, código, testes, dados, etc.) projetado e implementado para o propósito específico de ser reutilizado.
- Usuário: Uma pessoa ou uma aplicação que opera um sistema para executar uma tarefa.

A análise de domínio tem como objetivo o reúso. Uma compreensão mais abrangente de um domínio é necessária quando se vai desenvolver mais de um sistema. A compreensão mais detalhada do domínio serve principalmente para identificar padrões, semelhanças, componentes comuns e abstrações, visando o reúso. O custo de um estudo mais complexo compensa quando diminui o custo de desenvolvimento em larga escala (Czarnecki e Eisenecker, 2000). Também por isso, a análise de domínio

deve ser incluída em um processo de desenvolvimento. No desenvolvimento de linhas de produtos ela é essencial.

O conhecimento de um domínio aplicado ao desenvolvimento de software reside em várias formas de informação: da análise dos requisitos, passando pelos projetos até o código fonte. Outros artefatos também são muito importantes, como a documentação do código, histórico de decisões de projeto, planos de teste e manuais de usuário.

O objetivo da análise de domínio é facilitar o acesso a todo esse conhecimento para que um engenheiro de software possa facilmente escolher e reusar um componente. Para isso, ele precisa entender o contexto, a seqüência de decisões e as razões que levaram ao projeto do componente. Isso torna o reúso mais eficiente.

Uma grande melhora no reúso ocorre com a análise de domínio extraindo arquiteturas comuns, modelos genéricos e linguagens específicas de um determinado domínio. Isso é conseguido identificando-se características (*features*) comuns a um domínio de aplicações, selecionando e abstraindo os objetos e operações que definem essas características, e criando-se procedimentos que automatizam essas operações. Geralmente, isso é conseguido com o desenvolvimento de vários sistemas do mesmo tipo para um domínio. Enquanto se desenvolvem os vários sistemas em um domínio, a experiência e o conhecimento nesse domínio evoluem e, assim, várias abstrações podem ser isoladas e reutilizadas. Certas características são então padronizadas, isoladas e encapsuladas. O processo de análise de domínio resume-se em identificar e estruturar informação para o reúso.

Segundo Kang et al (1990), o sucesso da aplicação da análise de domínio depende de que ela consiga dois objetivos básicos e bem definidos:

1. O método conduz ao desenvolvimento de artefatos que apóiam a implementação de novas aplicações;
2. O método pode ser incorporado em um processo de desenvolvimento de software mais abrangente.

Neighbors (1981) diz que “a chave para o reúso de software é captada na análise de domínio que enfatiza o reúso da análise e do projeto, não do código”. Na Figura 2.1 ilustra-se a definição de análise de domínio.

Figura 2.1: Análise de Domínio (Prieto-Díaz, 1987)

Conceitos importantes envolvidos são agregação e generalização. Kang et al (1990) explica esses conceitos e também o de parametrização.

- Agregação é quando se abstraem vários conceitos em um, como por exemplo, motor, rodas, câmbio e etc… são agregados em "carro";
- Generalização é quando um conjunto de entidades são todas classificadas com o mesmo significado semântico que possuem em comum, por exemplo: azul, verde, vermelho são generalizadas em "cores";
- Parametrização é uma técnica de desenvolvimento de componentes na qual os componentes são adaptados de muitas maneiras pela substituição de parâmetros do componente.

De acordo com McCain (1985), o processo de análise de domínio tem três passos básicos:

- Identificação de entidades reutilizáveis;
- Abstração ou generalização;
- Classificação e catalogação para reúso posterior.

Arango (1988) propõe um método para análise de domínio no qual o reúso é um sistema de aprendizado. O processo de desenvolvimento de software é um processo que melhora a si próprio continuamente e gera uma infra-estrutura de reúso como fonte de conhecimento. Os novos sistemas são construídos a partir dessa infra-estrutura. Três papéis são fundamentais em uma infra-estrutura de reúso:

1. Um controlador de versões: cuida da biblioteca de reúso dos artefatos, mantendo o controle das versões de cada artefato, mantendo-os atualizados e disponibilizando-os para reúso;
2. Um gerenciador de artefatos: controla a qualidade, a padronização e a conformidade com as especificações dos artefatos;
3. Um gerenciador de recursos: ajuda no obtenção das informações relevantes e dos artefatos potenciais de reúso e coordena o esforço de reúso.

Prieto-Díaz (1987) faz um estudo sobre a análise de domínio e algumas abordagens propostas por outros autores. Ele mostra quais são as atividades mais importantes na análise de domínio tradicional.

- identificar componentes reutilizáveis;
- encapsular os componentes reutilizáveis;
- adaptar os componentes para reúso em diferentes contextos (por exemplo com parametrização);
- guardar os componentes em uma biblioteca; e
- usar os componentes da biblioteca na construção de novos sistemas.

Um processo comum para análise de domínio é explicado por Prieto-Díaz (1987). O processo é chamado Justificativa Literal (*literary warrant*). Ele faz a análise, classificando os termos do domínio. Nesse processo, os termos literais do domínio são colhidos para serem classificados e organizados em categorias. Sinônimos são identificados como sendo da mesma categoria. Os termos comuns são agrupados de acordo com seu significado semântico, além de importância. Por exemplo, em um domínio de biologia, teríamos os seguintes grupos:

1. fisiologia, respiração, reprodução;
2. tropical, deserto, montanhas, água salgada;
3. marinho, anfíbios;
4. fauna, animais, vertebrados, répteis, cobras, pássaros, peixes, traças, mamíferos;

5. experimento, relatório.

2.3. Modelo de Features

O modelo de *features* é uma das técnicas mais bem sucedidas para fomentar a reutilização de artefatos de software desde os estágios iniciais do desenvolvimento. Um modelo de *features* é uma representação hierárquica que visa captar os relacionamentos estruturais entre as *features* de um domínio de aplicação.

Ele também representa as *features* comuns e variáveis de instâncias de conceitos (como, por exemplo, sistemas de software) e as dependências entre as *features* variáveis (Alves, 2005). Um modelo de *features* consiste de um diagrama composto de *features* e alguma informação adicional, tais como descrições semânticas de cada *feature*, pontos variáveis, motivos para cada *feature*, seus interessados e usuários, prioridades e regras de dependência.

No contexto de linhas de produto de software, um modelo de *features* representa a própria linha de produtos. Uma *feature* pode ser de um dos seguintes tipos:

- Obrigatória: A *feature* deve estar presente em todos os membros da linha de produtos.
- Opcional: A *feature* pode ou não estar presente em um membro da linha de produtos.
- Alternativa: É uma *feature* que é composta de um conjunto de *features* das quais escolhe-se uma ou mais. Deve-se indicar se é necessário escolher apenas uma ou se pode-se escolher mais de uma.

Um modelo de *features* é representado por uma árvore onde cada nó é uma *feature*, como ilustrado na Figura 2.2. As arestas da árvore são interpretadas junto com o tipo do nó e indicam a configurabilidade do nó, isto é, se a *feature* representada no nó é obrigatória, opcional ou alternativa. No modelo, uma *feature* obrigatória é representada por uma aresta terminada por um círculo preenchido. Uma *feature* opcional é representada por uma aresta terminada por círculo vazio. *Features* alternativas são representadas por arestas que estão ligadas e conectadas por um arco. Se o arco for vazio, deve-se escolher apenas uma das alternativas, se o arco for

preenchido, é permitido escolher mais de uma alternativa (notação de Van Deursen e Klint, 2002).

É comum utilizar a raiz da árvore para especificar o que está sendo modelado. As arestas que partem de uma *feature* para outra indicam uma relação de *feature* e sub-*feature*. As sub-*features*, se houverem, são *features* pertinentes à outra *feature*, ou seja, partes que compõem a *feature*. Sub-*features* só podem estar presentes na composição de um produto se a *feature* à qual pertencem também estiver.

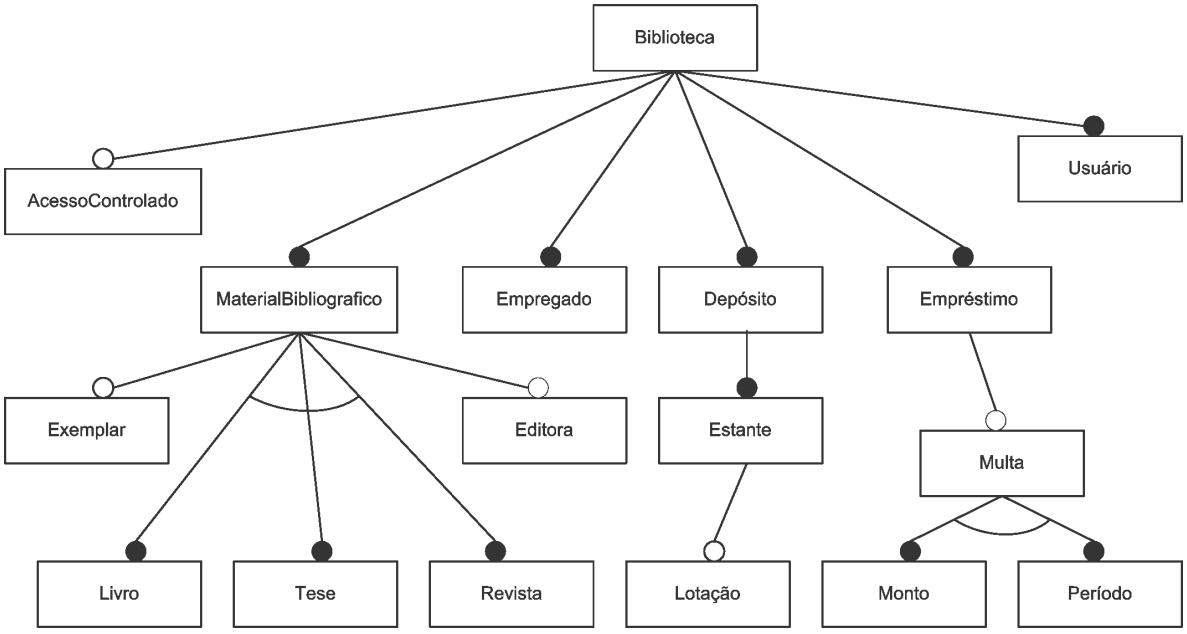


Figura 2.2: Exemplo de modelo de *features* (traduzido de Aragón, 2006)

2.4. FODA

A abordagem FODA (do inglês *Feature Oriented Domain Analysis*), desenvolvida e documentada pela SEI, é um método para descobrir e representar semelhanças entre sistemas de software relacionados (Kang et al, 1990). Ele procura captar o conhecimento dos especialistas, codificando os processos de raciocínio usados para desenvolver sistemas de software em um domínio. A análise de domínio ajuda o reúso de software, captando conhecimento no domínio, além de apoiar a comunicação, treinamento, desenvolvimento de ferramentas, projeto e especificação de software.

A abordagem FODA consegue reúso de sucesso, pois trata de maneira sistemática as semelhanças entre os sistemas de um domínio. As semelhanças dão origem a *features*

que são utilizadas em vários produtos do domínio. A análise de domínio permite que se saibam os requisitos comuns e os padrões para sua implementação.

O princípio da abordagem é a identificação de *features* proeminentes dos sistemas de software em um domínio específico. As *features* são características que se pode observar nos sistemas ou no próprio domínio. Uma vez preparadas, elas permitem a criação de diversos produtos no domínio. *Features* podem ser características comuns do domínio bem como diferenças entre os sistemas. O domínio é definido em termos de *features* classificadas como obrigatórias, opcionais ou alternativas.

A abordagem FODA se divide em três atividades básicas:

1. Análise de contexto: Define os limites de um domínio para análise. Nessa fase, o analista de domínio interage com os usuários e especialistas no domínio para estabelecer os limites e um escopo apropriado para a análise. O analista também procura fontes de informação para fazer a análise. Os produtos dessa fase fornecem o contexto do domínio. Para isso, é necessário representar as entradas e saídas do software e também identificar algumas interfaces do software.
2. Modelagem do domínio: Descreve os problemas do domínio que são tratados pelo software. O analista de domínio usa as fontes de informação e outros produtos da análise de contexto para ajudar na criação de um modelo de domínio. Este modelo é revisado pelo usuário do sistema, pelo especialista no domínio e pelo analista de requisitos. Os produtos dessa fase descrevem os problemas que os sistemas do domínio tratam. Eles fornecem:
   - As *features* dos softwares no domínio;
   - O vocabulário padrão no domínio;
   - A documentação das entidades envolvidas no software; e
   - Os requisitos mais gerais do software documentados formalmente.
3. Modelagem da arquitetura: Cria a arquitetura de software que implementa uma solução para os problemas no domínio. O analista de domínio usa o modelo de domínio para produzir um modelo da arquitetura. Esse modelo é então revisto pelo especialista no domínio, pelo analista de requisitos e pelo engenheiro de software. A estrutura da implementação do software é definida nessa fase. As representações geradas fornecem aos

desenvolvedores um conjunto de modelos de arquitetura para construção de aplicações e mapeamento dos modelos do domínio para as arquiteturas. As arquiteturas podem guiar o desenvolvimento das bibliotecas de componentes reutilizáveis que são utilizados nas futuras composições de software.

As fases da abordagem FODA e os seus respectivos produtos são mostrados na Figura 2.3.

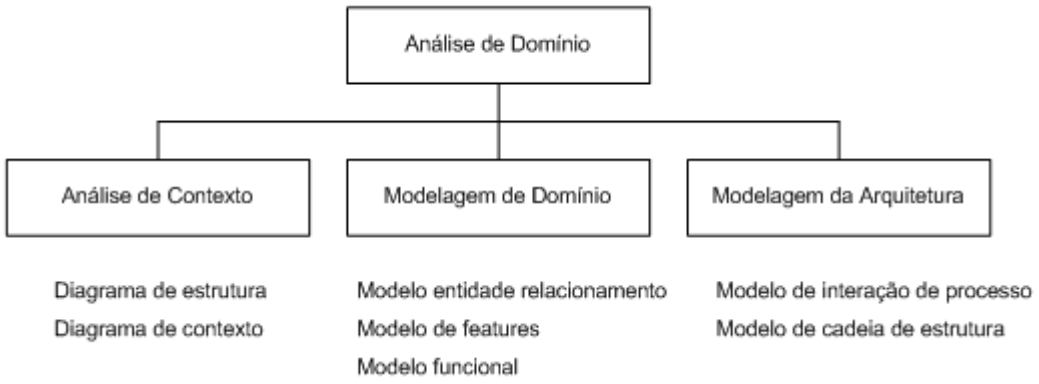


Figura 2.3: Fases e produtos da abordagem FODA (traduzido de Kang et al., 1990)

Os produtos da análise de domínio são usados pelo analista de requisitos e pelo projetista de software na implementação de um novo sistema. Os produtos são então combinados para produzir a especificação e o projeto do sistema. Isso traz feedback para o analista e especialista de domínio modificar ou aumentar a análise de domínio e também melhorar o processo de análise como um todo. Na Figura 2.4 é mostrado como se dá todo esse desenvolvimento e na Tabela 2.1 resume-se as atividades com as entradas e saídas de cada atividade e mostra-se como uma saída de uma atividade é usada em outras atividades.

A generalidade dos componentes é conseguida abstraindo fatores que fazem uma aplicação diferente das outras. O método FODA aplica os conceitos de agregação e generalização para captar semelhanças entre as aplicações do domínio. Parâmetros são usados para especificar o contexto para cada refinamento específico. O propósito da parametrização é desenvolver componentes genéricos que podem ser adaptados de muitas maneiras de acordo com valores passados para os parâmetros. O método FODA aplica este conceito a outros produtos da engenharia de software, incluindo requisitos e projetos.

Tabela 2.1: Resumo da abordagem FODA

Entradas	**Atividade**	**Produto**	**Descrição**
Ambientes operacionais, padrões	Análise de contexto	Modelo de contexto	Ambiente no qual as aplicações serão usadas e operadas
Features, modelo de contexto	Análise de *features*	Modelo de *features*	Perspectiva do usuário final das capacidades da aplicação em um domínio
Tecnologia do domínio, Modelo de contexto, Modelo de *features*, Modelo entidade-relacionamento, Requisitos	Análise Funcional	Modelo de fluxo de dados	Perspectiva do analista de requisitos da funcionalidade da aplicação
		Modelo de máquina de estados finitos	
Tecnologia de implementação	Implementação	Modelo de interação de processo	Perspectiva do projetista da arquitetura do sistema

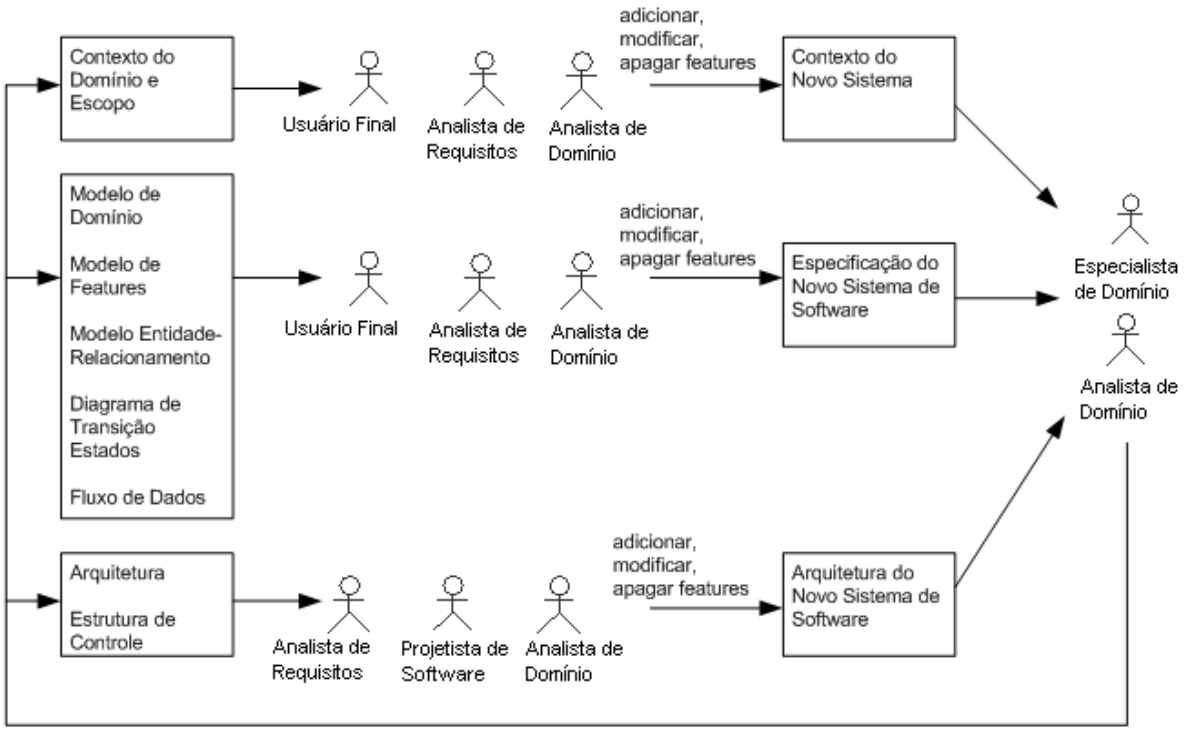


Figura 2.4: Utilização dos artefatos da análise de domínio e melhoria do processo e dos artefatos (traduzido de Kang et al., 1990)

A análise de domínio da abordagem FODA pode ser integrada em outros processos de desenvolvimento de software para:

- Apoiar o entendimento do domínio;
- Implementar aplicações no domínio;
- Criar recursos reutilizáveis como componentes ou projetos; e
- Criar a análise de domínio e outras ferramentas de reúso.

2.5. Linhas de Produtos

Linha de produtos – LP (do inglês, *Product Line*), também conhecida como família de produtos, é uma técnica que permite o reúso de software, em razão das características comuns entre vários softwares que podem ser desenvolvidos para um mesmo domínio. Podem existir vários sistemas com o mesmo propósito ou de mesmo tipo para um domínio, que possuem muitas características comuns. Os sistemas de uma linha são construídos a partir de componentes pré-fabricados. Para as características ou partes comuns entre os sistemas são reaproveitados os mesmos componentes e, para as chamadas "variabilidades", são desenvolvidos componentes específicos. Exemplos de variabilidades são o sistema operacional ao qual se destinam, funcionalidades presentes ou não em um produto da linha, o modo como as funcionalidades são implementadas ou o modo como são utilizadas. Na Figura 2.5 ilustra-se como uma biblioteca de componentes pode ser utilizada para criar vários produtos de uma linha.

De acordo com Griss (2001) e Bass et al. (1998), uma linha de produtos é um conjunto de produtos que compartilham um conjunto comum de requisitos, mas também exibem variabilidade significativa nos requisitos. Weiss e Lai (1999) definem uma linha de produtos de software como uma família de produtos projetada para tirar vantagem de suas características comuns e variabilidades previstas. Os membros de uma linha de produtos são sistemas no mesmo domínio; seu desenvolvimento e manutenção independentes, normalmente exigem esforço intensivo e redundante.

A engenharia de linhas de produtos, ou simplesmente linhas de produtos, é uma abordagem para reúso que garante métodos para planejar, controlar e melhorar a infra-estrutura de reúso para o desenvolvimento de uma família de produtos similares (John e Muthig, 2002).

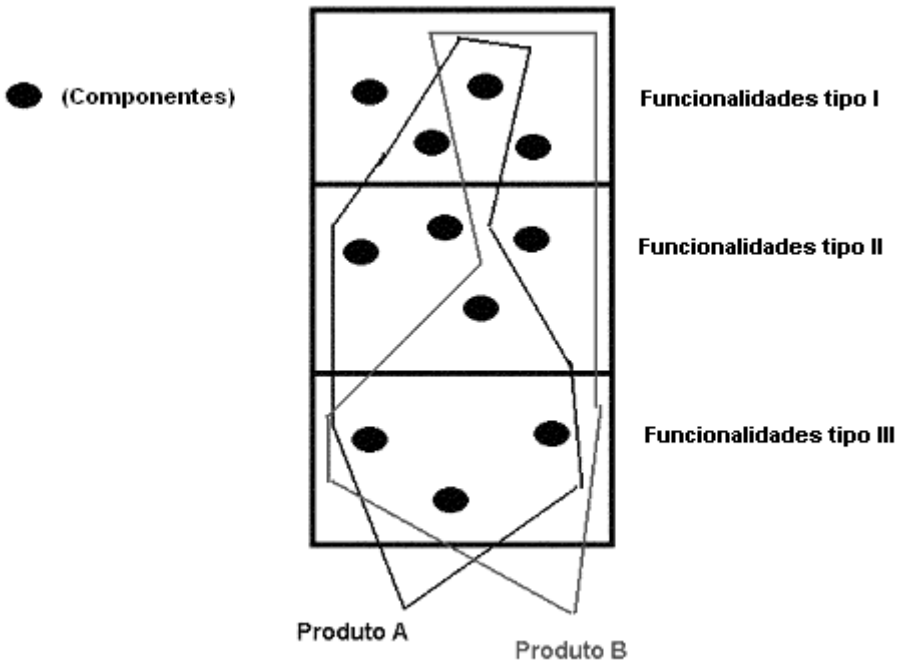


Figura 2.5: Dois produtos de uma linha construídos a partir de componentes em comum.

Em setores como o da indústria aeroespacial, automotiva e de componentes eletrônicos, as técnicas de linhas de produtos já vêm sendo exploradas há muito tempo (Travassos e Gurov, 2002). Na última década as linhas de produtos de software também alcançaram um amplo reconhecimento na indústria de software. Muitas organizações adotaram ou estão pensando em adotar essa tecnologia (Bosch, 2000). Por causa do aumento do interesse em linhas de produtos, muitos trabalhos recentes estão explorando as possibilidades de utilizações de linhas de produtos. Ziadi et al. (2004) desenvolveram um perfil UML (*Unified Modeling Language*) (Rational, 2000) próprio para linhas de produtos. Lee et al. (2002) e Bayer et al. (2000) apresentaram trabalhos para transmitir suas experiências com a utilização de linhas de produtos e servirem de guia para a utilização de linhas de produtos. Gomaa (2004) propôs um método para modelagem de domínios com UML, por meio de produtos pré-existentes para o desenvolvimento de uma linha de produtos.

O objetivo das abordagens de linhas de produtos é alcançar um reúso planejado e específico para um domínio, construindo-se uma família de aplicações ao invés de desenvolver os produtos separadamente (John e Muthig, 2002). Com isso, pode-se alcançar as vantagens do reúso de maneira ainda melhor por ser sistemática, pois não é

ao acaso que os componentes são necessários novamente, mas sim por causa das semelhanças entre os produtos.

Muitas vantagens são obtidas utilizando-se linhas de produtos. Componentes comuns são reusados muitas vezes e, além disso, reparos e correções de defeitos em um produto podem rapidamente ser propagados para outros membros da linha de produtos, pelo uso de componentes compartilhados (Griss, 2000).

O desenvolvimento de software baseado em componentes de linhas de produtos tem potencial para (Griss, 2001):

- Reduzir significativamente o custo e o tempo de desenvolvimento de sistemas de software, permitindo que sistemas sejam construídos a partir de componentes reusáveis ao invés de iniciar do zero;
- Melhorar a confiabilidade de sistemas de software, porque existe a tendência de que com o reúso, cada componente tenha passado por várias revisões e inspeções durante o seu desenvolvimento e usos prévios; e também porque o Desenvolvimento de Software baseado em Componentes utiliza uma arquitetura e interfaces bem definidas;
- Melhorar a manutenibilidade de sistemas de software, permitindo que novos componentes de melhor qualidade substituam outros;
- Melhorar a qualidade de sistemas de software, permitindo que especialistas em um domínio desenvolvam componentes e, posteriormente, engenheiros de software especializados em desenvolvimento de software baseado em componentes utilizem esses componentes para montar um sistema de software;
- Aumentar a capacidade das organizações em atender mudanças no mercado, permitindo que novos produtos sejam construídos rapidamente desenvolvendo-se somente alguns novos componentes e reutilizando o resto.

De acordo com Bergey et al. (2001), a maior dificuldade em trabalhar com linhas de produtos é a mudança cultural que ocorre pelo fato dos desenvolvedores estarem acostumados a trabalhar com o desenvolvimento de um sistema por vez, além das dificuldades para o extenso mapeamento do domínio, construção de uma arquitetura básica e definição do conjunto de componentes.

Para Bosch (2000), a maior dificuldade é a escolha da abordagem correta para o desenvolvimento de linhas de produtos, que se não for bem escolhida pode levar a outros problemas, tais como:

- Inconsistência entre os componentes desenvolvidos e necessidades dos produtos;
- Componentes mal projetados;
- Dificuldades para a construção das interfaces dos componentes;
- Dificuldades organizacionais com atribuição de papéis e responsabilidades;
- Gerenciamento incorreto do conhecimento;
- Evolução dos componentes sem modificar as interfaces.

Existem várias abordagens que podem ser adotadas para o desenvolvimento de linhas de produtos, que são descritas brevemente a seguir.

A iniciativa PLP (*Product Line Practice*)

Desenvolvida pelo *Software Engineering Institute* (SEI), a iniciativa PLP (SEI/CMU, 2006) envolve atividades relacionadas ao desenvolvimento de artefatos centrais e desenvolvimento e gerenciamento do produto utilizando esses artefatos centrais. Para a execução dessas tarefas é necessária a definição de áreas de trabalho mais gerenciáveis, com conjuntos menores de atividades. Cada área de trabalho[1] possui um plano de trabalho e métricas associadas que permitem acompanhar sua execução e avaliar o sucesso dos trabalhos realizados.

Usualmente, essas áreas de trabalho produzem artefatos concretos que levam à criação, de alguma forma, dos artefatos centrais que serão utilizados na linha de produtos. É uma descrição de abordagem similar a do CMM (*Capability Maturity Model*) (SEI/CMU, 2006). A estratégia PLP agrupa as áreas de trabalho em três áreas principais:

- Engenharia de Software: necessária para aplicar a tecnologia apropriada à criação e evolução dos artefatos ou componentes centrais do produto;
- Gerenciamento Técnico: relacionado à criação e evolução dos artefatos ou componentes centrais;

[1] Tradução do inglês escolhida para *Practice Área*.

- Gerenciamento Organizacional: utilizada para o gerenciamento dos esforços relacionados a toda a linha de produtos.

Na Figura 2.6 é ilustrado o relacionamento entre as três atividades essenciais da abordagem da SEI para linhas de produtos. Com essa figura, a SEI procura demonstrar que todas as atividades são essenciais e estão ligadas, além de procurar demonstrar que elas podem ocorrer em qualquer ordem, com uma puxando a outra. Outra característica do método que a SEI procura expressar na figura é que tanto os artefatos podem ser usados para gerar produtos, como os produtos podem gerar artefatos e ambas as atividades são influenciadas pela atividade de gerenciamento (SEI/CMU, 2006).

Figura 2.6: As três atividades essenciais para linhas de produtos de software (traduzido de SEI/CMU, 2006)

A versão atual da iniciativa PLP define todas as atividades para as áreas de trabalho e está evoluindo continuamente com experiências obtidas de diferentes estudos de caso realizados com a colaboração existente entre o SEI e as organizações que estudam a utilização de linha de produtos, além de contribuições da comunidade de software.

O método PLUS

O método PLUS (do inglês *Product Line UML Based Software Engineering*) (Gomaa, 2004) é um método para projeto de linhas de produtos baseado na UML. A

ênfase do reúso é dada aos requisitos e arquitetura, devido ao fato de a codificação representar uma porcentagem pequena do custo de um projeto. Assim, as técnicas de linhas de produtos são utilizadas para gerenciar esses artefatos.

No método PLUS é feita distinção entre casos de uso e *features*. Casos de uso são orientados ao desenvolvimento, pois determinam os requisitos funcionais das linhas de produtos. *Features* são orientadas ao reúso, pois organizam os resultados de uma análise de semelhanças e variabilidades em uma linha de produtos.

Atualmente o método PLUS é utilizado no contexto de um processo, o ESPLEP (do inglês *Evolutionary Software Product Line Engineering Process*). O ESPLEP é um modelo de processo de software que elimina a distinção tradicional entre desenvolvimento e manutenção de software (Gomaa, 2004). Ao invés disso, o software evolui por meio de várias iterações. Assim, os sistemas desenvolvidos são capazes de se adaptar a mudanças nos requisitos durante cada iteração. Para o desenvolvimento de linhas de produtos de software, as atividades de modelagem de requisitos, análise e projeto são substituídas por atividades de desenvolvimento que aumentam a linha de produtos.

O ESPLEP é constituído de duas fases principais, conforme exibido na Figura 2.7:

1. Engenharia de Linhas de Produtos: Durante essa fase, as semelhanças e variabilidades na linha de produtos são analisadas do ponto de vista geral dos requisitos. Essa atividade consiste do desenvolvimento de um modelo de casos de uso da linha de produtos, modelo de análise da linha de produtos, arquitetura da linha de produtos e componentes reutilizáveis. Os testes são feitos nos componentes, bem como nas configurações da linha de produtos. Os artefatos produzidos são armazenados em um repositório;
2. Engenharia de aplicações: Durante essa fase, um membro da linha de produtos é desenvolvido. Ao invés de iniciar do zero, é feito uso dos artefatos do repositório. Dados os requisitos da aplicação individual, o modelo de casos de uso da linha de produtos é adaptado para derivar o modelo de caso de uso da aplicação. A arquitetura da linha de produtos é adaptada para derivar a arquitetura da aplicação. Tendo a arquitetura da aplicação e os componentes apropriados do repositório, pode-se instanciar a aplicação desejada.

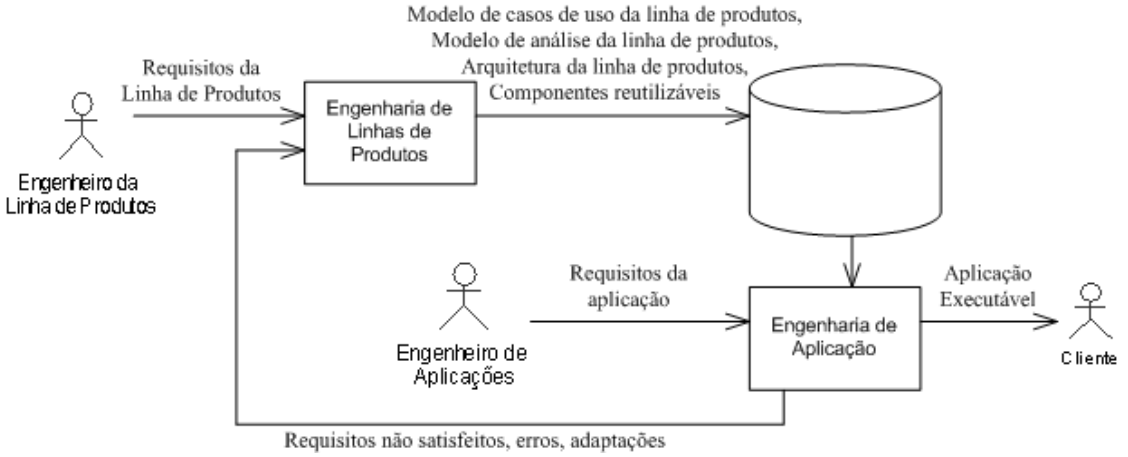


Figura 2.7: ESPLEP (traduzido de Gomaa, 2004)

As atividades principais do processo ESPLEP, conforme exibido na Figura 2.8, são:

1. Modelagem de requisitos da linha de produtos de software: Um modelo de requisitos consistindo de um modelo de casos de uso e um modelo de *features* é desenvolvido. O modelo de casos de uso define os requisitos funcionais em termos de atores e requisitos funcionais. Esse modelo é estendido para indicar as semelhanças e variabilidades da linha de produtos e então os casos de uso são classificados como obrigatório, opcional ou alternativo;
2. Modelo de análise da linha de produtos de software: São feitos modelos estáticos e dinâmicos. O modelo estático define o relacionamento estrutural entre as classes do domínio. Ele é constituído de um diagrama de classes em que as semelhanças e variabilidades são mostradas com a classificação das classes em obrigatórias, opcionais ou variantes. Devem ser anotadas dependências entre as classes, já que elas podem estar em *features* diferentes e pode haver dependência entre classes; também é constituído do modelo de casos de uso que mostra os objetos que participam em cada caso de uso. Os modelos dinâmicos são constituídos por diagramas de estado e diagramas de interação;
3. Modelo de projeto da linha de produtos de software: A arquitetura da linha de produtos de software é projetada e o modelo da análise é mapeado para um ambiente operacional;
4. Implementação dos componentes de software incremental: Um subconjunto da linha de produtos é selecionado para ser implementado em cada incremento.

Deve-se começar com os casos de uso obrigatórios, seguidos pelos opcionais e alternativos. Para cada caso de uso desenvolvido é feito o projeto, implementação e testes;

5. Teste da linha de produtos: Consiste de teste de unidade e testes de integração.

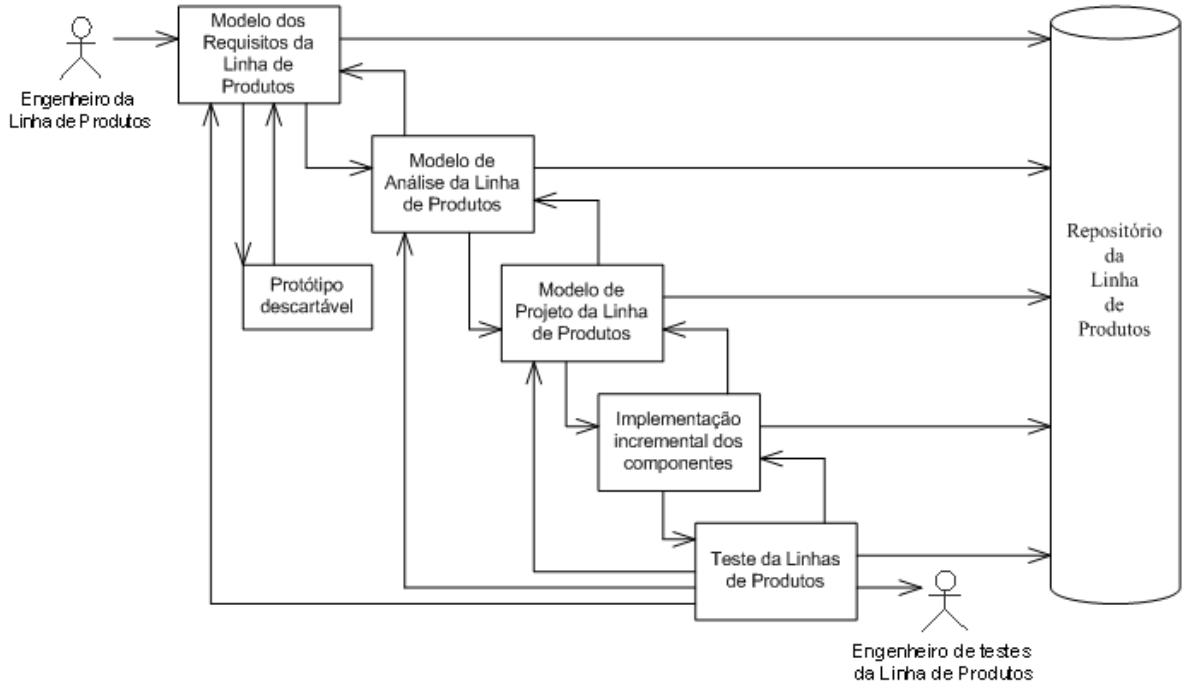


Figura 2.8: Atividades do processo ESPLEP (traduzido de Gomaa, 2004)

2.6. Programação Orientada a Aspectos

Interesses[2] (*concerns* em inglês) em software podem ser pensados como requisitos funcionais ou não funcionais que são úteis ou precisam estar presentes nos sistemas. Interesses podem ser desde noções de alto nível, tais como segurança e qualidade de serviço, a noções de baixo nível, como *caching* ou *buffering* (Elrad et al., 2001a).

Interesses transversais (do inglês, *crosscutting concerns*) são interesses cujo código espalha-se pelas divisões ou módulos do programa. Por exemplo, na programação orientada a objetos tem-se a divisão em classes, e o código de certos interesses pode ficar espalhado por várias classes. Por "atravessar" ou "cortar" várias classes ele é dito ser transversal (ou ortogonal).

[2] A tradução dos termos relativos à POA segue sugestão contida no site http://twiki.im.ufba.br/bin/view/AOSDbr/TermosEmPortugues

A Programação Orientada a Aspectos - POA (do inglês *Aspect-Oriented Programming*) é um paradigma de programação relativamente novo, tendo surgido para complementar o paradigma da programação orientada a objetos (POO) que não permite a separação de interesses em certas situações, como para alguns requisitos não funcionais. Segundo Elrad et al. (2001a), a POA é baseada na idéia de que sistemas de computador são melhor programados especificando os vários interesses de um sistema, bem como alguma descrição dos seus relacionamentos. A POA tenta enfatizar interesses espalhados como elementos de primeira classe, ou seja, elementos principais do sistema.

Na literatura da área, o termo interesse é muitas vezes substituído pelo termo aspecto. O termo aspecto foi definido por Kiczales et al. (1997) para se referir aos interesses de um sistema.

A POA permite que se visualizem e se compreendam os vários aspectos, as várias características ou pontos de vista do sistema, permitindo que esses aspectos sejam melhor isolados. Aquilo que for de interesse dos desenvolvedores do sistema pode ser isolado como um aspecto.

Nas palavras de Kiczales et al. (1997): "O objetivo da programação orientada a aspectos é tornar possível lidar com aspectos ortogonais de um sistema o mais separadamente possível. Queremos permitir à programadores expressar cada um dos aspectos de interesse de um sistema de forma isolada e natural, e então, automaticamente combinar essas descrições separadas no executável final".

A seguir apresentam-se alguns problemas e limitações da programação orientada a objetos que motivaram o surgimento da programação orientada a aspectos.

Separação de interesses

Um dos problemas com POO é que todo o comportamento relativo a um interesse muitas vezes não fica localizado em um único lugar. É difícil localizar interesses, pois eles estão envolvidos com várias variáveis e com o comportamento de vários objetos (Elrad et al., 2001a). Esse problema é conhecido como o problema da separação de interesses, e está relacionado com o problema do entrelaçamento ou espalhamento de código, explicado a seguir.

Entrelaçamento e Espalhamento

De acordo com Elrad et al. (2001b), dois interesses se entrecortam se os métodos relacionados a eles se interceptam, ou seja, existe código tanto de um quanto de outro interesse dentro do mesmo ou dos mesmos métodos e, assim, os métodos se tornam menos específicos e menos compreensíveis. É importante entender também que interesses transversais não podem ser separados claramente um do outro na programação convencional orientada a objetos (OO). Justamente pelo fato do código ficar espalhado por vários métodos e misturado com código relativo a outros interesses, fica difícil separar e isolar o código relativo a apenas um único interesse. Assim, é fácil entender as dificuldades quando é necessário visualizar ou fazer manutenção de um interesse (ou aspecto) separadamente. Seria necessário procurar por todo o código ou alterá-lo em muitos lugares.

Decomposição dominante

Outro problema com a POO é que, do ponto de vista dos interesses, um sistema dificilmente pode ser dividido claramente. Cada interesse tem sua forma, e se um deles for destacado no código, dando ênfase a isolar todo o código relativo a ele, dificilmente outro poderá ser destacado (Elrad et al., 2001a e Kiczales et al., 1997). Isso é conhecido como o problema da decomposição dominante.

Segundo Ossher e Tarr (2001), no paradigma da POO, quando lidamos com um interesse que é um objeto do sistema, todo o código relativo a esse objeto pode ser encapsulado em forma de uma classe. Assim, todo código relativo a esse interesse é centralizado. Porém, outros tipos de interesses não podem ser efetivamente representados no paradigma OO. Interesses como segurança ou *buffering*, por exemplo, não se restringem a uma classe apenas.

A maneira como o programa divide-se em classes impõe uma estrutura no software que torna difícil ou impossível agregar em um único lugar de maneira efetiva outros tipos de interesses, como características ou regras de negócios. Essa tirania resulta em problemas associados a uma fraca separação de interesses, tais como evolução cara e difícil; baixo reúso; integração complicada e software muito sensível a detalhes da aplicação específica.

Para entender como os problemas da orientação a objetos são tratados pela orientação a aspectos, é necessário conhecer os principais elementos comuns nas diversas abordagens para POA (Elrad et al., 2001a):

Pontos de junção

"Pontos de junção" (do inglês *join points*) são pontos bem definidos na execução de um programa, como por exemplo, chamada a um método, execução de um método ou acesso ao valor de um atributo. Assim, são locais no código fonte que possivelmente podem ser afetados semanticamente por outros interesses, alterando o comportamento do programa.

Os pontos de junção são o elemento chave da orientação a aspectos, porque graças a eles é possível separar cada comando do código de um software e fazer composições entre os comandos, inclusive acrescentando outros comandos com trechos de código escritos separadamente.

Conjunto de junção

As abordagens para POA devem permitir que o desenvolvedor especifique pontos de junção. Um conjunto de pontos de junção é chamado conjunto de junção (do inglês *pointcut*), e é uma das construções mais básicas da POA, já que são os conjuntos de junção que definem na prática onde o programa será afetado.

Um meio de especificar o comportamento nesses pontos

As abordagens para POA devem permitir que o desenvolvedor especifique comportamento, ou seja, escreva o código que será utilizado nos pontos de junção. Esse código, chamado de "adendo" (do inglês *advice*), deve permitir especificar não apenas instruções a serem realizadas antes ou depois de outras, como também a serem realizadas no lugar de outras ou paralelamente.

Declarações intertipos

Declarações intertipos (do inglês *inter-type declarations*) são declarações de atributos ou métodos que são feitas em algum local do aspecto, mas que no código resultante será como se elas estivessem declaradas na classe entrecortada.

Unidades encapsuladas combinando conjuntos de junção, adendos e declarações intertipos

Esse conceito depende da abordagem, mas comumente é conhecido como aspecto (do inglês *aspect*). Como os pontos de junção dividem o software em um nível mais baixo, é possível e desejável para os desenvolvedores que todas as instruções relativas a

um interesse sejam colocadas em um só lugar. Assim, são alcançados os objetivos da modularização e da separação de interesses, com os benefícios descritos anteriormente.

Um método para acrescentar essas unidades ao programa (varia dependendo da abordagem)

O código desenvolvido com orientação a aspectos é todo separado por interesses. A separação é feita com o auxílio das unidades encapsuladas mencionadas anteriormente. De alguma maneira, o comportamento do programa final deve conter o comportamento original mais o comportamento especificado pelos aspectos, ou seja, todos os comportamentos especificados em separado posteriormente irão atuar juntos. A composição resulta em código OO normal, com os problemas descritos anteriormente, porém agora a única fase que resta é a compilação. Existem várias maneiras de compor o programa final, dependendo da tecnologia.

Para o desenvolvimento deste trabalho foi escolhida a linguagem AspectJ (AspectJ, 2006) como linguagem orientada a aspectos para complementar a linguagem orientada a objetos Java, a qual foi utilizada. AspectJ é uma linguagem OA de propósito geral, formulada como uma extensão de Java para trabalhar com aspectos (Elrad et al., 2001b). AspectJ possibilita que se definam conjuntos de junção e também adendos para os pontos de junção desses conjuntos.

Os pontos de junção são definidos com os operadores *call* e *execution*. O operador *call* especifica pontos de junção no local da chamada a algum método e o operador *execution* indica um ponto de junção dentro da execução de algum método.

O conjunto de pontos de junção, ou simplesmente conjunto de junções, e os adendos são encapsulados em unidades denominadas Aspectos, que são um novo elemento construtivo da linguagem.

Os adendos são definidos com os operadores *before*, *after* ou *around*, que definem onde o código do adendo irá agir. O operador *before* determina que o adendo irá agir antes dos locais definidos pelo conjunto de junção. O operador *after* determina que o adendo irá agir depois dos locais definidos pelo conjunto de junção. Por fim, o operador *around* determina que o adendo irá agir antes e depois dos locais definidos pelo conjunto de junção.

Na programação utilizando o AspectJ, os objetos são implementados em classes, enquanto os interesses transversais são implementados em aspectos e , assim, o código que antes se espalhava pelo programa é removido das classes e passa para os aspectos.

As classes ficam contendo somente código relativo ao comportamento inerente àquela classe, ou ao interesse que essa classe implementa.

Os interesses transversais são programados cada um em um Aspecto. Esses Aspectos "sabem", graças à definição dos pontos de junção, quando devem interferir na execução do programa e executar o comportamento definido nos adendos do Aspecto. Para o AspectJ existe um compilador próprio, denominado Ajc, que faz a composição[3] (do inglês *weaving*) e, em seguida, a compilação.

No exemplo a seguir (Elrad et al., 2001b), tem-se o código não orientado a aspectos, relativo às classes (escritas em Java) `Linha` e `Ponto`, mostrado nas Figuras 2.9a e 2.9b. Nesse exemplo, quando se move um ponto ou um ponto de uma linha, é necessário que se atualize a visualização dos mesmos em um outro objeto denominado `Visualizador`, o que é feito por meio da chamada do método `visualizador.atualiza()`.

O Aspecto `AtualizacaoVisualizador`, mostrado na Figura 2.10, declara um conjunto de pontos de junção, chamando `mover()`, que define como pontos de junção os métodos `set()` de `Linha` e `Ponto`. Em seguida, é definido o comportamento nesses pontos, justamente a atualização da visualização. O resultado final é que as linhas em negrito nas classes `Linha` e `Ponto` podem ser removidas.

(a)

```
class Ponto {

 private int x, y;

 int getX() { return x; }
 int getY() { return y; }

 void setX(int x){

  this.x = x;
  visualizador.atualiza();
 }
 void setY(int y){
  this.y = y;
  visualizador.atualiza();
 }
}
```

(b)

```
class Linha {

 private Ponto p1, p2;

 Ponto getP1() {
  return p1; }

 Ponto getP2() {
  return p2; }

 void SetP1(Ponto p1){
  this.p1 = p1;
  visualizador.atualiza();
 }
 void SetP2(Ponto p2){
  this.p2 = p2;
  visualizador.atualiza();
 }
}
```

Figura 2.9: Classes de um editor de figuras (Elrad et al., 2001b)

[3] Processo no qual o código dos aspectos e o código do programa principal são combinados de maneira a formar o programa final com os comportamentos resultantes esperados.

(c)

```
aspect AtualizacaoVisualizador {
 pointcut mover():
 call(void Ponto.set*(int)) ||
 call(void Linha.set*(Ponto))

 after() returning: mover(){
  visualizador.atualiza();
 }
}
```

Figura 2.10: Aspecto intercepta os métodos (Elrad et al., 2001b)

2.7. Temas

O desenvolvimento de software orientado a aspectos (DSOA) está emergindo como uma abordagem aprovada para permitir a expressão separada de múltiplos interesses no desenvolvimento de software.

Um grave problema relacionado ao DSOA era a falta de trabalhos a respeito do tratamento dos princípios de aspectos nas fases iniciais do desenvolvimento. Em todo desenvolvimento de software é importante que o desenvolvedor possa projetar o sistema e mapear o projeto para o código (Clarke and Baniassad, 2005).

A grande dificuldade na análise e projeto orientado a aspectos é saber quais serão os aspectos. Alguns aspectos são óbvios e já bastante conhecidos como *logging* ou *debugging*, mas pode ser difícil identificar os aspectos específicos de uma aplicação em particular (Clarke and Baniassad, 2005).

Para poder encapsular o comportamento transversal em aspectos, o desenvolvedor precisa primeiro identificá-los a partir dos requisitos. Isso é difícil, pois por natureza os aspectos estão misturados com outros comportamentos e costumam ser descritos em várias partes do documento de requisitos.

Abordagens ad-hoc ou com base na experiência dos desenvolvedores não são adequadas. Tentar programar usando orientação a objetos e usar aspectos para as funcionalidades que não se encaixarem, também não é uma boa idéia, pois assim os aspectos não são projetados, eles apenas “aparecem” quando se começa a codificar (Clarke and Baniassad, 2005).

As técnicas de análise e projeto orientadas a objetos também não são adequadas, pois os elementos de projeto são modularizados em classes, interfaces e métodos. Para alguns interesses isso é suficiente, mas para interesses transversais que podem surgir de

características do sistema, pode não ser suficiente. Isso reduz a compreensibilidade e capacidade de mapeamento, fazendo com que o projeto seja difícil de desenvolver, reutilizar e estender (Clarke, 2004).

Para identificar aspectos no início do ciclo de vida do software e estabelecer um mapeamento adequado, os desenvolvedores precisam de apoio para análise e identificação de aspectos na documentação de requisitos, independentemente da linguagem de programação orientada a aspectos que for escolhida.

A abordagem Tema surge então para apoiar a análise e o projeto orientado a aspectos e para resolver os problemas e dificuldades mencionados acima. De acordo com Clarke e Baniassad, (2005) a abordagem Tema permite ver os relacionamentos entre os comportamentos em um documento de requisitos, identificar e isolar aspectos nos requisitos e modelar esses aspectos utilizando uma linguagem de projeto.

A palavra "tema" não deve ser considerada um sinônimo de "aspecto". Temas são mais gerais do que aspectos. Cada pedaço de funcionalidade ou requisito ou aspecto ou um interesse que um desenvolvedor possa ter pode ser um dos temas do sistema.

Um tema pode ser qualquer característica, ou interesse, ou requisito que deve estar presente no sistema. Um tema pode ser entendido como um elemento de projeto, ou seja, uma coleção de estruturas e comportamentos que representam uma característica ou um interesse (Clarke e Baniassad, 2005). Temas podem se relacionar uns com os outros do mesmo modo que requisitos ou recursos são relacionados com outras partes do sistema.

A abordagem Tema é dividida em dois níveis, para diferentes fases do ciclo de vida do software:

- **Tema/Doc (*Theme/Doc*):** é utilizada para a fase requisitos, fornecendo visões do texto da especificação de requisitos e expondo o relacionamento entre os comportamentos de um sistema. Ela permite ao desenvolvedor refinar a visão dos requisitos de modo a revelar quais funcionalidades do sistema são transversais, e onde elas se entrecortam.
- **Tema/UML (*Theme/UML*):** é utilizada no nível de projeto, possibilitando ao desenvolvedor projetar as características e capacidades de um sistema e especificar como elas devem ser combinadas.

A abordagem traz também a vantagem de permitir a compreensão dos requisitos do ponto de vista dos aspectos. Ela permite que os requisitos sejam mapeados (característica conhecida em inglês como *traceability*) corretamente para um projeto com aspectos, ou seja, mesmo os requisitos transversais são mapeados e também modelados corretamente em aspectos. Nas palavras de Clarke e Baniassad (2005): "A abordagem Tema ajuda a manter um mapeamento dos requisitos para o projeto, já que os requisitos são mapeados diretamente para as visões da Tema/Doc, as quais são mapeadas diretamente os modelos da Tema/UML".

Existem dois tipos de temas: os temas-base, que compartilham parte da estrutura e comportamento de outros temas base no modelo, mas exibem ambos, estrutura e comportamento, de acordo com sua própria perspectiva, ou seja, um tema-base pode mostrar parte de outros temas, mas apenas partes que sejam do interesse do seu próprio tema e do modo como for apropriado a seu próprio tema; e os temas transversais, cujo comportamento sobrepõe a funcionalidade dos temas-base. Temas transversais são aspectos.

Tema/Doc

No documento de requisitos, os aspectos se manifestam como descrições de comportamentos que são entrelaçados e interdependentes. Alguns aspectos podem ser óbvios, como especificações de comportamento transversal típico, enquanto outros podem ser mais sutis, sendo difíceis de identificar. Em todo caso, é difícil analisar os requisitos para localizar todos os pontos do sistema onde os aspectos deveriam ser aplicados. A abordagem Tema/Doc possibilita que se identifiquem os relacionamentos entre os comportamentos e se isolem aspectos nos requisitos.

No código, os aspectos são implementados como métodos e linhas de código misturados e espalhados. No documento de requisitos, os aspectos são descrições de funcionalidades misturadas e espalhadas. Uma funcionalidade misturada só tem sua descrição completa junto com a descrição de outras funcionalidades e uma funcionalidade espalhada é descrita ao longo do documento de requisitos. Assim, é difícil para o desenvolvedor decidir quais são os aspectos do sistema.

Antes que o desenvolvedor possa considerar como projetar e implementar aspectos em um sistema, ele deve poder identificar comportamento transversal. Além disso, ele deve poder ver e manipular o modo como os elementos de funcionalidades se

relacionam uns com os outros. Tema/Doc fornece visões do documento de especificação de requisitos, mostrando os vários comportamentos de um sistema. Essas visões ajudam a determinar quais elementos de funcionalidades são aspectos e quais não são (comportamento básico). Todas as visões são depois mapeadas para modelos Tema/UML (Seção 2.7.4) que fazem o projeto da aplicação.

A abordagem Tema/Doc trabalha com a premissa básica de que, se dois comportamentos são descritos no mesmo requisito, eles são relacionados. Esse relacionamento pode ser de três tipos: a) errado ou por coincidência, o documento de requisitos pode estar mal escrito e dar a entender que dois comportamentos podem se relacionar quando na verdade isso não deveria ocorrer, assim se o documento de requisitos for escrito mais cuidadosamente eles não aparecerão mais relacionados; b) hierarquicamente, quando um comportamento é um sub-comportamento de outro; e c) por entrecorte, de modo que o requisito descreve o comportamento de um aspecto (Clarke e Baniassad, 2005).

Tema/Doc fornece visões que expõem quais comportamentos são co-localizados nos requisitos. Essas visões ajudam o desenvolvedor a determinar que tipo de relacionamento existe entre comportamentos e se esses comportamentos são base ou aspectos. As visões fornecidas são as visões de ação e são vistas no exemplo a seguir. O exemplo trata de um pequeno sistema para avaliação de expressões. Os requisitos são dados pela Tabela 2.2.

Passo 1: Identificar Ações e Entidades

Os comportamentos relativos aos requisitos são identificados por um conjunto de palavras-chave fornecidas pelo desenvolvedor para a ferramenta Tema/Doc. Esses comportamentos são referidos como “ações”.

Procurar os verbos que indicam ações é um bom ponto de partida para encontrar temas. Requisitos que parecem não conter ações podem freqüentemente ser refinados para incluir ações. O desenvolvedor procura também um conjunto de entidades-chave. Todos esses elementos são usados para gerar as visões Tema/Doc.

A seguir, são descritos os quatro passos da abordagem Tema/Doc.

No exemplo, foram identificadas seis ações: `avaliação`, `exibição`, `determinar`, `checar a sintaxe`, `registrar` e `definida por uma gramática`. Também foram identificadas nove entidades: `Expressão`, `ExpressãoVariável`, `ExpressãoNumérica`, `OperadorAdição`,

`OperadorSubtração`, `OperadorAdiçãoUnário`, `OperadorSubtraçãoUnário`, `Mais` e `Menos`.

Tabela 2.2: Requisitos para o sistema de avaliação (Clarke e Baniassad, 2005)

R1.capacidade de avaliação, que determina o resultado da avaliação de uma expressão.
R2. capacidade de exibição, que descreve uma expressão textualmente.
R3. capacidade de checar a sintaxe, que determina se as expressões estão sintaticamente corretas.
R4. capacidade de registrar, que registra as avaliações mostradas e as atividades de checagem de sintaxe.
R5. uma expressão é uma definida por uma gramática como uma `ExpressãoVariável` ou uma `ExpressãoNumérica` ou um `OperadorAdição` ou um `OperadorSubtração` ou um `OperadorAdiçãoUnário` ou um `OperadorSubtraçãoUnário`.
R6. um `OperadorAdição` é definido por uma gramática como uma expressão, um sinal de adição e uma expressão.
R7. um `OperadorSubtração` é definido por uma gramática como uma expressão, um sinal de subtração e uma expressão.
R8. um `OperadorAdiçãoUnário` é definido por uma gramática como um sinal de adição e uma expressão.
R9. um OperadorSubtraçãoUnário é definido por uma gramática como um sinal de subtração e uma expressão.
R10. uma `ExpressãoVariável` é definida por uma gramática como uma letra e uma expressão.
R11.uma `ExpressãoNumérica` é definida por uma gramática como um número e uma expressão.

Passo 2: Categorizar Ações em Temas

Na abordagem Tema, nem todas as ações são modeladas como funcionalidades separadas do sistema: algumas ações são sub-comportamentos de outras ações. Neste passo, identificam-se as características ou temas do sistema, definindo-se quais ações são importantes o bastante para serem modeladas separadamente. Para isso, é utilizada uma visão do Tema/Doc, denominada visão de ação (do inglês: *Action View*).

A visão de ação para o exemplo é mostrada na Figura 2.11. Uma visão de ação consiste de dois elementos: *ações*, mostradas como losangos e *requisitos*, mostrados como retângulos arredondados. Se uma ação é mencionada em um requisito, existe uma linha entre a ação e o requisito.

Essa visão é usada para definir se ações devem se tornar temas ou apenas comportamento (talvez métodos) dentro dos temas. Para decidir quais ações são importantes o suficiente para se tornarem temas, deve-se verificar para cada ação, se faz sentido tê-la como características do sistema. Caso não valha a pena, elas são removidas da visão de ação. As ações restantes serão os temas.

No exemplo, a característica de `registrar` é válida para ser um tema, pois é algo que talvez se deseje colocar ou remover do sistema, ou pelo menos, modelar separadamente das outras visões.

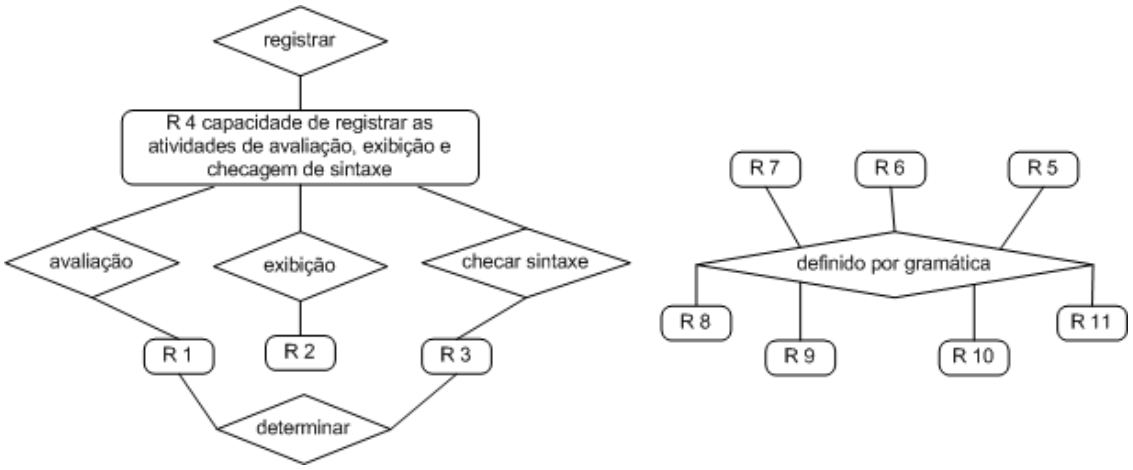


Figura 2.11: Visão de Ação (Baniassad e Clarke, 2004)

A ação `determinar` não é tão forte para que deva estar presente no sistema. Ela está envolvida em dois requisitos, mas de maneira secundária e é difícil imaginá-la como uma coleção de classes. Ela é considerada mais como um sub-comportamento, um método, ao invés de uma característica.

Passo 3: Identificar Temas Transversais

A seguir, a visão de ação é usada para determinar quais temas são base e quais são transversais. A visão de ação aparada é feita a partir das principais ações da visão de ação. Tendo a visão de ação, deve-se observá-la para determinar quais requisitos são compartilhados por mais de um tema. Se um requisito é compartilhado por mais de um tema, deve-se decidir qual deles deve fornecer a funcionalidade. Requisitos compartilhados indicam que aspectos podem ter sido encontrados no sistema, já que eles implicam que dois temas não podem operar sem precisar do comportamento um do outro.

O requisito R4 é compartilhado por vários temas. Deve-se então checar os requisitos para assegurar que não há requisitos vagos ou mal escritos, ou mesmo requisitos ocultos. Assim, deve-se checar para verificar se os requisitos podem ser reescritos em vários requisitos que se refiram a um tema cada.

No caso do exemplo, a única forma de reescrever R4 seria dividir o requisito em três sentenças, que ainda mencionariam `registrar` associado com outro tema (por exemplo, `capacidade de registrar a atividade de avaliação`).

Não há maneira de reescrever que consiga um relacionamento 1-1 entre os temas e os requisitos: a funcionalidade de registro deve ser coberta pela avaliação, exibição e checagem de sintaxe. Isso significa que o tema `registrar` é um aspecto.

Denota-se que o tema `registrar` entrecorta os outros três temas associados com o requisito compartilhado R4. Para isso, removem-se as ligações de R4 para os outros três temas e coloca-se uma seta indicando um relacionamento de composição, partindo do tema aspecto para os temas base (Figura 2.12).

O desenvolvedor deve garantir que o relacionamento 1-1 é conseguido. Se um requisito compartilhado é ambíguo demais para tornar a associação clara, então o desenvolvedor deve rever os requisitos e resolver a ambigüidade.

No exemplo, não há mais requisitos compartilhados, assim pode-se continuar a examinar os temas individualmente e planejar o modelo. O produto deste processo é denominado "visão de ação aparada" (do inglês *clipped action view*). As setas indicam comportamento transversal.

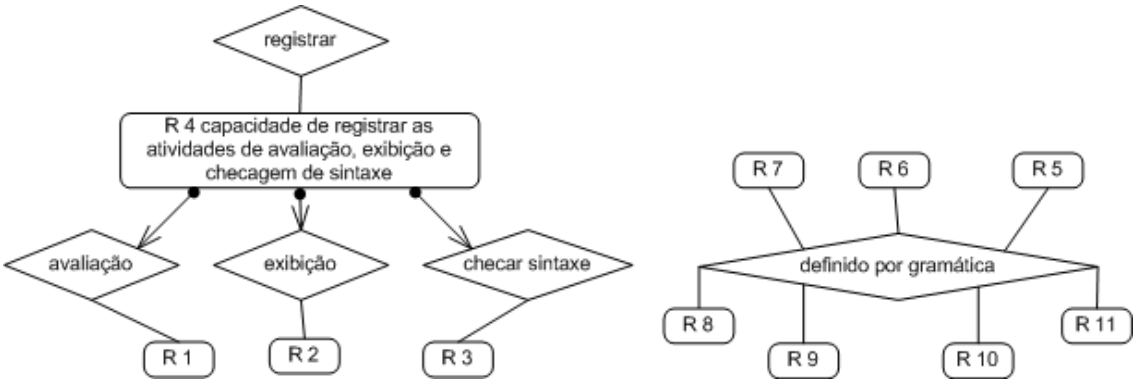


Figura 2.12: Visão de Ação Aparada (Baniassad e Clarke, 2004)

Passo 4: Visualizar os Temas Individuais

Os Temas podem ser visualizados individualmente ou agrupados nas "visões de ação". A "visão tema" mostra um tema individualmente e mostra os requisitos associados com a ação principal e as ações menores mencionadas nos requisitos. Entidades-chave também são mostradas individualmente nesta visão por meio de retângulos. Essas visões são usadas para checar se as associações foram feitas corretamente manipulando a visão de ação principal para formar a visão de ação

aparada. As visões também podem ser usadas para determinar como os temas devem ser modelados utilizando Tema/UML (Seção 2.5.4).

Theme/UML

A Tema/UML apresenta uma abordagem para modelar sistemas com base no modelo de orientação a objetos e fazendo uma extensão dele, adicionando novas capacidades de decomposição (Clarke, 2004). Cada modelo contém seu próprio tema, ou seja, o projeto de um requisito individual, com conceitos do domínio (que podem aparecer em múltiplos requisitos) modelados a partir da perspectiva daquele requisito. Os modelos dos temas individuais podem tratar de requisitos sobrepostos ou requisitos que se entrecortam. A UML padrão é usada para modelar partes decompostas do sistema. Extensões da UML são requeridas para a composição dos modelos temáticos. Isso é conseguido com um relacionamento de composição, que especifica como os modelos são compostos, identificando-se os conceitos sobrepostos e os conceitos compartilhados que estão em diferentes modelos e, também, explica como os modelos devem ser integrados.

Os modelos de temas individuais devem ser compostos para que se visualize o projeto do sistema todo. Para isso, a Tema/UML possui elementos denominados relacionamentos de composição, que especificam como os modelos devem ser compostos. Com os relacionamentos de composição um analista pode (Clarke, 2002):

- **Identificar e especificar sobreposições:** Onde a decomposição permitir sobreposições nos diferentes modelos de projeto, recursos de composição correspondentes devem fornecer meios para indicar onde as sobreposições estão. Para integrar modelagens separadas, os elementos de modelagem sobrepostos (elementos correspondentes e que, portanto, devem ser integrados em uma única unidade do sistema) são especificados com relacionamentos de composição;
- **Especificar como os modelos devem ser integrados:** Os modelos podem ser integrados de diferentes maneiras, dependendo do motivo pelo qual eles foram modularizados de determinado modo. Por exemplo, se diferentes modelos foram feitos separadamente para tratar diferentes requisitos, um modelo composto no qual todos os requisitos são incluídos, deve ser obtido com uma integração – isso significa que todos os elementos de projeto são importantes na modelagem

integral. Alternativamente, se uma modelagem contém o projeto de um requisito que se refere a uma mudança de um requisito previamente modelado (por exemplo, se uma regra de processo mudar), então essa modelagem pode substituir a modelagem prévia. Nesse caso, a integração por meio de uma estratégia de sobreposição é apropriada, na qual elementos de projeto existentes são substituídos por elementos novos;

- **Especificar como conflitos em elementos correspondentes são reconciliados:** Especificar as regras para a integração, especificando os elementos correspondentes que devem ser integrados em uma única modelagem. Existem várias possibilidades de reconciliação – por exemplo, um modelo de projeto pode tomar precedência sobre outro, ou valores padronizados podem ser usados.

A Tema/UML possui artefatos completos. Um tema já pode ser utilizado como um projeto de software e de fato ele representa um projeto de uma parte do software referente a um determinado aspecto. Na Figura 2.13 pode-se ver um exemplo de um Tema transversal. Um tema transversal é um tema que contém aspectos que interceptam o comportamento de outros temas. Temas transversais possuem métodos gabarito. Os métodos gabarito reservam lugar nos diagramas para as operações que serão interceptadas. Como o projeto da *feature* transversal diz respeito somente a seu interesse, ele não tem conhecimento de qual método ou quais classes serão entrecortadas. As classes e métodos que serão entrecortadas irão aparecer somente na composição dos temas, que é uma fase posterior da abordagem Tema.

Uma vez que se tenham os temas, é preciso compô-los para que se chegue ao projeto do sistema completo, assim como os aspectos são compostos para conseguir a aplicação completa. Os temas são compostos com regras de composição bem definidas, que especificam elementos que se sobrepõem e elementos que se interceptam. Na Figura 2.14 está ilustrado um exemplo de diagrama de composição especificando uma regra para composição de três temas.

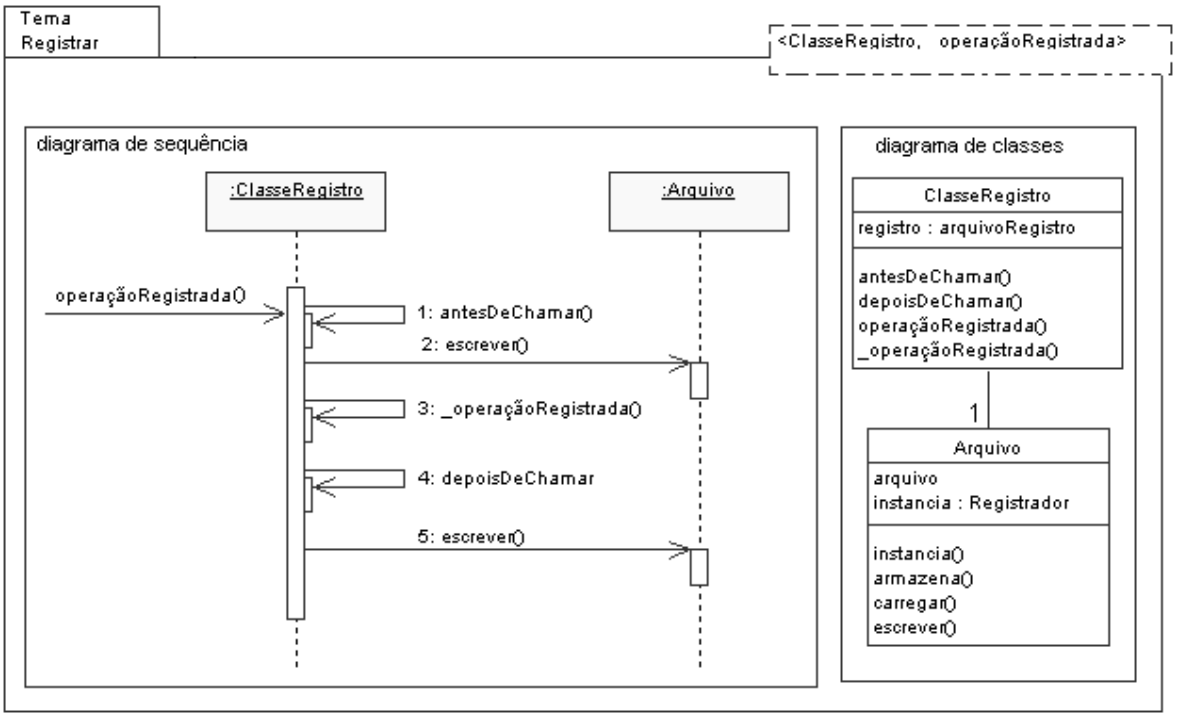


Figura 2.13: Tema Registrar (traduzido de Clarke, 2004)

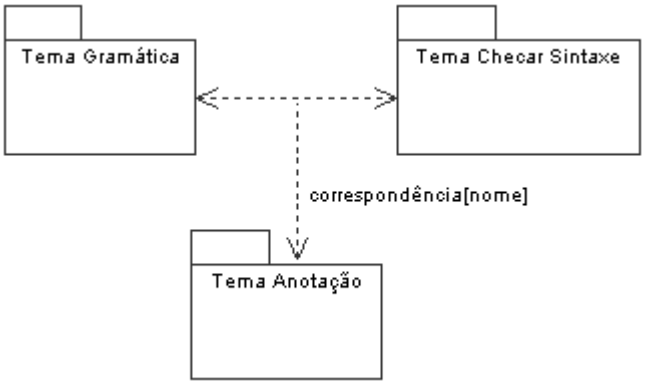


Figura 2.14: Relacionamento de composição – Conceitos compartilhados (traduzido de Clarke, 2004)

2.8. Considerações Finais

Este capítulo apresentou várias tecnologias utilizadas para promover o reúso de software, solucionar problemas do desenvolvimento e trazer mais qualidade ao software. Linhas de produtos podem ser implementadas com uma combinação de várias das técnicas existentes atualmente.

A orientação a aspectos proporciona melhor qualidade aos componentes, pois procura tratar da separação de interesses e, melhorando os componentes, pode-se

melhorar todas as outras técnicas que utilizam componentes. Essas possibilidades de melhorias vêm sendo objeto de estudos atuais, por ser um campo ainda em aberto.

Outro campo de estudo atualmente são as técnicas de análise e projeto orientado a aspectos. Pelo fato dos aspectos comportarem-se de maneira muito diferente da dos objetos, a análise e projeto orientados a objetos não é adequada para utilização com aspectos. Uma das técnicas para análise e projeto orientados a aspectos é a abordagem Tema, que foi descrita neste capítulo.

A abordagem Tema tem vantagens e desvantagens. Partes do processo da abordagem foram utilizadas neste trabalho, por serem consideradas adequadas às necessidades de linhas de produtos, principalmente a parte de análise Tema Doc e os artefatos visão Tema. Outras partes da abordagem, como os gabaritos dos temas, apesar de serem propostas para indicar os locais onde haverá interceptação por aspectos, não foram aqui utilizadas. A utilização dessas técnicas se mostrou inviável no presente trabalho, já que a abordagem Tema considera aspectos na granularidade de requisitos, enquanto que o presente trabalho considera aspectos na granularidade de *features*.

Assim, no Capítulo 3 é proposta uma abordagem incremental para desenvolvimento de linhas de produtos de software utilizando orientação a aspectos. Aspectos podem trazer às LPs o benefício de modularizar as suas *features*. Isso porque os aspectos conseguem a separação de interesses. Assim, se as *features* da LP forem tratadas como interesses de um sistema, elas podem ser implementadas com POA e dessa maneira podem ser combinadas de maneira mais fácil para criar os produtos. A capacidade dos aspectos de alterar o comportamento do sistema de maneira não invasiva é que traz esse benefício.

Outro benefício que pode ser conseguido com a utilização dos aspectos é a capacidade de desenvolver a LP de maneira incremental. Pelo mesmo motivo, a facilidade de composição proveniente da POA, pode-se desenvolver a LP em iterações e a cada nova iteração, as novas *features* podem ser combinadas com as anteriores sem necessidade de manutenção no código existente.

3. Abordagem Incremental Usando Aspectos

3.1. Considerações iniciais

Neste capítulo, é apresentada a abordagem proposta neste trabalho para construção de linhas de produtos orientadas a aspectos. A UML é utilizada como ferramenta de modelagem, combinada com artefatos da notação provenientes da Tema/Doc (Clarke e Baniassad, 2005) e algumas contribuições sugeridas pelo autor do presente trabalho. Parte da abordagem, a análise de domínio, foi criada a partir de revisão da literatura e é semelhante aos processos existentes estudados na revisão bibliográfica (Prieto-Díaz, 1987; Kang et al., 1990), porém foram reunidos em uma única proposta aqui apresentada.

A outra parte da abordagem é referente à implementação. Nessa parte, orienta-se como implementar a linha de produtos para garantir a modularização das *features* e garantir que a linha de produtos seja incremental, isto é, possa crescer com o tempo, sem a necessidade de previsão de todas as *features* no início do desenvolvimento. A orientação para implementação foi criada a partir da experiência de desenvolvimento de uma linha de produtos orientada a aspectos (ver estudo de caso – Capítulo 4).

A próxima seção fornece uma visão geral da abordagem. O restante deste capítulo está organizado como segue: A Seção 3.3 descreve a fase de Análise de Domínio, sendo dividida em subseções que explicam os passos e a criação dos artefatos. A seção 3.4 trata do desenvolvimento da base da linha de produtos. A seção 3.5 explica o desenvolvimento das *features*, com detalhamento sobre projeto e implementação. Na Seção 3.6 é abordado o desenvolvimento de produtos. Na Seção 3.7 são apresentadas as considerações finais do capítulo.

3.2. Visão Geral da Abordagem

A abordagem proposta neste trabalho para criação de linhas de produtos incrementais orientadas a aspectos é ilustrada na Figura 3.1. Ela se divide em três fases:

1. Análise de Domínio
2. Desenvolvimento da Base

3. Desenvolvimento dos Produtos

A abordagem se inicia com a Análise de Domínio, que deve ser feita para proporcionar o entendimento necessário para a construção da linha de produtos. Ela irá propiciar a descoberta do que a linha de produtos deve abranger e o que é esperado dos produtos da linha.

A segunda fase da abordagem é o Desenvolvimento da Base. A base da linha de produtos é a implementação do conjunto de todas as *features* básicas, entendendo-se por básicas as *features* que devem estar presentes em todos os produtos da linha.

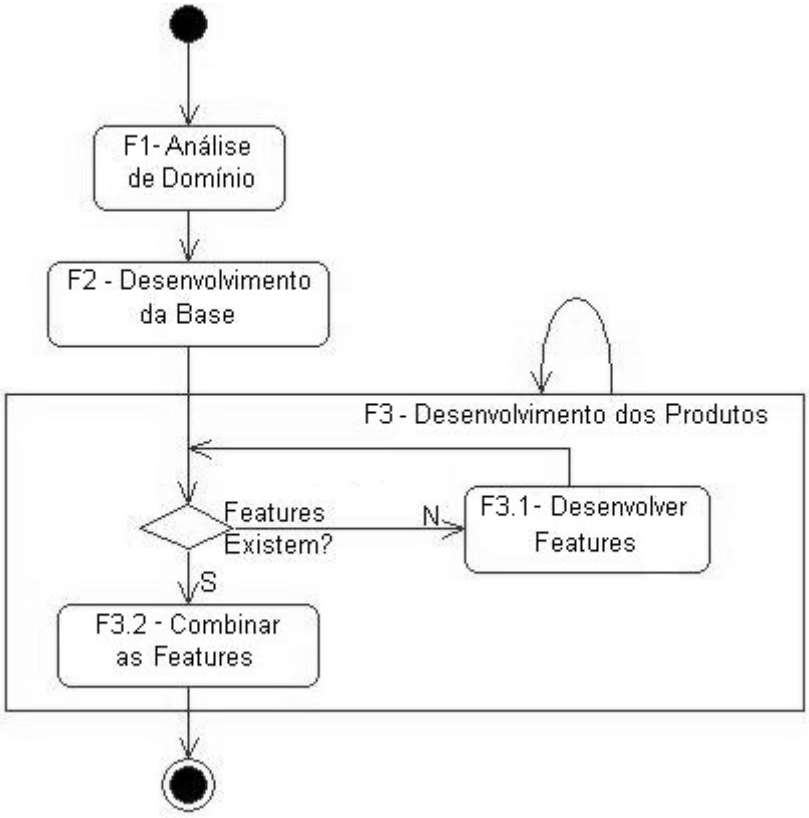


Figura 3.1: Abordagem para desenvolvimento de uma linha de produtos incremental

O desenvolvimento das *features* básicas deve ser feito de forma a permitir que as *features* que serão desenvolvidas posteriormente possam ser compostas com a base. Para isso, é utilizada uma arquitetura específica e técnicas de desenvolvimento e implementação descritas detalhadamente na Seção 3.5.2. Os artefatos produzidos durante a análise de domínio são utilizados para o desenvolvimento da base.

A fase de Desenvolvimento dos Produtos é aquela em que são criados os produtos da linha. A criação dos produtos é feita por meio da combinação de *features*.

Inicialmente, só existem as *features* básicas e, por isso, é necessário desenvolver as novas *features* e fazer sua composição com as da base. As *features* desenvolvidas são armazenadas em um repositório, então, novos produtos podem ser desenvolvidos também sem o desenvolvimento de novas *features*, ou seja, apenas com a utilização das *features* do repositório. As *features* são desenvolvidas sob demanda do produto atualmente sendo desenvolvido.

Existe também a possibilidade de desenvolver novos produtos criando variações das *features* existentes. As variações de uma *feature* também devem ir para o repositório da linha de produtos para poderem ser reusadas futuramente.

Deve-se ressaltar que, embora a fase de desenvolvimento dos produtos seja a fase final do desenvolvimento da linha de produtos, ela é iterativa, podendo haver várias iterações enquanto houver necessidade de instanciar produtos da linha.

Como a linha de produtos é incremental, não há forte necessidade de definição do escopo da linha de produtos no início do desenvolvimento. Pode-se projetar e implementar poucas *features*, e deixar que a linha de produtos cresça incrementalmente de acordo com a demanda por novas *features*. A utilização de aspectos nesse contexto proporciona facilidade para adição de novas *features*, evitando ter que projetar e desenvolver novamente partes da base da LP.

3.3. Análise de Domínio – F1

O objetivo principal da análise de domínio é identificar as funcionalidades necessárias aos sistemas do domínio. Ela fornece um entendimento do domínio como um todo, o qual é necessário para a construção de sistemas. Esse conhecimento do domínio deve ser documentado de forma apropriada para o desenvolvimento da linha de produtos.. Devem ser identificadas as características e funcionalidades desejadas nos sistemas em termos de *features*. Conforme foi visto na Seção 2.4 da revisão bibliográfica, existem *features* obrigatórias, opcionais e alternativas. É importante que se descubram principalmente as *features* obrigatórias e também aquelas que tem grande chance de serem necessárias posteriormente. Secundariamente, procura-se descobrir *features* mais incomuns. É necessário impor um limite nesse processo de descoberta de *features* e isso é feito com a definição do escopo (Prieto-Diáz, 1987). A definição do escopo define a abrangência da linha de produtos, ou seja, quais *features* serão

fornecidas e quais requisitos serão atendidos inicialmente pela linha de produtos. Posteriormente, nas fases subseqüentes da abordagem proposta, o escopo pode aumentar, já que a linha de produtos é incremental. Por isso, a definição do escopo não precisa ser muito rígida nesta fase, não sendo necessário tentar prever todas as funcionalidades necessárias ou estudar o domínio muito a fundo. A definição do escopo inicial é feita ao longo da fase de análise de domínio.

A análise de domínio tem atividades específicas para o estudo do material existente. Dentre elas pode-se destacar as principais:

1. Leitura ou estudo de material (bibliográfico e outros tipos) sobre o domínio;
2. Entrevistas com especialistas do domínio;
3. Entrevistas com usuários dos sistemas;
4. Coleta de requisitos;
5. Estudo de sistemas do domínio.

Essas atividades não são todas essenciais e durante o desenvolvimento deste trabalho considerou-se que as mais importantes são a 2 e a 5. O analista de domínio pode decidir quais atividades e quão profundamente vai realizá-las, dependendo das necessidades e recursos disponíveis no momento. O importante na fase de análise de domínio é que ao final sejam produzidos os seguintes artefatos de saída:

- documento de *features* de domínio;
- documento de requisitos;
- modelo de *features*;
- relação *features*-requisitos;
- visão de ação;
- visão de ação aparada.

Devem ser escolhidos alguns sistemas existentes no domínio para realizar o estudo. É bom escolher sistemas que tenham amplo uso, se for possível, pois isso indica que o sistema é bem aceito e, sendo assim, tem maior chance de se conseguir um bom modelo. A quantidade de sistemas que devem ser analisados depende do domínio apresentar ou não muita diferença entre um sistema e outro. Quanto mais os sistemas forem parecidos, menos sistemas precisam ser analisados. É esperado que os sistemas

possuam as principais funcionalidades em comum, já que pela definição de domínio, são agrupados sistemas que abordam os mesmos problemas.

A análise de domínio e a criação dos artefatos são atividades iterativas. Os artefatos podem ser criados e depois aumentados e melhorados com o tempo. Para a análise de domínio, podem ser usadas quaisquer das atividades de estudo de material do domínio mencionadas ou ainda outras atividades, determinadas pelo analista de domínio, que forem mais convenientes para o entendimento do domínio. O documento de *features* é o primeiro artefato que deve começar a ser criado, mas os demais artefatos podem ser criados concomitantemente.

Na Figura 3.2 (notação BMPL – *Business Process Modeling Language*, Arkin, 2002) é explicada a fase de análise de domínio. A relação a seguir resume os artefatos da análise de domínio e uma possível seqüência de atividades. A seqüência não impõe uma ordem fixa para a criação dos artefatos. Alguns deles podem ser feitos concomitantemente com outros. Pode-se fazer partes dos documentos e depois complementá-los de acordo com o que for mais apropriado ao analista, ou de acordo com o que estiver mais acessível no momento em cada caso.

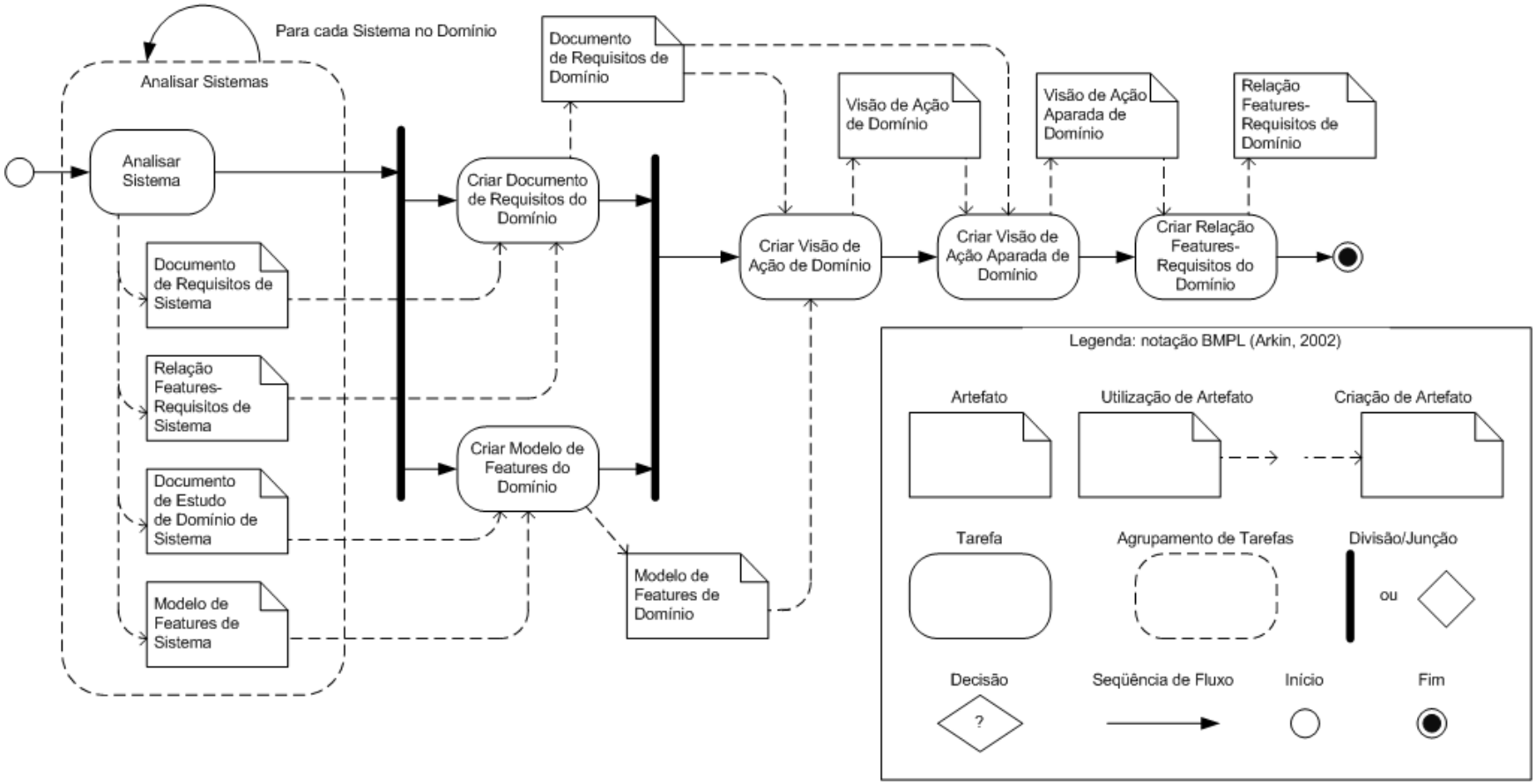


Figura 3.2: Fase de Análise de Domínio

Como pode ser visto na Figura 3.2, o processo inicia com o estudo de um ou mais sistemas do domínio criando, para cada um deles, um documento de *features*, um documento de requisitos, um modelo de *features* e uma relação *features*-requisitos. Essas versões são chamadas versões individuais, por se referirem a uma aplicação específica. Outras atividades de estudo de domínio podem ser utilizadas para complementar a criação desses documentos, bem como para auxiliar eventualmente na análise de domínio, ou mesmo nas demais fases da abordagem de criação da LP.

Para completar a análise de domínio é produzida uma versão final, chamada versão do domínio, de cada um dos artefatos, sendo que essas versões do domínio abrangem e organizam tudo o que havia nas versões individuais.

A próxima sub-Seção descreve os artefatos da análise de domínio.

3.3.1. Documento de *Features*

O documento de *features* (DF) ajuda no entendimento do domínio, na criação dos outros artefatos e no processo de análise de domínio em geral.

O DF deve conter pré-*features*. São chamadas de pré-*features* pois, nesse estágio, ainda são candidatas a *features*, ou seja, ainda não se tem certeza de que essas *features* levantadas inicialmente serão as *features* definitivas da linha de produtos.

Além de listar as *features*, o documento de *features* deve conter diversas informações sobre elas. Não é obrigatório colocar todas as informações, mas quanto mais completo estiver o documento, melhor será para o processo. As principais informações são: quais são as pré-*features*, suas relações, seus atributos, suas funcionalidades e sua descrição.

O DF descreve de maneira textual as *features* inicialmente identificadas no domínio e a relação delas com as sub-*features*. Neste trabalho, essa relação é mostrada na forma de uma hierarquia, indicando as *features* e sub-*features* com numeração e sub-numeração. As *features* alternativas também precisam ser indicadas com letras associadas aos números para indicar as alternativas.

Na Tabela 3.1 são exibidos exemplos de partes de um documento de features. Na primeira linha podem ser vistas pré-*features*, sendo que duas são sub-*features* da primeira. Na segunda linha tem-se um exemplo de duas *features* alternativas. Na terceira linha são exibidas duas *features*, sendo que não são sub-*features* da primeira.

Tabela 3.1: Trechos de um Documento de Features

1. Jogador **1.1. Agenciador** **1.2. Passe**
4.a. Pagamento à Vista **4.b. Pagamento à Prazo**
X. Partida **Y. Estádio**

Para criar o documento de features, neste trabalho, é utilizado um método semelhante ao processo conhecido por *Literary Warranty* apresentado por Prieto-Díaz (1987), no qual o material bibliográfico a respeito do domínio é analisado e são procurados os conceitos que aparecem recorrentemente ou para os quais parece haver necessidade de que sistemas de informação para o domínio manipulem seus dados.

Conceitos, segundo Larman (2000), são abstrações de entidades físicas ou mesmo idéias e informações que existem no mundo real. Essa mesma definição é aplicada na análise de domínio e a maioria dos conceitos são fortes candidatos a *features*. Pelo fato de muitas vezes haver a necessidade de que os sistemas de informação manipulem os dados referentes aos conceitos, provavelmente existirá uma *feature* que irá gerenciar a manipulação dessas informações.

Esse processo de análise também permite que se descubram as relações entre as *features*, quais são sub-*features* e quais são alternativas. Muitas atividades de análise de domínio podem ajudar nessa tarefa, como por exemplo, as entrevistas com os especialistas e com os usuários.

O documento de *features* pode ser criado de maneira iterativa. O passo inicial é fazer um levantamento das *features* e sub-*features*. Uma segunda análise mais atenta do material do domínio pode trazer quais são as informações ou atributos mais importantes para cada *feature*. As próximas iterações podem esclarecer o que é esperado das funcionalidades relativas às *features* e uma descrição das *features*. Esse tipo de informação também ajuda a registrar as decisões de projeto (termo conhecido em inglês como *design rationale*).

Uma atenção especial deve ser dada à análise dos sistemas existentes. A análise dos executáveis é uma maneira simples para a identificação de *features* que a linha de produtos deve fornecer. Na Tabela 3.2 tem-se o exemplo de um trecho do documento de *features* com os atributos elicitados para a *feature* `Jogador`, bem como algumas funcionalidades e observações explicativas sobre a *feature*.

Tabela 3.2: Exemplo de um trecho de um documento de *features* com informações complementares

1. Jogador *Atributos:* nome, nascimento, sexo, endereço, bairro, CEP, cidade, estado, telefone, fax, e-mail, CPF e RG. *Funcionalidades*: inclusão, alteração, consulta e exclusão. *Observações*: Representa o registro dos dados pessoais sobre um jogador. Abrange documentação, endereço, contatos e descrição pessoal.

3.3.2. Modelo de *Feature*s – versão individual

O modelo de *features* - versão individual representa um dos sistemas que foram analisados no domínio. Sua criação ocorre em paralelo com a criação dos outros documentos que possuem versões para cada sistema, que são o documento de estudo de domínio, o documento de requisitos e o documento de mapeamento.

Para criar o modelo de *features*, as pré-*features* candidatas presentes no documento de *features* devem ser analisadas para verificar se irão se tornar *features* reais. Para cada *feature* candidata deve-se verificar:

- as funcionalidades anotadas no DF;
- as observações feitas no DF;
- os requisitos associados.

Procura-se verificar se a *feature* candidata possui uma ou mais funcionalidades necessárias a produtos da linha. Se isso ocorrer, a *feature* é validada e incluída no modelo. As sub-*feature*s e *feature*s alternativas que forem validadas também são incluídas de maneira correspondente no modelo. Como é feita a análise de um sistema por vez, inicialmente não existem *feature*s opcionais.

3.3.3. Documento de Requisitos – versão individual

A elicitação de requisitos é um processo já amplamente discutido na literatura e não convém ser discutido aqui. Para a criação de linhas de produtos, a análise de sistemas do domínio é uma das melhores maneiras de obtenção de requisitos. Os requisitos são ainda melhor definidos utilizando as demais fontes de informação sobre o

domínio, principalmente as entrevistas com os usuários, especialistas e o material específico sobre o funcionamento da organização que utiliza o sistema. Essas fontes também podem trazer ciência de requisitos não atendidos pelo sistema e ou atendidos erroneamente.

No documento de requisitos, os requisitos devem estar descritos textualmente de maneira clara e organizada. É interessante que os requisitos sejam quebrados em uma granularidade fina, para diminuir a possibilidade de misturar mais de um requisito em um único. Recomenda-se como boa prática de engenharia de software que se escolha algum padrão existente para documentação de requisitos. Neste trabalho os requisitos foram organizados segundo as práticas descritas por Turine e Masiero (1996) e o padrão IEEE (IEEE, 1984).

3.3.4. Relação *Feature*s-Requisitos – Versão Individual

A relação *feature*s-requisitos é um documento auxiliar que indica quais requisitos pertencem a cada *feature*. O objetivo desse documento é separar os requisitos por *feature*, já que se deseja fazer o desenvolvimento de cada *feature* de maneira independente. Isso ajuda na não dependência entre as *feature*s e na sua modularização. Esse documento é utilizado nas fases subseqüentes do projeto, o desenvolvimento do núcleo e o desenvolvimento dos produtos.

A versão individual desse documento abrange as *feature*s e os requisitos do sistema analisado no momento. As *feature*s dos sistemas são elucidadas ao mesmo tempo em que se identificam seus requisitos. A relação *feature*s-requisitos é criada paralelamente a esse processo. Como exemplo, considera-se os requisitos R1 a R5 exibidos na Tabela 3.3.

As *feature*s que haviam sido percebidas juntamente com os requisitos foram: `Jogo`, `Time` e `Estadio`. A relação *feature*s requisitos versão individual registra a menção das *feature*s pelos requisitos.

Tabela 3.3: Requisitos obtidos de um sistema analisado

R1: O sistema deve permitir a inclusão, consulta, alteração e exclusão de jogos com os atributos data e hora.
R2: O sistema deve permitir a inclusão, consulta, alteração e exclusão de times de futebol com os atributos nome e estado.
R3: O sistema deve permitir a inclusão, consulta, alteração e exclusão de dois times participantes de um jogo, indicando o nome dos times e a pontuação para cada um.
R4: O sistema deve permitir a inclusão, consulta, alteração e exclusão de estádios com os atributos nome, local e capacidade.
R5: O sistema deve permitir a inclusão, consulta, alteração e exclusão do estádio onde um jogo acontece.

Tabela 3.4: Exemplo de relação *Feature*-Requisitos – Versão individual

Feature	**Requisitos**
`Jogo`	R1, R3, R5
`Time`	R2, R3
`Estádio`	R4, R5

3.3.5. Modelo de *Feature*s, Documento de Requisitos, Visão de Ação, visão de Ação Aparada e Relação *Feature*s-Requisitos – versão de domínio

O modelo de *feature*s do domínio é um modelo da linha de produtos. Ele exibe todas as *feature*s que os sistemas no domínio podem ou devem conter e é usado para a escolha das *feature*s durante a instanciação dos produtos. Ele deve abranger todas as *feature*s, sub-*feature*s e *feature*s alternativas que foram descobertas na análise de cada produto.

É preciso ter cuidado para que não se repitam *feature*s que podem aparecer com nomes diferentes em cada produto e também cuidado para identificar sub-*feature*s e *feature*s alternativas entre os produtos, conforme explicado adiante. Para o bom entendimento das *feature*s é são utilizadas as versões individuais dos DFs, que podem ser utilizadas para criar uma versão de domínio.

Para construir o modelo de *feature*s do domínio, comparam-se os modelos de *feature*s de cada sistema que foi analisado. Para cada *feature*, deve-se verificar no respectivo documento de *features* a descrição do conceito, para identificar com clareza do que se trata a *feature* e identificar possíveis sinônimos.

Junto com a criação do modelo de *feature*s são feitas a versão de domínio do documento de requisitos e a relação *feature*s-requisito. Ao mesmo tempo em que se

identificam as *feature*s do domínio, são anotados os seus requisitos e o documento de mapeamento mantém registro de a qual *feature* pertence cada requisito.

A inclusão das *feature*s e dos requisitos segue as seguintes regras:

1 *Feature* que aparece em apenas um modelo de *feature*s.

→ A *feature* é incluída como opcional no modelo. Os requisitos são incluídos no documento.

2 *Feature* que aparece em mais de um modelo de featues

(Comparar requisitos)

a. Requisitos compatíveis

→ A *feature* é incluída no modelo como obrigatória. Todos os requisitos são incluídos no documento.

b. Requisitos muito variados

(Decidir por uma das três possibilidades)

→ Criar *feature*s opcionais diferentes, cada uma com seus requisitos específicos.

→ Criar *feature*s alternativas, cada uma com seus requisitos específicos.

→ Criar *feature*s e sub-*feature*, e a sub-*feature* fica com um sub-conjunto dos requisitos.

3 *Feature* é ou possui sub-*feature*s em um modelo e é *feature* em outro

→ Prevalece a *feature* e sub-*feature* como no modelo onde existe a sub-*feature*.

4 *Feature* é ou possui *feature*s alternativas em um modelo e é *feature* em outro

→ Prevalecem as *feature*s alternativas como no modelo onde elas existem.

O processo persiste até que todas as *feature*s de todos os modelos de *feature*s de sistemas tenham sido verificadas. O modelo de *feature*s do domínio é concluído dessa forma.

As *feature*s devem receber uma numeração no modelo de *feature*s. As sub-*feature*s recebem uma sub-numeração de acordo com a *feature* a qual pertencem. Os

requisitos também devem ser numerados. Assim é possível fazer a relação *feature*s-requisitos.

Um requisito deve estar associado a apenas uma *feature*. De acordo com Clarke e Baniassad (2005) (página 99), a ortogonalidade já no nível de requisitos é alcançada não deixando que os grupos de requisitos de diferentes interesses se sobreponham. É comum ocorrer de um requisito citar duas *feature*s. Quando isso ocorrer, deve-se decidir por uma das *feature*s e associar o requisito a ela.

Para criar a versão de domínio da relação *feature*s-requisitos deve-se criar as visões de ação (ver Seção 2.7.3). Elas facilitam a visualização das relações entre as *feature*s e os requisitos, e ajudam na decisão de a qual *feature* pertence um requisito.

Na abordagem proposta neste trabalho, a utilização da visão de ação irá considerar as *feature*s como interesses, o que já era uma premissa desde o início da abordagem, e a visão de ação deve representar esses interesses. Todas as *feature*s que estão presentes no modelo de *feature*s são anotadas na visão de ação. Conforme explicado (ver Seção 2.7.3) sobre a visão de ação, os requisitos devem ser ligados aos temas, no caso, *feature*s.

Tomando como exemplo os requisitos da Tabela 3.3 e as *feature*s `Jogo`, `Time` e `Estadio`, supondo que fossem para o modelo de *feature*s – versão de domínio, a visão de ação obtida ficaria como na Figura 3.3. Pode-se perceber que existe um requisito para o objetivo (ou informações) principal de cada *feature* e outros requisitos que relacionam duas *feature*s. É recomendado escrever desta forma para facilitar a divisão dos requisitos para as *feature*s.

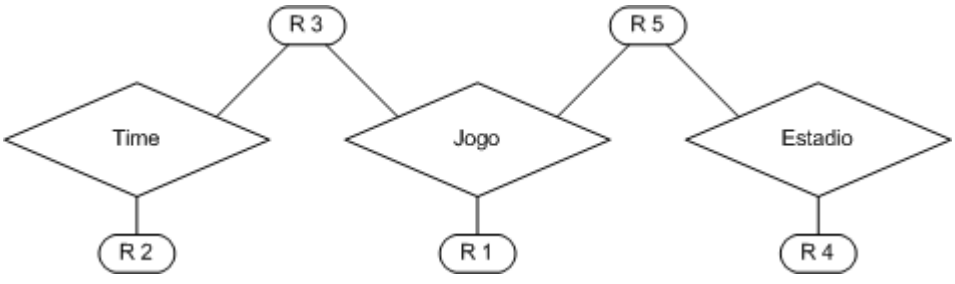


Figura 3.3: Visão de Ação

Tendo-se a visão de ação, deve-se criar a visão de ação aparada, seguindo o processo da abordagem Tema. Todo requisito que está associado a mais de uma *feature* deve ficar associado a apenas uma. Para escolher a qual tema ficará associado o requisito é recomendado verificar qual tema está aumentando ou acrescentando

informações a outro, e nesses casos, o requisito fica associado apenas ao tema que é aumentado.

No exemplo, tem-se que no requisito R3, a *feature* `Time` acrescenta informações e aumenta a *feature* `Jogo`, então na visão de ação aparada criada, vista na Figura 3.4, R3 aparece ligado apenas a `Jogo` e a seta partindo de `Time` para `Jogo` indica que `Time` "aumenta" `Jogo` de alguma maneira. O mesmo se dá entre `Jogo` e `Estadio`. A Tabela 3.4 é um exemplo de como ficaria a relação *feature*s-requisitos neste caso.

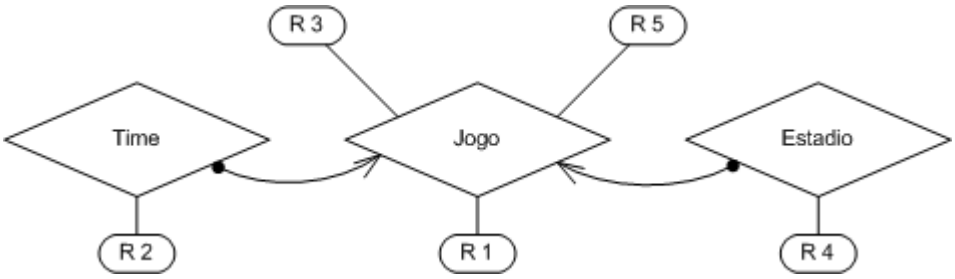


Figura 3.4: Visão de Ação Aparada

Tabela 3.4: Exemplo de relação *Feature*-Requisitos

Feature	**Requisitos**
Jogo	R1, R3, R5
`Time`	R2
Estadio	R4

3.3.6. Considerações Finais da Análise de Domínio

O modelo de *feature*s e o documento de requisitos – versões de sistema – são utilizados para verificar se estão presentes todas as *feature*s e todos os requisitos e reavaliar as ligações entre *feature*s e requisitos para checar se estão consistentes.

Isso conclui a análise de domínio. Já se tem os artefatos básicos necessários para o desenvolvimento do núcleo da LP e das demais *feature*s. Esse desenvolvimento será feito nas fases subseqüentes. Nada impede que, durante essas fases, sejam necessárias mais atividades de análise de domínio para complementar com informações necessárias e os artefatos da análise de domínio produzidos possam ser melhorados.

3.4. Desenvolvimento da Base – F2

O desenvolvimento da base da linha de produtos consiste no projeto e implementação das *feature*s obrigatórias. A partir do modelo de *feature*s (produzido na fase anterior) verifica-se quais são as *feature*s obrigatórias. Com a numeração presente no modelo de *feature*s, utiliza-se a relação *feature*s-requisitos para obter quais são os requisitos das *feature*s obrigatórias. A partir dos requisitos, as *feature*s são projetadas e implementadas de acordo com as diretrizes descritas nas Seções 3.5.1 e 3.5.2.

O desenvolvimento das *feature*s deve ser iniciado pelas *feature*s que não influenciam outras *feature*s, o que pode ser percebido pela visão de ação aparada desenvolvida na fase de análise. O desenvolvimento das *feature*s que influenciam outras *feature*s é facilitado se já houver o projeto e mesmo a implementação da *feature* que sofrerá influência.

Dependendo do número de *feature*s obrigatórias existentes, o desenvolvimento pode ser alocado em diversos ciclos, usando modelos como o PU, por exemplo. O início do desenvolvimento, que pode ser antes ou durante o desenvolvimento da primeira *feature*, necessita de várias atividades complementares:

1. Criação da interface principal com o usuário do sistema, da qual as outras funcionalidades presentes nas *feature*s serão chamadas;
2. Projeto dos detalhes da arquitetura do software, se necessário. Neste trabalho considerou-se que a arquitetura será baseada no modelo de três camadas;
3. Projeto do esquema da base de dados que persistirá os objetos;

Implementação da camada de persistência oferecendo uma interface (métodos) para a manipulação dos dados no BD, por exemplo, caso seja utilizado um banco de dados relacional.

3.5. Desenvolvimento de Features – F 3.1

O desenvolvimento das *feature*s é uma atividade da abordagem comum às fases de Desenvolvimento da Base (F2) e de Desenvolvimento de Produtos (F3). Por esse motivo, o método para desenvolvimento de *feature*s deve ser conhecido antes mesmo

do método para desenvolvimento da base, já que a base também é composta de *features*. O método para desenvolvimento de *feature*s é a parte mais complexa e extensa do presente trabalho. Ele é dividido em projeto e implementação.

3.5.1 Projeto

O projeto de uma *feature* origina vários artefatos que servem de base e possibilitam sua implementação. O processo para criação do projeto utiliza uma mescla de técnicas e artefatos do PU e técnicas e artefatos da abordagem Tema/Doc (Clarke e Baniassad, 2005).

Deseja-se projetar uma *feature* isolando apenas o que é pertinente a ela e nada mais. A abordagem Tema/Doc é empregada para isso, pois possibilita a separação de interesses no nível de projeto.

O projeto de uma *feature* é composto de três etapas, que envolvem análise e projeto. Cada uma produz uma parte do projeto da *feature*. A primeira etapa cria a parte do projeto para tratar do interesse da *feature* o mais estrita e exclusivamente possível, livre influências de outras *feature*s. A segunda etapa cria a parte do projeto da *feature* que existe por influência da presença de outras *feature*s. A terceira etapa cria a parte do projeto da *feature* que existe por influência da *feature* nova no resto do sistema.

Apesar da intenção de isolar as *feature*s, em determinados momentos elas influenciam umas nas outras, pois o comportamento esperado do sistema pronto contém a interação entre *feature*s conforme foi visto no exemplo da Seção 3.3.4. Com o projeto em três etapas consegue-se a capacidade para combinar as *feature*s e reutilizá-las para outros produtos da linha, minimizando a quantidade de retrabalho. Na Figura 3.5 estão resumidas as atividades e artefatos do Projeto de *Feature*s.

Etapa 1

Nesta etapa cria-se a parte do projeto para tratar do interesse da feature o mais estrita e exclusivamente possível, livre influências de outras features. Os requisitos pertinentes à *feature* são obtidos utilizando-se a relação *feature*-requisitos. Para a compreensão desses requisitos, o documento de *features* pode ser consultado para sanar

dúvidas quanto aos conceitos envolvidos na *feature* e garantir que os objetos criados na *feature* tenham uma nomeação consistente.

A seguir, os requisitos devem ser analisados para verificar se eles mencionam outras *feature*s. É comum que requisitos envolvam mais de uma *feature*. Durante a criação da relação *feature*-requisitos (Seção 3.3.4), cada requisito foi associado a uma *feature*. Neste momento pode-se rever essa decisão analisando novamente os requisitos de uma *feature*.

Os artefatos da abordagem Tema, Visão de Ação e Visão de Ação Aparada, podem ser utilizados neste momento para uma melhor visualização da situação e auxiliar na decisão. Neste momento, recomenda-se rever as decisões quanto às associações de requisitos com *feature*s, presentes na visão de ação e visão de ação aparada, pois agora os requisitos foram reexaminados e tem-se um melhor embasamento. Novamente, o documento de *features* e o modelo de *feature*s ajudam a identificar conceitos e objetos mencionados nos requisitos e identificar a qual *feature* eles pertencem.

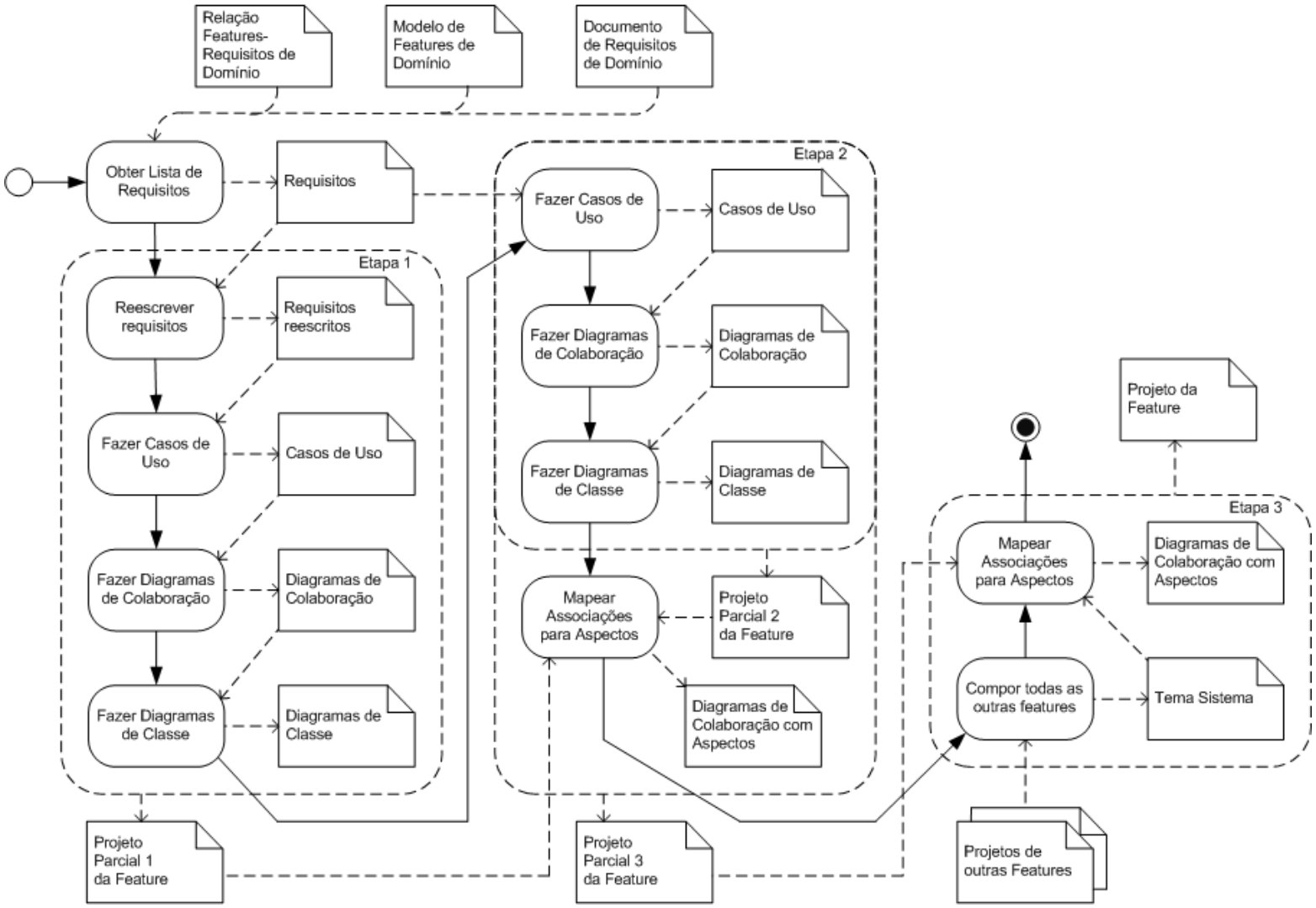


Figura 3.5: Atividades e Artefatos do Desenvolvimento de *Feature*s

Quando um requisito menciona mais de uma *feature*, isso significa entrelaçamento de interesses, que no caso dos objetivos de criação de linhas de produtos deste trabalho, significa que uma *feature* influi em outra. Nesses casos, o seguinte procedimento é adotado:

> Os requisitos são reescritos removendo-se deles o que é referente a outras *feature*s. Pode ocorrer que o requisito, a funcionalidade envolvida ou mesmo a própria *feature* pareça não fazer mais sentido semanticamente com os requisitos reescritos, mas mesmo assim, os requisitos devem ser reescritos livres de referências a outras *feature*s.

Com os requisitos reescritos, cria-se a primeira parte do projeto da *feature*. Esse projeto é feito com casos de uso, diagramas de colaboração e diagrama de classes.

Etapa 2

Nesta etapa cria-se a parte do projeto da feature que existe por influência da presença de outras features. Com base nos requisitos originais, onde havia entrelaçamento de interesses, cria-se outro projeto para a mesma *feature*. Esse projeto é feito com casos de uso, diagramas de colaboração e diagrama de classes. Ele será similar ao primeiro, porém maior, com mais elementos. Também é possível que os requisitos da *feature* não mencionem outras *feature*s, e nesses casos, esta etapa é desconsiderada.

A diferença entre os projetos da etapa 1 e 2 são as partes que as outras *feature*s acrescentam ou influenciam na *feature* que está sendo projetada. Uma vez que se sabe quais elementos são acrescentados por outras *feature*s e sabendo o modo como alguns comportamentos serão alterados, pode-se então projetar esta parte isoladamente, o que também irá permitir uma implementação modularizada.

As influências entre *feature*s serão implementadas por aspectos, portanto, é utilizado um diagrama de colaboração modificado, como artefato auxiliar, para denotar o envolvimento dos aspectos. São feitos diagramas para os métodos que tenham seu comportamento modificado. Uma pequena parte da notação do diagrama é proposta neste trabalho como sugestão, mas o desenvolvedor é livre para utilizar as extensões da

UML para trabalhar com aspectos. A notação dos diagramas é explicada a seguir, já que sua criação é a próxima atividade da Etapa 2.

Diagrama de colaboração com aspectos

Para o método que terá seu comportamento modificado, utiliza-se como base o diagrama de colaboração criado para ele na Etapa 1. Copia-se então o diagrama de Etapa 1. O diagrama de colaboração criado na Etapa 2 é consultado para indicar onde estão os métodos e atributos a mais.

Para denotar a interceptação de um aspecto é utilizada uma seta tracejada. Cada interceptação tem um título (colocado junto à seta) que começa com a letra "A" maiúscula seguida de um número para nomear as diferentes interceptações. O título "`A0`" é reservado para indicar declarações intertipos, indicando atributos e métodos que o aspecto acrescenta por necessidade. Os atributos e valores acrescentados são anotados com um "`A0:`" seguido dos nomes dos atributos e métodos acrescentados, podendo haver vários na mesma seta. Para essa primeira interceptação `A0`, a seta vai do aspecto para a classe, enquanto nas outras interceptações a seta vai do aspecto para um método. Isso representa o que ocorre na prática, as declarações intertipo são para a classe toda, enquanto as interceptações se dão em métodos. As outras setas `A1`, `A2` e assim por diante, denotam interceptações a métodos, indicam explicitamente o tipo da interceptação, *before*, *after* ou *around* e *execution* ou *call*.

Uma interceptação a um método geralmente tem um adendo que executa algum comportamento. Então, o fluxo de execução do adendo para essa interceptação é exibido com a sub-numeração começando com `An`, onde n é o número dado à interceptação, conforme exemplificado na Figura 3.6.

Pode-se ver o fluxo original do diagrama com a numeração sem nenhuma letra A maiúscula. Além disso, existe a seta tracejada com três declarações intertipos indicadas por `A0`, e uma interceptação ao método principal do diagrama. Essa interceptação, do tipo *before execution*, recebeu a numeração `A1`. A partir de então, o fluxo de execução do adendo é mostrado pelas chamadas com título iniciado com `A1`. Na Figura 3.6 pode ser visto que o adendo chama o método `metodo4()` da classe `ClasseC` e depois o método `método_z()` da classe `ClasseA`.

As interceptações do tipo *around* podem ter as mensagens `proceed()` e `return`. Essas mensagens são anotadas junto à seta que iniciou a interceptação, da mesma maneira que as demais mensagens do fluxo de execução, com o respectivo `An`, para indicar à qual interceptação se referem, seguidas da sub-numeração para indicar a ordem de ocorrência. A chamada a `return` deve mostrar qual o valor do retorno, exceto quando a função interceptada também não tinha retorno. Na Figura 3.7 pode-se ver um exemplo de interceptação do tipo *around*.

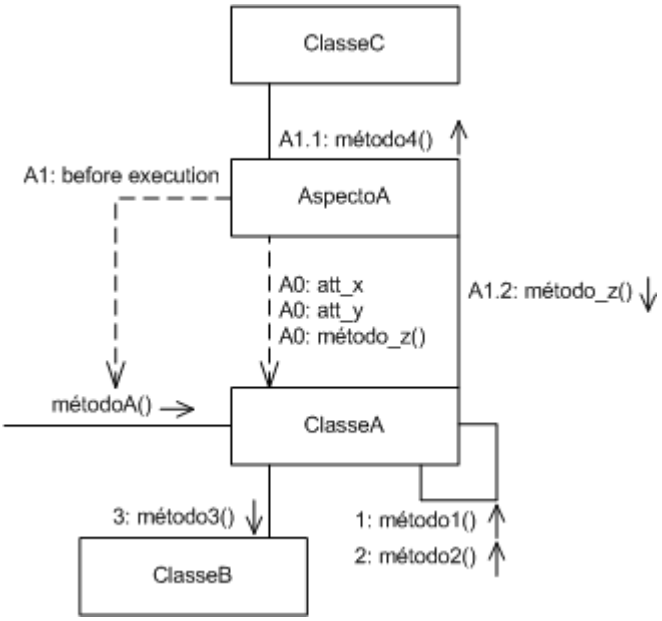


Figura 3.6: Diagrama de colaboração com aspectos

O adendo para `A1` inicia chamando o método `método5()` da classe `ClasseE` e, em seguida, devolve o fluxo de execução para o método original no passo `A1.2`. Nesse passo ainda existe uma condição, indicada do mesmo modo como é feito com o diagrama de colaboração UML padrão. A existência do passo `A1.3` indica que, após a execução do método, o aspecto retoma o fluxo de execução e executa o método `método6()` da classe `ClasseE`, para finalmente retornar um valor no passo `A1.4`.

Também existem os casos em que as alterações causadas pelo envolvimento de outra *feature* não é tão pontual e precisa que mais de um método seja interceptado para contribuir para atingir o comportamento desejado. Por isso, as setas podem ter os títulos `A1`, `A2`, e assim por diante. Essa primeira parte da numeração não indica que a interceptação `A1`, por exemplo, ocorra antes da `A2`. A ordem da interceptação é determinada pelo momento em que o fluxo de execução passar pelo método que é

interceptado, correspondendo ao que ocorre na prática. Um exemplo pode ser visto na Figura 3.8.

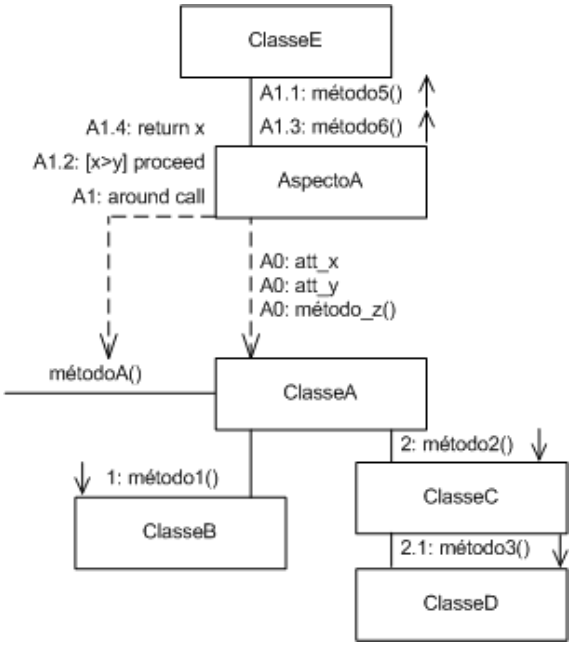


Figura 3.7: Diagrama de colaboração com aspectos com interceptação do tipo *around*

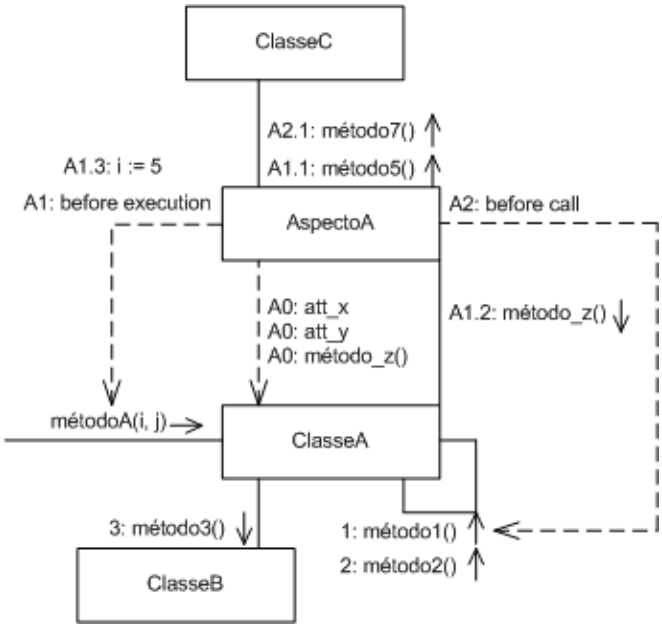


Figura 3.8: Diagrama de colaboração com aspectos com mais de uma interceptação

A interceptação `A1` ocorre antes da execução do método principal. Quando a interceptação termina, o método prossegue, e no passo 1, o método `método1()` da classe `ClasseA` é chamado. Essa chamada sofre a interceptação `A2`.

Outro detalhe que pode precisar ser representado no diagrama são os casos onde um aspecto muda o valor de um parâmetro de uma função. Isso também é anotado na seta da interceptação, pois o argumento pertence à chamada ao método, e o momento que o valor do parâmetro é alterado é indicado com o mesmo esquema de numeração. Na Figura 3.8 o passo `A1.3` é a alteração do valor do parâmetro `i` da função `metodoA(i, j)` da classe `ClasseA`.

Isso conclui a descrição dos diagrama de colaboração com aspectos. Para encerrar a Etapa 2 do projeto são feitos outros artefatos caso sejam necessários, tais como diagramas de colaboração sem aspectos para métodos que não envolvam aspectos.

Etapa 3

Nesta etapa cria-se a parte do projeto da *feature* que existe por influência da *feature* nova no resto do sistema. Para descobrir qual é essa influência, é necessário ter o projeto da *feature* e o projeto do restante do sistema. O projeto do restante do sistema é obtido com o processo de composição de temas-base. As demais *feature*s presentes no produto são compostas para obter o projeto do restante do sistema, chamado de tema "Sistema".

Conforme definido na abordagem Tema, isso é feito para poder analisar o tema da *feature* atual com o tema Sistema e verificar como será a composição dos dois. A análise da composição indica quais partes da nova *feature* devem ser projetadas como classes, quais partes devem ser projetadas como intromissões[4] em outras *feature*s e onde essas intromissões ocorrem.

O projeto da nova *feature* deve ser analisado em relação ao projeto contido no tema Sistema. Cinco fatores devem ser analisados do ponto de vista do tema Sistema. São eles:

1. Novas classes.
2. Novos atributos e métodos em classes que já existiam.

[4] Intromissões e influências não se referem a termos técnicos do glossário de POA, mas são termos para mencionar em um nível mais alto o que uma *feature* pode causar em outra.

3. Mudanças no comportamento de métodos existentes. Alguns métodos que já existiam podem apresentar um comportamento diferente no projeto da nova *feature* (modelado por um diagrama de colaboração diferente).
4. Associações das classes da *feature* com classes de outras *feature*s.
5. Associações entre classes de *features* diferentes existentes anteriormente podem precisar ser desfeitas.

Para esses cinco fatores, as respectivas cinco ações serão tomadas:

1. As novas classes são implementadas como classes comuns e sem aspectos.
2. Os novos atributos e métodos para classes já existentes são introduzidos nas classes com declarações intertipos.
3. As mudanças no comportamento dos métodos são feitas com interceptações e adendos aos métodos.
4. As novas associações são implementadas com aspectos, para garantir a conectividade da *feature* e permitir que quando necessário, elas possam ser desfeitas.
5. Associações são desfeitas não incluindo-se os aspectos que fazem a associação na montagem do produto.

Um exemplo de desenvolvimento e implementação de uma nova *feature* é ilustrado pela Figura 3.9. Para um dado sistema, uma nova *feature* `FeatureB` é projetada. O projeto da *feature* `FeatureB` é representado pelo tema `FeatureB`. O projeto do resto do sistema é representado pelo tema `FeatureA`.

Os números na figura representam ocorrências dos fatores mencionados acima. O número 1 indica um objeto de lógica de aplicação próprio da *feature*, uma nova classe. O número 2 indica um novo método necessário em uma classe que é pertinente a uma outra *feature*. Os números 3 indicam um método que tem o seu comportamento modificado com a presença da nova *feature*. Os números 4 indicam associações entre classes, sendo que uma das classes da associação é pertinente à outra *feature* Associações podem ser percebidas pelas inclusões de referências ao objeto em uma das classes. O número 5 indica uma associação entre classes que deve ser desfeita por cause de um novo comportamento do sistema.

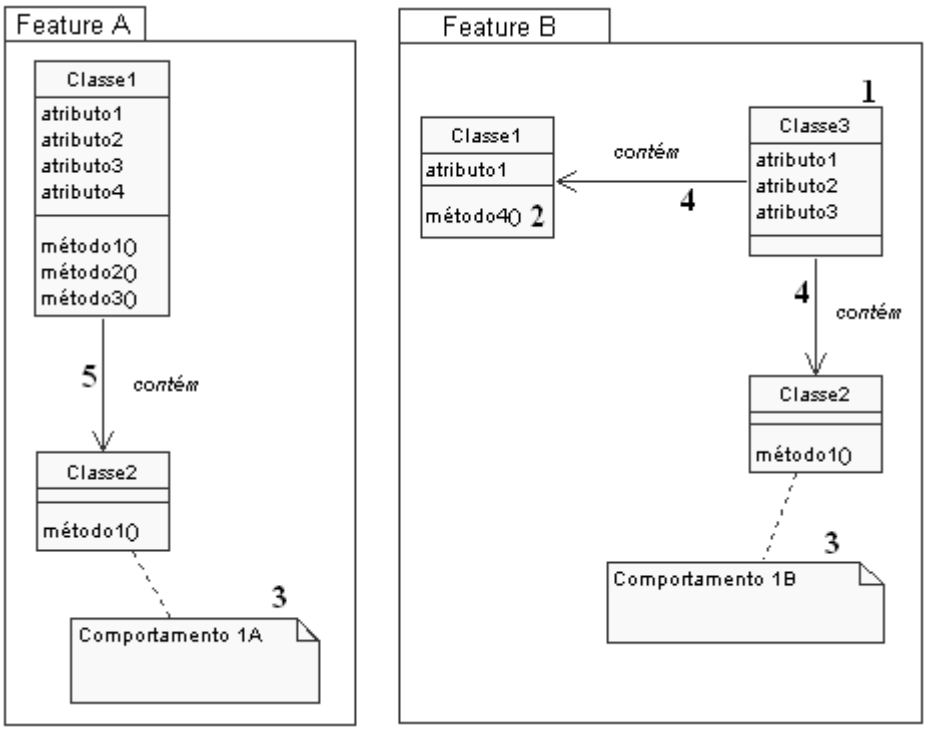


Figura 3.9: Adição de uma *feature*

Para essas medidas a serem tomadas, são criados basicamente dois tipos de interceptações e adendos: os que mudam o comportamento dos métodos conforme necessário pela nova *feature* e os que mudam o comportamento dos métodos apor causa da inserção de mais atributos na classe. Esses últimos são alterações em todos os métodos da classe que manipulavam seus atributos.

Os conjuntos de junções e adendos devem ser projetados com diagramas de colaboração com aspectos, os quais mostram como, quando e onde o aspecto altera o comportamento do método. Com isso, conclui-se o projeto da *feature* e parte-se para sua implementação.

3.5.2 Implementação

A implementação da linha de produtos é feita com a implementação de cada *feature*. Esta seção especifica diretrizes para a implementação das *feature*s, visando garantir que a linha de produtos seja adaptável o suficiente para permitir que as *feature*s sejam combinadas de maneira fácil e que novas *feature*s possam continuar a ser adicionadas.

As soluções propostas para essa questão foram encontradas e utilizadas durante o estudo de caso desenvolvido. Elas são aqui descritas textualmente especificando o problema a ser resolvido, a solução proposta, o contexto da solução e suas implicações. As soluções também são descritas tecnicamente com exemplos.

A implementação feita neste trabalho utilizou-se das linguagens Java e AspectJ, mas as soluções não se prendem a essas linguagens. Elas são adaptáveis para outras linguagens orientadas a objetos que possuam uma extensão para orientação a aspectos que implemente os conceitos de declarações intertipos, conjuntos de junção e adendos. Isso deixa uma ampla gama de possibilidades.

Optou-se pela arquitetura de três camadas (*Three Tier Software Architecture*) (SEI, 2006) para padronizar a criação das *feature*s. Essa escolha se deve ao fato da separação nas três camadas permitir que se definam padrões de implementação específicos para cada camada.

Independentemente da arquitetura escolhida, a adição de *feature*s provoca alterações nas classes de lógica da aplicação. A interface com o usuário e a persistência devem acompanhar essas mudanças. Com essas três partes bem delineadas pelo uso da arquitetura de três camadas, é mais fácil acompanhar e tratar as alterações resultantes nas outras partes causadas por alterações na lógica de aplicação.

3.5.2.1 Camada de Aplicação

Os princípios básicos que irão guiar a implementação da *feature* são os cinco fatores mencionados na Seção 3.5.1. O primeiro princípio fala a respeito da implementação de novas classes. A partir do segundo princípio percebe-se a necessidade de mudanças em classes existentes. Antecipando-se a essas mudanças, já a partir da criação de novas classes, a estrutura da classe é feita de forma a facilitá-las.

Para cada *feature* e para cada classe existente que ganhe novos atributos e ou métodos, será criado um aspecto. Esse aspecto deverá:

- Introduzir os novos atributos;
- Introduzir os novos métodos;
- Fornecer o valor inicial dos novos atributos;

- Cuidar para que os métodos da classe que manipulam seus atributos também manipulem os novos atributos;
- Tratar possíveis regras de negócios associados a esses novos atributos.

Durante o desenvolvimento do estudo de caso, percebeu-se que grande parte do trabalho de implementação dos aspectos era cuidar dos métodos que manipulavam os atributos da classe, para fazer com que eles passassem a incluir os atributos acrescentados pela *feature*.

A solução adotada para evitar grande parte deste trabalho foi a utilização de meta-atributos. Colocou-se todos os valores dos atributos em um vetor. Isso possibilitou que bastasse que os aspectos colocassem seus novos atributos no vetor, e não era mais necessário fazer um adendo para cada função incluir os novos atributos.

Funções que recebiam todos os atributos do objeto ou que retornavam todos esses atributos foram modificadas para receber e retornar objetos do tipo vetor. Um caso particularmente comum de funções desse tipo são as funções para consultas à base de dados. Por outro lado, perdem-se as vantagens de checagem estática das variáveis (em tempo de compilação) (Dijkstra, 1976). Outras possíveis soluções que facilitem o acréscimo de novos atributos seriam usar a reflexividade da linguagem Java ou modelos de objetos ativos (Yoder et al., 2001).

Assim, na implementação sugerida neste trabalho, as classes da camada de aplicação passaram a ter apenas dois atributos, um vetor com todos os valores dos atributos, que foi chamado de "`valores`" e um outro vetor com os nomes de todos os atributos, que foi chamado de "`campos`". A linguagem Java contém um tipo `Vector` que pode armazenar objetos de tipos diferentes e ele foi utilizado neste trabalho para implementar os meta-atributos.

Com esta solução adotada, o construtor das classes (exemplo na Figura 3.10) passou a declarar os dois vetores e inicializar o vetor `campos` com os nomes dos atributos e o vetor `valores` com os valores default. É importante que a posição dos nomes no vetor `campos` esteja de acordo com a posição dos valores no vetor `valores`. Isso é feito no construtor inserindo os elementos na ordem correta.

```
Public Jogo(){
  this.campos = new Vector();
  this.valores = new Vector();

  this.campos.add("id_jogo");       //00
  this.campos.add("data");          //01
  this.campos.add("hora");          //02
  this.campos.add("pontosA");       //03
  this.campos.add("pontosB");       //04

  this.valores.add("0");            //00
  this.valores.add(new date());     //01
  this.valores.add(new time());     //02
  this.valores.add("0");            //03
  this.valores.add("0");            //04
}
```

Figura 3.10: Exemplo de construtor

A camada de persistência deve ser preparada para oferecer uma interface de serviços que possa lidar com esses vetores. Os métodos desses serviços irão trabalhar com parâmetros do tipo dos meta-atributos, sendo capazes de obter os dados dos atributos contidos nos meta-atributos. O seguinte método é um exemplo utilizado para a inserção de um objeto:

```
public boolean insercao(String nomeTabela, Vector colunas,
                Vector valoresColunas)
```

O método recebe uma *String* com o nome da tabela no banco de dados que corresponde ao objeto sendo inserido, um vetor que deve conter os nomes das colunas na tabela e um vetor que deve conter os valores para a inserção nas colunas. Assim, no projeto da base de dados, os nomes das colunas das tabelas precisam ter os nomes iguais aos nomes que são colocados no vetor `campos`.

No exemplo a seguir, uma *feature* `Estadio` é adicionada à *feature* `Jogo`. O projeto da *feature* `Jogo` é mostrado na Figura 3.11a e o projeto da *feature* `Estadio` é mostrado na Figura 3.11b.

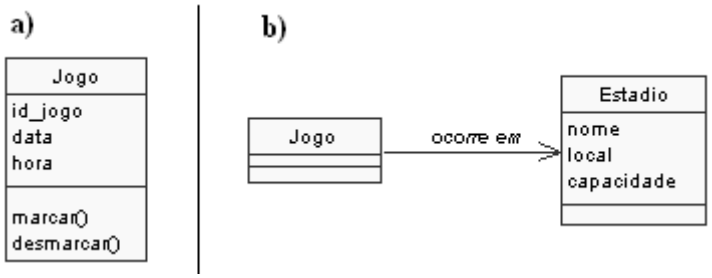


Figura 3.11: Projeto das *features* `Jogo` e `Estadio`

Por exemplo, para tratar das introduções da *feature* `Estadio` na classe `Jogo` é utilizado um aspecto denominado `Jogo_Estadio`, que representa a associação entre as classes `Jogo` e `Estadio`.

Conforme discutido anteriormente, uma das incumbências de um aspecto pode ser de incluir novos atributos na classe. Associações entre classes são um caso particular, onde uma referência a uma das classes é incluída como um atributo da outra. A introdução da associação do exemplo acima poderia ser feita por declarações intertipos, mas como foram usados meta-atributos ela é feita por um adendo que age depois da execução do construtor da classe. Esse adendo introduz a string "estadio" no vetor `campos` da classe `Jogo` e um objeto do tipo `Estadio` no vetor `valores`. O construtor da classe `Jogo` foi mostrado na Figura 3.10. Na Figura 3.12 é mostrado o conjunto de junção e o adendo que introduz um objeto tipo `Estadio` no objeto `Jogo`. Nesse adendo também já é feita a inicialização dos valores *default* para os atributos.

```
pointcut construtor(Jogo j1):
   execution (Jogo.new(..))
   && this(j1);
after (Jogo j1): construtor(j1){
   j1.campos.add("Estadio");
   j1.valores.add(new Estadio());
}
```

Figura 3.12: Conjunto de junção e Adendo que incluem e inicializam novos atributos na classe `Jogo`

Os novos métodos necessários para que a classe `Jogo` possa cuidar do interesse `Estadio` são incluídos com declarações intertipos. Uma parte da implementação relativamente difícil são os detalhes referentes à manipulação dos novos atributos pelas funções de `Jogo`. Existem funções que precisam manipular todos os atributos do objeto, como por exemplo, as funções de inserção no banco de dados, em que muitas vezes é necessário validar ou tratar os atributos. Para cada uma dessas necessidades é preciso

implementar um conjunto de junção que capte o momento certo para a manipulação dos dados e um adendo que faça a manipulação, e nesses casos, se torna mais difícil conseguir algum tipo de padronização, já que a ação a ser feita depende da classe, dos atributos e outros fatores.

Os três tipos de interceptações fornecidos pela linguagem AspectJ, *before*, *after* e *around* são suficientes para o tratamento dos novos atributos nesses casos. O mais adequado é o *around*, pois ele pode fazer os tratamentos ou validações necessários antes da função cumprir seu objetivo e, dependendo do resultado, deixar que a função retome o fluxo de execução com a cláusula `proceed()`, ou dar um retorno no lugar da função sinalizando o resultado. Nos casos em que a função continuar o processamento, o *around* ainda retoma o controle depois que a função retorna, e pode fazer para os novos atributos o objetivo da função e, ainda, manipular o retorno da função.

3.5.2.2 Camada da GUI

No contexto da inserção de novas *feature*s a um conjunto de *feature*s já implementadas, a camada de interface precisa acompanhar as mudanças que ocorrem nas classes de lógica de aplicação que elas representam.

No desenvolvimento deste trabalho, foi decidido que a maioria dos objetos de lógica de aplicação teria duas classes de interface, uma para a busca dos objetos (nomeadas `FrameBuscar<nome do objeto>`) e outra para fazer consulta, inserção, atualização e exclusão do objeto (nomeada `FrameIncluir<nome do objeto>`). Alguns objetos podem precisar de interfaces associadas com as demais funcionalidades. Se a classe de lógica de aplicação ganhar novos atributos, a classe de interface precisa ganhar novos *widgets*. Similarmente, as introduções de *widgets* relativos a atributos que são inerentes a outras *feature*s devem ser feitos por aspectos, para que também possam ser removidos quando necessário.

Na camada da GUI, os novos atributos são adicionados por aspectos com declarações intertipos, pois os atributos das classes desta camada não são transformados em meta-atributos. Isso não é necessário, já que a camada de interface não tem tanta necessidade de acessar métodos que passam todos os atributos de uma classe como parâmetros, como as classes da camada de aplicação. O mais comum é utilizarem métodos específicos que manipulem alguns atributos.

A implementação da interface é estruturada para dar suporte aos novos atributos. Faz-se com que o código da interface tenha pontos de junção em locais específicos onde os *widgets* devem atuar. Dado o tipo de sistema e a linguagem de programação a que se direciona este trabalho, foram identificadas cinco atividades realizadas pelo código para tratar as variáveis de interface:

1. A declaração das variáveis
2. A inicialização das variáveis
3. A preparação dos valores das variáveis
4. A preparação do leiaute
5. A finalização do *frame* da interface.

Com exceção da declaração de variáveis, essas atividades serão implementadas por métodos separados. Assim, quando a interface ganhar novos *widgets*, ou novas variáveis, esses novos *widgets* também precisam sofrer essas atividades. Então, são feitas interceptações com aspectos em cada método e essas interceptações realizam as atividades para os novos *widgets*. A primeira atividade, a declaração de variáveis, é feita por declarações intertipos.

O leiaute da interface refere-se ao posicionamento e ordem dos objetos. A preparação do leiaute é muito específica para cada tipo de sistema e linguagem de programação escolhida. Mas a idéia aplicada neste trabalho pode ser aplicada a muitos casos. A preparação de interface é dividida muitas vezes até a granularidade de cada elemento, em métodos que fazem a interface parte por parte. Assim, ficam disponíveis os pontos de junção para serem interceptados e permitir que o leiaute da interface ganhe novos elementos.

Na Figura 3.13 há um exemplo de implementação de leiaute para um caso específico em que é necessário incluir objetos no leiaute tanto na horizontal quanto na vertical. Primeiro o método de preparação do leiaute se divide em dois métodos, um para preparar o leiaute horizontal e outro para preparar o leiaute vertical (Figura 3.13a). Os métodos específicos de preparação horizontal e vertical (Figura 3.13b e 3.13c respectivamente) usam funções específicas para adicionar *widgets* hipotéticos X e Y (Figura 3.13d).

Para que o aspecto possa colocar um ou mais *widgets* antes ou depois de algum elemento, basta interceptar antes ou depois das chamadas aos métodos que inserem

elementos específicos (por exemplo `adicionaComponentesXHorizontalJPanel1()` na Figura 3.13b). No caso de dois aspectos precisarem colocar componentes depois de um mesmo método, pode ser necessário declarar a precedência.

Na Figura 3.14 tem-se um exemplo dos conjuntos de junção e adendos necessários para incluir *widgets* de `Estadio` em `Jogo`. O primeiro conjunto de junção e adendo, na Figura 3.14a, intercepta a função que adiciona *widgets* referentes à hora do jogo no leiaute horizontal. Na Figura 3.14b pode-se ver os respectivos *widgets* para o leiaute vertical.

a)
```
private void preparaLeiauteJPanel1() {
   preparaLeiauteHorizontalJPanel1();
   preparaLeiauteVerticalJPanel1();
}
```

b)
```
private void preparaLeiauteHorizontalJPanel1() {
   adicionaComponentesXHorizontalJPanel1()
   adicionaComponentesZHorizontalJPanel1()
}
```

c)
```
private void preparaLeiauteVerticalJPanel1() {
   adicionaComponentesXVerticalJPanel1()
   adicionaComponentesZVerticalJPanel1()
}
```

d)
```
private void adicionaComponentesXHorizontalJPanel1() {
   //adiciona os widgets de X no leiaute horizontal
}
private void adicionaComponentesZHorizontalJPanel1() {
   //adiciona os widgets de Z no leiaute horizontal
}
private void adicionaComponentesXVerticalJPanel1() {
   //adiciona os widgets de X no leiaute vertical
}
private void adicionaComponentesZVerticalJPanel1() {
   //adiciona os widgets de Z no leiaute vertical
}
```

Figura 3.13: Esquema de preparação dos leiautes em uma classe de interface

Na arquitetura utilizada neste trabalho, as classes da camada de interface chamam métodos de classes da camada de aplicação, passando valores que foram definidos durante a interação com o usuário do sistema. Assim, primeiramente é a interface que possui esses valores, inclusive para os *widgets* que foram introduzidos pelos aspectos. Então, algumas vezes ocorre a necessidade de que seja o aspecto responsável pela interface que faça a interceptação a uma chamada a métodos de classes da camada de

aplicação. O aspecto de interface precisa ter privilégio para visibilidade, mesmo a certas introduções de outros aspectos. Ele deve ter acesso aos métodos *sets* e *gets* que dão acesso ao atributo introduzido na classe de lógica de aplicação, como se eles fossem originais da classe. A linguagem AspectJ fornece a palavra reservada `privileged` que garante esse privilégio ao aspecto.

No aspecto da interface, então, cria-se um conjunto de junção para cada chamada a métodos da lógica de aplicação que também precisariam manipular o novo atributo e é feito um adendo, para cada conjunto de junção, para fazer a manipulação do atributo.

a)

```
pointcut preparaLeiauteHorizontalJPanel1(FrameJogo fj,
GroupLeiaute.ParallelGroup pg, GroupLeiaute lo):
   call ( *
   FrameJogo.add_hora_leiauteHorizontalJPanel1(..))
   && target(fj)
   && args(pg, lo);
after(FrameJogo fj, GroupLeiaute.ParallelGroup pg,
   GroupLeiaute lo): preparaLeiauteHorizontalJPanel1(fj, pg, lo)
{
   //adiciona os widgets de Estádio com métodos específicos da
   //linguagem.
}
```

b)

```
pointcut preparaLeiauteVerticalJPanel1(
FrameJogo fj, GroupLeiaute.SequentialGroup sg, GroupLeiaute lo):
   call ( *
   FrameJogo.add_hora_leiauteVerticalJPanel1(..))
   && target(fj)
   && args(sg, lo);
after(FrameJogo fj, GroupLeiaute.SequentialGroup sg,
   GroupLeiaute lo): preparaLeiauteVerticalJPanel1(fj, sg, lo)
{
   //adiciona os widgets de Estádio com métodos específicos da
   //linguagem.
}
```

Figura 3.14: Aspecto FrameJogo_Estadio

Na Figura 3.15 é exibido um exemplo. A interface `FrameJogo` instancia um objeto `Jogo` e chama o método `setJogo()` para passar os valores fornecidos pelo usuário para o objeto. O conjunto de junção `FrameJogoSetJogo` e um adendo do tipo *after* cuidam de passar os valores dos novos *widgets* da interface para os novos atributos da classe de aplicação.

```
pointcut FrameJogoSetJogo (FrameJogo fj, Jogo j):
   call (* Jogo.setJogo (..))
   && this(fj)
   && target(j);

after(FrameJogo fj, Jogo j): FrameJogoSetJogo (fj, j)
{
   int idE = 0; //número identificador para o Estadio
   if (fj.jListEstadio.getSelectedIndex() >= 0)
   idE = Integer.parseInt((String) fa.vetorEstadio.get(
         fj.jListEstadio.getSelectedIndex())) ;
   j.setEstadio(idE);
}
```

Figura 3.15: Aspecto de interface interceptando uma classe de lógica de aplicação

O conjunto de junção precisa especificar que são válidas somente as chamadas ao método feitas pela classe de interface correspondente. A cláusula `this()` com parâmetro do tipo da classe de interface garante isso.

No exemplo da Figura 3.15, o conjunto de junção `FrameJogoSetJogo` especifica que somente as chamadas ao método `setEstadio()` feitas pela classe `FrameJogo` são interceptadas. A classe `FrameJogo` pertence à camada de interface e o método `setEstadio()` pertence à classe `Jogo`, que pertence à camada de aplicação. Os objetos `vetorEstadio` e `jListEstadio` haviam sido introduzidos na classe `FrameJogo` pelo mesmo aspecto que contém o conjunto de junção e adendo do exemplo.

3.5.2.3 Artefatos de Código

Os artefatos de código (classes e aspectos) pertinentes a uma *feature* são reunidos em um ou mais pacotes. Sugere-se colocar em um pacote separado as novas classes propriamente ditas e criar um pacote para cada nova associação entre *feature*s diferentes. Dessa forma, fica mais fácil reusar as novas classes ou somente as associações.

O pacote com as classes novas próprias da *feature* é nomeado com o nome da *feature*. Os demais pacotes, se houver, contém as associações de classes da *feature* com classes de outras *feature*s. Sugere-se que esses pacotes tenham um nome dividido em três partes. A primeira parte é o nome da *feature* que tem as classes alteradas pelos aspectos, a segunda parte é o nome da *feature* que tem as classes que fornecem atributos ou métodos utilizados colocados na *feature* que foi aumentada e a terceira parte é nome

da nova *feature* que foi desenvolvida e ocasionou essa associação. Duas partes no nome apenas não bastam, pois uma *feature* pode precisar criar uma associação entre classes de duas *feature*s e posteriormente uma outra *feature* poderia precisar criar uma associação diferente entre classes das mesmas *feature*s.

Por sua vez, os aspectos dentro dos pacotes têm um nome formado por duas partes. A primeira parte é o nome da classe que está sendo modificada pelo aspecto e a segunda parte é o nome da *feature* que contém as classes que fornecem atributos ou métodos para as classes alteradas. Neste trabalho foi utilizado o caractere sublinhado "_" para separação de partes de nomes. O esquema de criação dos nomes de pacotes é ilustrado no exemplo da Figura 3.16.

Nome: Parte1_Parte2_Parte3

Parte1: *Feature* aumentada com classes, atributos ou métodos de outras *feature*s.

Parte2: *Feature* que fornece classes, atributos ou métodos incluídos na primeira *feature*

Parte3: *Feature* que causou a necessidade de alterações na primeira *feature* devido a sua lógica.

Inicialmente tem-se uma *feature* `FeatureA`, cujo projeto contém apenas a classe `ClasseA` (Figura 3.16a). A implementação da *feature* `FeatureA` gera um pacote nomeado `FeatureA`, que contém a classe `ClasseA`.

Posteriormente cria-se uma *feature* `FeatureB`, cujo projeto contém as classes `ClasseA` e `ClasseB` e uma associação `AssociacaoABB` entre as duas (Figura 3. 16b). A implementação da *feature* `FeatureB` gera um pacote nomeado `FeatureB` que contém a classe `ClasseB` e um pacote nomeado `FeatureA_FeatureB_FeatureB` que contém um aspecto chamado `ClasseA_`*`Feature`*`B`. Esse aspecto implementa todas as modificações na classe `ClasseA` utilizando recursos de classes da *feature* `FeatureB`.

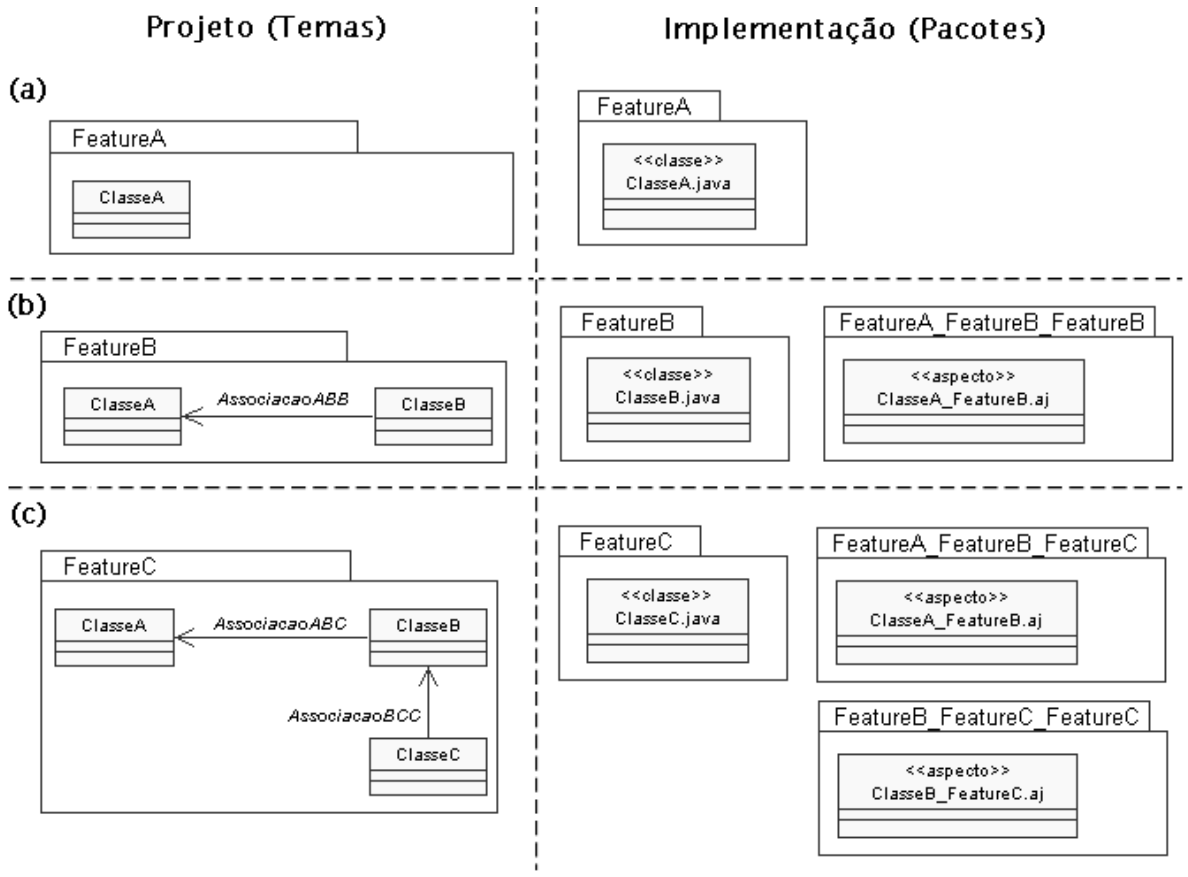


Figura 3.16: *Feature*s, Associações e Pacotes

Posteriormente cria-se uma *feature* `FeatureC` cujo projeto contém as classes `ClasseA`, `ClasseB` e `ClasseC` e as associações: `AssociacaoABC` entre as classes `ClasseA` e `ClasseB;` e `AssociacaoBCC` entre `ClasseB` e `ClasseC` (Figura 3. 16c) A implementação da *feature* `FeatureC` gera um pacote nomeado `FeatureC` que contém a classe `ClasseC`, um pacote nomeado `FeatureA_FeatureB_FeatureC` e um pacote nomeado `FeatureB_FeatureC_FeatureC`. O pacote `FeatureA_FeatureB_FeatureC` contém um aspecto chamado `ClasseA_FeatureB` que implementa as modificações na classe `ClasseA` utilizando recursos de classes da *feature* `FeatureB`. O pacote `FeatureB_FeatureC_FeatureC` contém um aspecto chamado `ClasseB_FeatureC` que implementa as modificações na classe `ClasseB` utilizando recursos de classes da *feature* `FeatureC`.

É importante ressaltar que o pacote `FeatureA_FeatureB_FeatureB` é diferente do pacote `FeatureA_FeatureB_FeatureC`. Ambos contém um aspecto chamado `ClasseA_FeatureB`, mas esses aspectos são diferentes, pois as necessidades da *feature* `FeatureB` que envolvem as *features* `FeatureA` e `FeatureB` são diferentes

das necessidades da *feature* `FeatureC` que envolvem as *features* `FeatureA` e `FeatureB`.

Isso ocorre pelo fato de duas classes poderem se associar de maneiras diferentes, dependendo de qual *feature* causa a associação. Por exemplo, existe um aspecto de nome *`ClasseA_FeatureB`* no pacote `FeatureA_FeatureB_FeatureB`, e esse aspecto é responsável por fazer a associação de classes da *feature* `FeatureB` na classe `ClasseA` que são causadas pela presença da `FeatureB`. Existe ainda um outro aspecto de mesmo nome no pacote `FeatureA_FeatureB_FeatureC`, que é responsável por fazer a associação de classes da *feature* `FeatureB` na classe `ClasseA` que são causadas pela presença da `FeatureC`. Caso as duas *features* `FeatureA` e `FeatureB` causarem uma associação igual entre a classe `ClasseA` e a *feature* `FeatureB`, o aspecto é reutilizado, sendo colocado nos dois pacotes.

Assim, para facilitar ainda mais o reúso da *feature*, cria-se um artefato denominado Relação *Features*-Pacotes, que é uma relação dos pacotes pertinentes a cada *feature*, chamados de pacotes a incluir (p.i.).

Como algumas *features* podem requerer que se desfaçam associações anteriores, na relação também são enumerados, para cada *feature*, os pacotes que não se deve utilizar quando a *feature* for escolhida, chamados de pacotes a excluir (p.e.).

A relação também traz a informação das dependências entre *features*, isto é, *features* que são requeridas por cada *feature*. Essa informação já estava presente e é copiada do modelo de *features*. A Tabela 3.5 é um exemplo de Relação *Features*-Pacotes para o exemplo da Figura 3.16. A relação é aumentada toda vez que uma nova *feature* é desenvolvida. Ela é utilizada no momento da instanciação dos produtos, que é a atividade de composição F3.2 da Figura 3.1.

Associações que só existem no sistema desejado se duas *features* estiverem presentes são indicadas colocando-se as duas *features* que devem estar presentes no campo Nome Feature, como é o caso das três últimas linhas da Tabela 3.5. A quarta linha indica que o pacote `FeatureA_FeatureB_FeatureB`, ou seja, a associação entre classes da *feature* `FeatureA` e classes da *feature* `FeatureB` causadas pela *feature* `FeatureB`, obviamente só deve estar presente no sistema se ambas as *features* `FeatureA` e `FeatureB` estiverem presentes.

Tabela 3.5: Exemplo de Relação *Feature*s Pacotes

Núm. *Feature*	**Nome *Feature***	**Requer *Feature***	**Pacotes a Incluir (p.i.)**	**Pacotes a Excluir (p.e.)**
1	*Feature*A		*Feature*A	
2	*Feature*B		*Feature*B	
3	*Feature*C		*Feature*C	
	*Feature*A *Feature*B		*Feature*A_*Feature*B_*Feature*B	
	*Feature*A *Feature*C		*Feature*A_*Feature*B_*Feature*C	*Feature*A_*Feature*B_*Feature*B
	*Feature*B *Feature*C		*Feature*B_*Feature*C_*Feature*C	

3.6. Combinação das Features – F3.2

O desenvolvimento de produtos envolve a escolha e combinação das *feature*s correspondentes às funcionalidades desejadas para o produto. Para cada *feature* escolhida deve-se localiza-la no modelo de *feature*s e verificar na relação *feature*s-pacotes se ela requer outras *feature*s. Assim, chega-se ao conjunto total de *feature*s que devem estar presentes no produto. Dessas *feature*s, verifica-se quais já estão implementadas no repositório. *Feature*s que ainda não estão implementadas precisam ser. Isso significa que as *feature*s são desenvolvidas sob demanda das instanciações de produtos. O desenvolvimento das *feature*s é feito seguindo as diretrizes da Seção 3.5.

Quando todas as *feature*s estiverem prontas, o produto pode ser montado. A relação *feature*s pacotes é consultada para que se saiba quais pacotes incluir no produto ("p.i."). Deve-se prestar atenção aos pacotes a excluir, indicados com "p.e." na relação. Tendo-se todos os artefatos de código necessários, a combinação (do inglês *weave*) das classes e aspectos é feita utilizando-se o combinador apropriado para a linguagem. A linguagem AspectJ, utilizada neste trabalho, utiliza o combinador AJC (AJC, 2006) .

Na Figura 3.17 estão as atividades do desenvolvimento de produtos e seus artefatos.

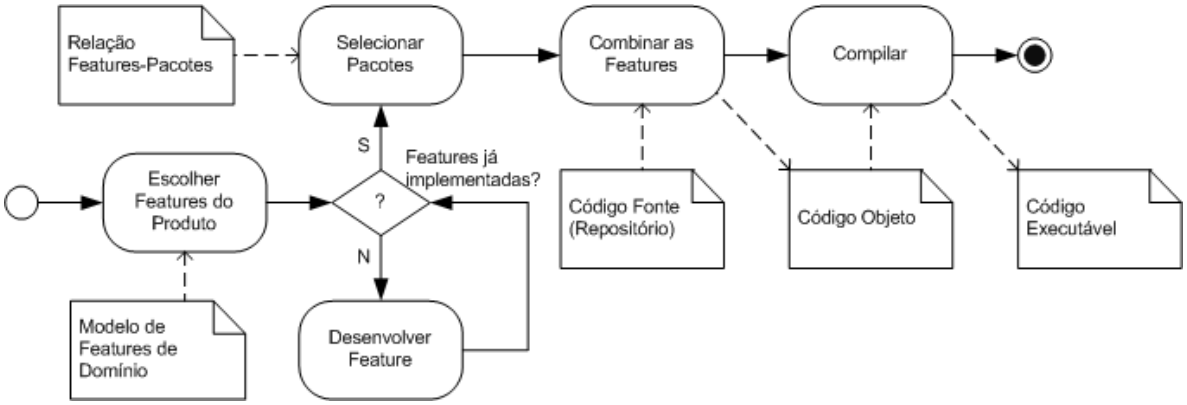


Figura 3.17: Fase de Desenvolvimento dos Produtos

3.7. Considerações Finais

Diante das atuais pesquisas sobre problemas ou partes específicas do desenvolvimento de software OA, este trabalho buscou agrupar algumas das idéias e técnicas dessas pesquisas em uma abordagem de desenvolvimento. A abordagem não chega a ter os detalhes de um processo, mas pode servir de base para a evolução de um processo de desenvolvimento de software OA. Foram incorporadas algumas das pesquisas existentes sobre técnicas e soluções de problemas específicos. Além disso, há a possibilidade de, com o aprimoramento dessas técnicas e surgimento de novas, incorporá-las ou substituir partes na abordagem proposta, pois a abordagem deixa margem para evolução. Todas as fases, sub-fases, passos, métodos e artefatos descritos na abordagem podem evoluir com contribuições futuras rumo a um processo operacional, como desejado para desenvolvimento de software orientado a aspectos.

A abordagem foi preparada para ser utilizada no contexto de criação de linhas de produtos de software, que é mais abrangente e complexo do que o contexto de desenvolvimento de software convencional. Porém, nada impede que partes específicas da abordagem relativas ao desenvolvimento de linhas de produtos sejam desconsideradas (por exemplo, análise de domínio) e sejam consideradas, por exemplo, apenas partes relativas à análise e projeto. O processo pode ser usado com flexibilidade e o desenvolvedor pode utilizar apenas partes da abordagem para gerar os artefatos que julgar necessário, com devidas exceções.

A abordagem tem uma seqüência de atividades bem definidas para análise, projeto e implementação da linha de produtos, iniciando-se com a análise de domínio. Foram acrescentadas as seguintes contribuições no processo de análise de domínio:

- A criação de um documento, chamado Documento de *Features* (DF), que resume e descreve os conceitos importantes do domínio.
- A criação, por meio da engenharia reversa, para cada um dos softwares analisados do domínio, dos seguintes documentos: Documento de *features*, documento de requisitos, diagrama de casos de uso, modelo conceitual, modelo de *features* e um documento indicando a relação entre os demais documentos, chamado documento de mapeamento.
- O método de mapeamento entre toda a documentação criada para o produto, a qual permite entender os motivos das decisões de projeto e rastrear de onde partiu cada uma delas.
- Um modo simples de elicitação do que é comum e do que variável na linha de produtos, fazendo uma comparação dos documentos para cada sistema analisado no domínio, sendo que a comparação principal é entre modelos de *features*.

A atividade seguinte, o desenvolvimento da base, cria as *features* obrigatórias da linha de produtos. O desenvolvimento de *features* segue uma metodologia que é uma contribuição deste trabalho, feita com base em outros trabalhos publicados recentemente ou já bem aceitos. A metodologia combina técnicas de análise e projeto orientadas a aspectos e técnicas para implementação que facilitam a evolução e manutenção da linha de produtos. Uma sugestão de diagrama de colaboração com aspectos foi apresentada por ser necessária para a criação de um projeto orientado a aspectos.

Espera-se, com isso, conseguir a redução do esforço desprendido na criação de linhas de produtos, principalmente o esforço inicial (Figura 3.18a – o esforço esperado com a abordagem descrita é mostrada em 3.18b). A utilização dos aspectos pode também pode causar diminuição do esforço para criação dos produtos. Esses dados, entretanto, só poderiam ser realmente comprovados com estudos mais detalhados.

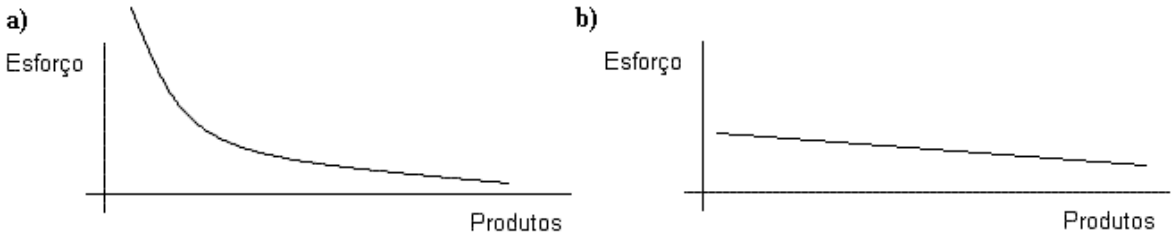


Figura 3.18: Gráficos Esforço por Produtos

4. Estudo de caso

4.1. Considerações Iniciais

Este capítulo apresenta o estudo de caso realizado neste trabalho. Ele consistiu no desenvolvimento de uma linha de produtos no domínio de sistemas de clínicas de psicologia. A escolha do domínio foi relacionada à oportunidade proporcionada por um projeto desenvolvido em paralelo, o GestorPsi (ver seção 4.2).

O objetivo inicial do estudo de caso era investigar se os aspectos são benéficos para linhas de produtos e que a abordagem Tema é adequada para a análise e projeto da linha de produtos orientada a aspectos. Porém, durante o desenvolvimento da linha de produtos, surgiram problemas e dificuldades decorrentes da utilização das técnicas e tecnologias mencionadas. Dessa maneira, foi necessário pesquisar e utilizar técnicas mais apropriadas para linhas de produtos e fazer adaptações para utilizar técnicas distintas em conjunto.

Assim, o estudo de caso acabou por servir para definir uma abordagem para desenvolvimento de linhas de produtos orientadas a aspectos apresentada no Capítulo 3.

Este capítulo está organizado como segue: A Seção 4.2 descreve o projeto GestorPsi. A Seção 4.3 descreve as clínicas de psicologia estudadas para a realização do estudo de caso. Na Seção 4.4 é explicado como foi feita a análise de domínio. Na Seção 4.5 o desenvolvimento da base da LP para o GestorPsi é descrito. Na Seção 4.6 é descrito o desenvolvimento de *features* opcionais do GestorPsi. Na seção 4.7 é explicado como foi feita a instanciação de produtos da linha com a composição de *features*. Por fim, na Seção 4.8 são apresentadas as considerações finais.

4.2. Projeto GestorPsi

O GestorPsi (GestorPsi, 2006) é um Sistema de Gestão de Serviços em Psicologia e está sendo desenvolvido pelo Instituto de Psicologia Comportamental de São Carlos sob a coordenação do psicólogo Oliver Zancul Prado, com colaboração de diversas entidades públicas e privadas, dentre elas o Instituto de Ciências Matemáticas e de Computação (ICMC) da USP de São Carlos por intermédio da professora Rosana Teresinha Vaccare Braga.

O GestorPsi nasceu da demanda pela criação de um sistema informatizado que possa trazer a psicologia para a era digital, possibilitando a criação de métodos padronizados para:

- O registro de informações clínicas;
- O gerenciamento de informações administrativas e financeiras;
- O gerenciamento da prestação de serviços psicológicos; e
- O registro de diagnósticos, avaliações e resultados.

O projeto teve apoio da FAPESP na linha de financiamento PIPE (Programa de Inovação tecnológica em Pequenas Empresas), fase I, e está sendo solicitado o apoio para a fase II. Na fase I foi desenvolvida toda a especificação do sistema e na fase II será realizado o desenvolvimento de fato. Neste trabalho de mestrado, o GestorPsi é utilizado como estudo de caso, desenvolvendo-se um protótipo para algumas partes do sistema. Contudo, deve-se ressaltar que esse protótipo não tem a intenção de ser operacional, pois na fase II do projeto o sistema será reconstruído.

A ocorrência desse projeto viabilizou o acesso às informações necessárias para o estudo do domínio, como material bibliográfico, sistemas existentes, especialistas e usuários. O projeto já havia sido iniciado e parte da análise de domínio já havia sido concluída. Os produtos dessa análise de domínio foram vistos e considerados, mas não foram utilizados no processo de desenvolvimento, pois era desejado testar o processo desde o início e desenvolver os artefatos de acordo com o processo.

A seguir são descritas algumas das clínicas que tiveram seu funcionamento estudado e documentado pelo projeto GestorPsi.

4.3. Descrição das Clínicas psicológicas estudadas

A compreensão do funcionamento das clínicas de psicologia, bem como dos sistemas de informação nelas utilizados foi importante para o estudo de domínio. Foram estudadas três clínicas, descritas a seguir.

Clínica Escola da PUC-SP "Ana Maria Poppovic"

A clínica da Pontíficia Universidade Católica de São Paulo (PUC-SP) (PUC-SP, 2006), também batizada de "Ana Maria Poppovic" atende à população com uma grande variedade de serviços de psicologia e áreas afins.

A clínica presta alguns serviços pagos pelos pacientes, mas também concede o benefício da gratuidade no atendimento para pacientes com condições econômicas desfavoráveis. A clínica também auxilia alguns pacientes com ajuda de custo para transporte para a realização dos tratamentos.

Por ser uma clínica-escola vinculada à PUC-SP, o trabalho lá desenvolvido também tem como objetivo a formação profissional de alunos de graduação da PUC, alunos de pós-graduação e profissionais que realizam cursos de especialização. A clínica é mantida pela universidade e também recebe doações que ajudam em suas despesas de manutenção e funcionamento.

É uma das mais completas e abrangentes em termos de serviços e atividades.

Clínica Escola da Faculdade Paulistana

É uma clínica-escola vinculada à Faculdade Paulistana (Paulistana, 2006). Tem como objetivo o estágio clínico/educacional dos alunos do Curso de Psicologia e o oferecimento de serviços à comunidade em geral. Todos os serviços de saúde prestados para a população são gratuitos.

Instituto de Psicologia Comportamental de São Carlos

O Instituto de Psicologia Comportamental de São Carlos (IPC) (IPC, 2006) é uma clínica particular de pequeno porte. A clínica possui alguns profissionais da área de psicologia e áreas afins, como fonoaudiologia, e presta diversos serviços nessas áreas. Uma característica importante dessa clínica é que ela tem se expandido, aumentando a quantidade de profissionais e aumentando a variedade de serviços que oferece.

4.4. Análise de domínio

A primeira fase da abordagem foi o estudo de domínio. A análise dos sistemas existentes foi a principal atividade indicada e seguida. O estudo do funcionamento das clínicas também foi muito importante. Para isso, foi utilizado material fornecido pelo projeto GestorPsi, em que entrevistas já haviam sido feitas para coleta de dados. Para algumas das clínicas, o projeto forneceu um documento textual descrevendo o funcionamento das rotinas da clínica. O documento mostrava formulários, planilhas e outros artefatos que a clínica utilizava e explicava como eles eram utilizados. Algumas

clínicas também possuíam sistemas de informação que eram parte integrante das atividades rotineiras. Em certas clínicas havia vários sistemas diferentes que não estavam integrados, além da utilização de outras formas de registros não eletrônicos. Isso tornava o trabalho dos funcionários da clínica mais difícil, lento e sujeito a erros. O projeto GestorPsi evidenciou a carência uma família de sistemas para esse domínio.

De acordo com a abordagem, foi feita uma análise individual para cada clínica, que produziu artefatos para cada uma delas. A análise consistiu em observar quais eram as tarefas realizadas, quais informações eram relevantes para as tarefas e quais eram as entidades envolvidas. Outros fatores importantes observados foram o modo como essas informações deveriam ser armazenadas e como elas deveriam ser exibidas em sua recuperação.

A primeira clínica estudada foi a da PUC-SP e o primeiro artefato a começar a ser produzido foi o Documento de Features. Ele reuniu os conceitos que foram identificados. Na Tabela 4.1 é exibida uma parte do documento de features. Pode-se observar nessa tabela os conceitos (pré-*features*) `Paciente`, `Dados Sócio-Econômicos` e `Dados Familiares`.

O processo de identificação dos conceitos continuou até que todo o material sobre a clínica fosse analisado. O documento de requisitos foi feito em paralelo com o DF. A elicitação dos requisitos foi feita com a identificação e compreensão dos conceitos e a identificação das atividades realizadas pela clínica, de acordo com o que estava escrito a respeito do seu funcionamento e de acordo com o que foi observado das telas do sistema que a clínica utilizava quando havia um (engenharia reversa).

Foram consideradas como funcionalidades as necessidades a respeito da manipulação dos dados sobre os conceitos identificados no documento de *features*. Isso porque para muitos conceitos, suas informações precisavam ser armazenadas para consultas.

Na Tabela 4.2 são exibidos alguns dos requisitos para o conceito `Paciente` e para alguns de seus sub-conceitos (requisitos de 1 a 3). O requisito 24 é um dos requisitos para o conceito `Agendamento`.

Tabela 4.1: Parte do documento de features

1. Paciente Atributos: nome; nascimento; sexo; endereço; bairro; CEP; cidade; estado; telefone; fax; e-mail; CPF e RG. Funcionalidades: Persistir os dados; Permitir consultas. Observações: Representa o registro dos dados pessoais de um Paciente. Abrange documentação, endereço, contatos e descrição pessoal.
1.1. Dados sócio-econômicos Atributos: renda familiar (salários mínimos): [00-03, 03-05, 05-10, 10-20, 20+, sem info]; profissão; local de trabalho; da clínica [s, n]; cargo; telefone comercial; moradia; tipo: [cedida, alugada, própria]; quitada: [s, n]; número de cômodos; obs; saneamento/serviço rede pública ou privada; água encanada; energia elétrica; esgoto; asfalto; TV a cabo; (posses) TV; vídeo; microondas; geladeira; rádio; máq. de lavar louça; máq. de lavar roupa; microondas; (transporte público) ônibus; metrô; trem; lotação; (transporte particular) modelo; ano; outros (especificar); (assistência médica) particular; pública; convênio (especificar); (gastos relevantes da família) descrição; valor; tipo: [água, luz, aluguel, condomínio, prestações moradia, prestações veículo, mensalidade escola, mensalidade clube, pensão, outros]; observações. Funcionalidades: Formulário sócio econômico. Persistir os dados. Observações: Os dados do formulário sócio econômico que o paciente preenche são armazenados para consultas.
1.2. Dados de familiares Atributos: nome; parentesco; sexo; idade; estado civil; escolaridade; profissão/ocupação; da clínica [s, n]. Funcionalidades: Persistir os dados. Permitir consulta. Observações: Salvar informações, quando necessário, sobre parentes de Pacientes.

Para ter o registro de a qual *feature* pertence cada requisito, foi feita a relação *feature*-requisitos. A Tabela 4.3 mostra uma parte da relação *feature*-requisitos da clínica da PUC para os requisitos mencionados na Tabela 4.2 e as *features* da clínica da PUC, exibidos na Figura 4.1. A *feature* 5.1 é a *feature* `Agendamento` e a *feature* 9 é a *feature* `Servico`.

Para a compreensão das necessidades do sistema e melhor esclarecimento sobre os requisitos, foram necessárias algumas consultas aos psicólogos do projeto GestorPsi. O domínio de clínicas de psicologia não apresentou muitos sinônimos ou homônimos e as entrevistas foram suficientes para sanar as poucas dificuldades nesse sentido. Um exemplo dos poucos sinônimos encontrados foram `Agendamento` e `Consulta`.

Tabela 4.2: Parte dos requisitos da linha de produtos

1. O sistema deve permitir a inclusão, consulta, alteração e remoção de pacientes (dados cadastrais) da clínica, com os seguintes atributos: nome, data de nascimento, sexo, endereço, bairro, CEP, cidade, estado, telefone, fax, e-mail, CPF e RG.
2. O sistema deve permitir a inclusão, consulta, alteração e remoção de informações sócio-econômicas de pacientes da clínica, com os seguintes atributos: renda familiar (sal min): [0-3; 3-5; 5-10; 10-20; 20+; nd] profissão local de trabalho da clínica [s; n] cargo telefone comercial moradia tipo: [cedida; alugada; própria] quitada: s/n. número de cômodos Obs serviços / saneamento / eletricidade água encanada, energia elétrica, esgoto, asfalto, tv a cabo, telefone, internet rápida posses tv, vídeo, microondas, geladeira, rádio, lava louça, lava roupa, microondas, computador. transporte público [ônibus; metrô; trem; lotação] particular quantidade, modelo, ano. outros - especifique assistência médica particular; pública. convênio (qual) gastos relevantes da família descrição, valor, observações, tipo: [água; luz; aluguel; condomínio; prestações moradia; prestações veículo; mensalidade escola; mensalidade clube; pensão; outros]
3. O sistema deve permitir a inclusão, consulta, alteração e remoção de informações sobre familiares dos pacientes da clínica com os seguintes dados: nome, parentesco, sexo, idade, estado civil, escolaridade, profissão/ocupação, é da clínica (s/n), função, curso, ano, setor
7 – O sistema deve permitir a inclusão, consulta, alteração e remoção de informações sobre o serviço que o paciente está recebendo da clínica com os seguintes dados: Se é encaminhado de outra instituição s/n, nome da instituição e nome do terapeuta. Serviço (pré cadastrado). Paciente Terapeuta Horário. Disponibilidade de horários do paciente: manhã, tarde, noite. Situação: AM – (arquivo morto), AG (aguardo), AT (atendimento). Conclusão: arquivado por faltas, arquivado sem relatório, desistência, encerramento do caso (Alta), falecimento, final do processo, mudança de endereço, não localizado, vencimento da triagem (aguardou muito tempo).
24 – O sistema deve permitir a inclusão, consulta, alteração e remoção de agendamentos de consultas/atendimentos na clínica com os seguintes dados: nome do paciente, nome do terapeuta, sala, dia, horário, serviço sendo prestado e núcleo/grupo/laboratório que está atendendo.
42 – O sistema deve permitir a inclusão, consulta, alteração e remoção de informações sobre os Serviços da clínica com as seguintes informações: nome, descrição, responsável (não obrigatório), capacitados (preenchido pelo lado dos terapeutas) e abordagens.

Tabela 4.3: Exemplo de relação *Feature*-Requisitos

Feature	**Requisitos**
1	R1
1.1	R2
1.2	R3
5.1	R24
9	R7, R42

O processo de identificação dos requisitos continuou até que todo o material sobre a clínica fosse analisado. O próximo artefato produzido foi o modelo de *features*. Para sua criação, o documento de *features* foi analisado.

Os conceitos foram verificados, e nos casos onde a persistência, gerenciamento e consulta dos dados do conceito eram necessárias, essas funcionalidades para o conceito se tornaram uma *feature* com o nome do conceito. As demais funcionalidades necessárias relativas ao conceito se tornaram sub-*features* da *feature* do conceito. O modelo de *features* criado para a clínica da PUC pode ser visto na Figura 4.1.

Isso concluiu a análise individual do sistema utilizado na clínica da PUC e produziu os quatro artefatos necessários, o DF, o documento de requisitos, o modelo de *features* e a relação *features*-requisitos. Uma análise individual semelhante foi feita para a clínica da Faculdade Paulistana e para o IPC.

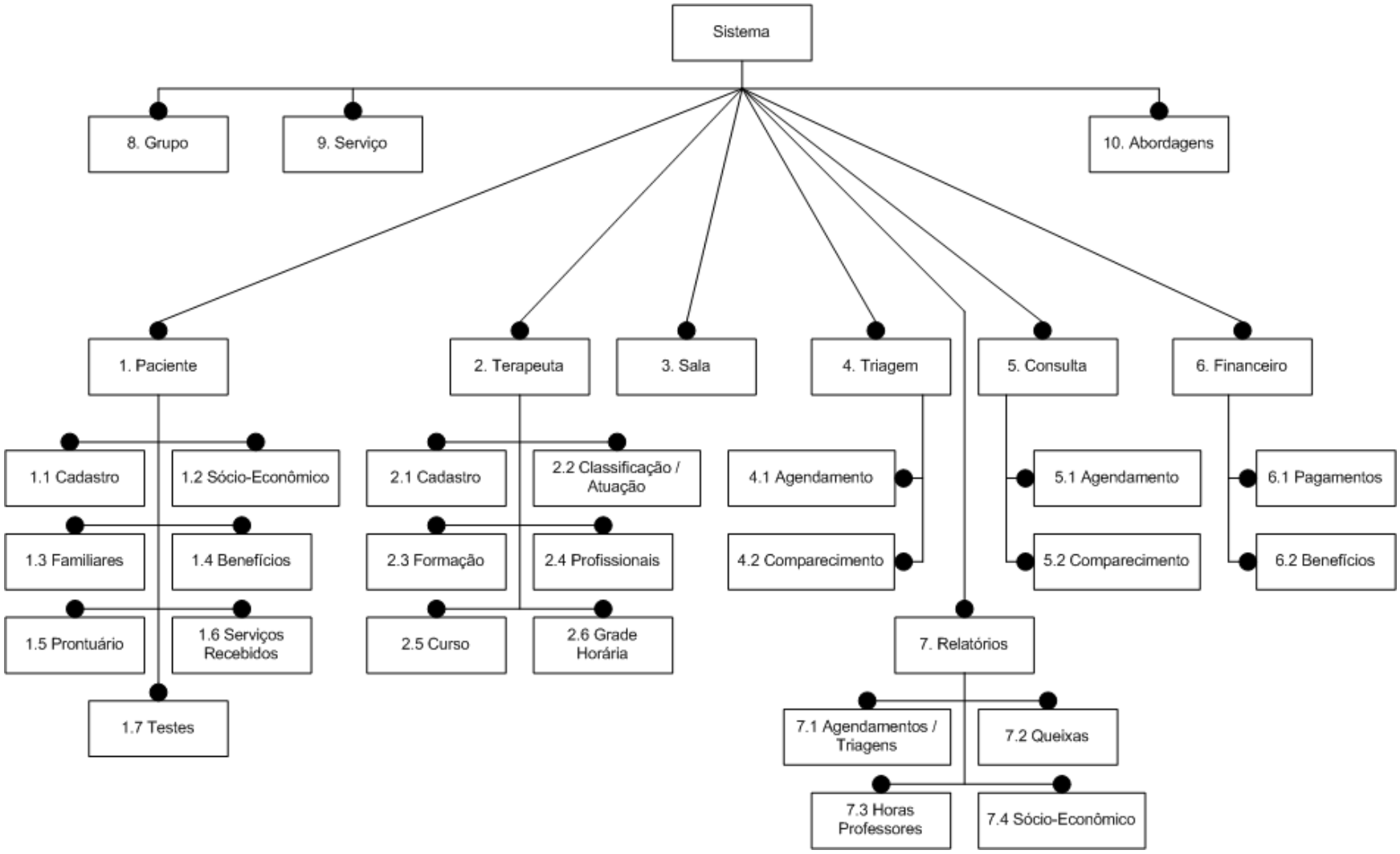


Figura 4.1: Modelo de *features* para o sistema da PUC

Os artefatos produzidos nas análises individuais foram então utilizados para completar a análise de domínio. Os modelos de *features* foram comparados para se produzir o modelo de *features* de domínio, segundo as regras definidas na Seção 3.3.5. Dessa maneira também foram obtidos o documento de requisitos e a relação *features* requisitos – versões de domínio, e as visões de ação. O modelo de *features* versão de domínio pode ser visto na Figura 4.2.

De posse dos artefatos e conhecimentos necessários, iniciou-se a próxima fase, o desenvolvimento da base, o qual é comentado na próxima Seção.

4.5. Desenvolvimento da Base

Conforme pode ser visto na Figura 4.2, as *features* com círculos preenchidos são as *features* obrigatórias que formam a base da LP. São elas: `Paciente`, `Terapeuta`, `Sala`, `Agendamento` e `Relatorio`.

Não foi necessário desenvolver todas as *features* obrigatórias já que, por ser apenas um protótipo, o sistema não seria usado na prática e isso não afetaria as intenções do estudo de caso de estudar as técnicas de desenvolvimento e composição das *features*. Iniciou-se o desenvolvimento pela *feature* `Paciente`. O requisito para `Paciente` é o requisito 1 da Tabela 4.2. Esse requisito havia passado do documento de requisitos versão de sistema para um documento de requisitos versão de domínio na fase anterior, juntamente com a *feature* `Paciente`.

O diagrama de classes de projeto da *feature* `Paciente` pode ser visto na Figura 4.3. No projeto, as classes de lógica de aplicação exibem os atributos que correspondem à representação dos objetos, mas quando forem implementadas, os atributos serão substituídos por meta-atributos, conforme definido na Seção 3.5.2.1.

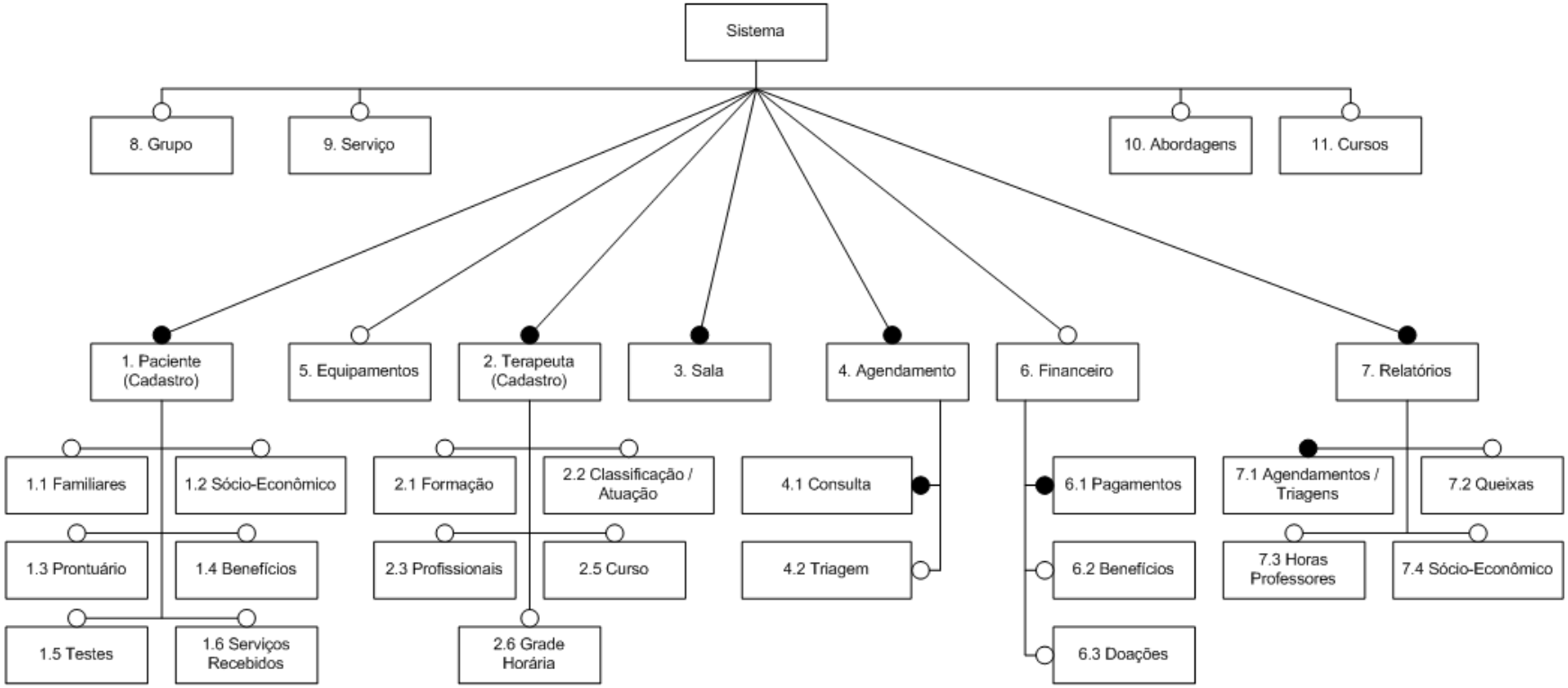


Figura 4.2: Modelo de *Features* do domínio de Clínicas de Psicologia

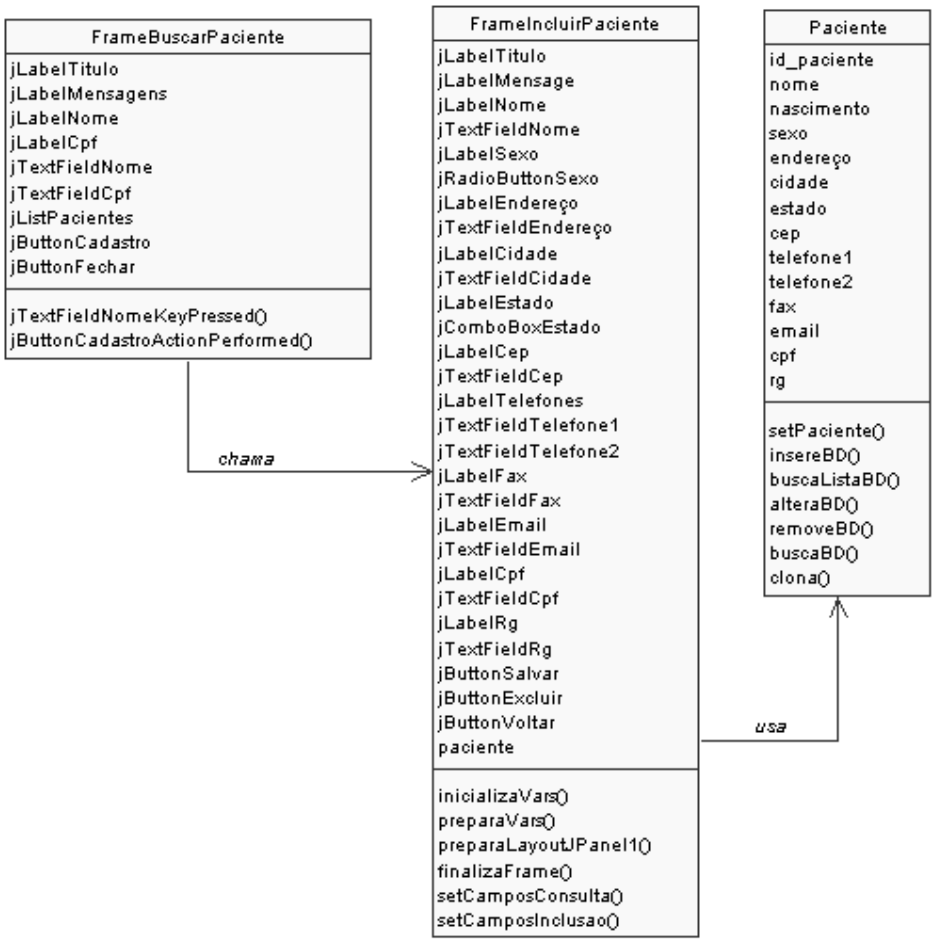


Figura 4.3: Projeto da *Feature* `Paciente`

As próximas *features* desenvolvidas foram `Prontuario` (sub-*feature* de `Paciente`), `Terapeuta`, `Sala` e `Agendamento`. Dessas, `Terapeuta` e `Sala` não tinham menção de outras *features*, e seu desenvolvimento foi semelhante ao desenvolvimento da *feature* `Paciente`, o que não acrescentou novidades interessantes do ponto de vista do desenvolvimento orientado a aspectos. A *feature* `Prontuario` se relacionava com a *feature* `Paciente` e a *feature* `Agendamento` mostrou um entrelaçamento de interesses complexo, o qual será apresentado como exemplo.

O requisito para essa *feature* (Agendamento) é o requisito 24, visto na Tabela 4.2 (Seção 4.3). Há um envolvimento das *features* `Paciente`, `Terapeuta`, `Sala`, `Grupo` e `Servico`. O entrelaçamento de interesses é evidenciado pela visão de ação exibida na Figura 4.4. A visão de ação aparada foi criada para indicar como é a relação entre as *features* e pode ser vista na Figura 4.5.

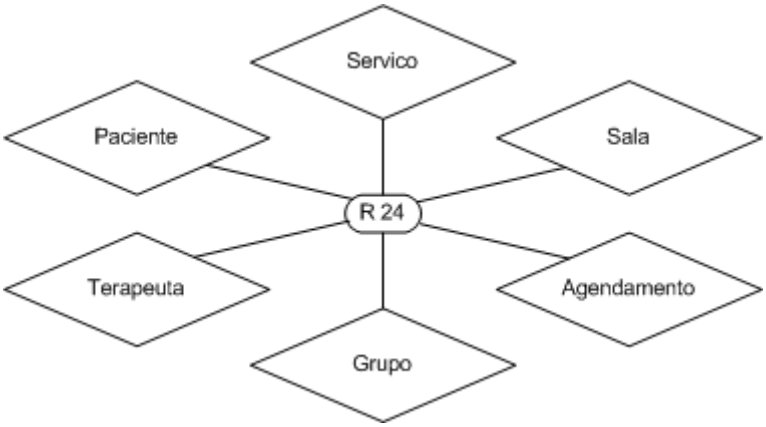


Figura 4.4: Visão de Ação da *Feature* `Agendamento`

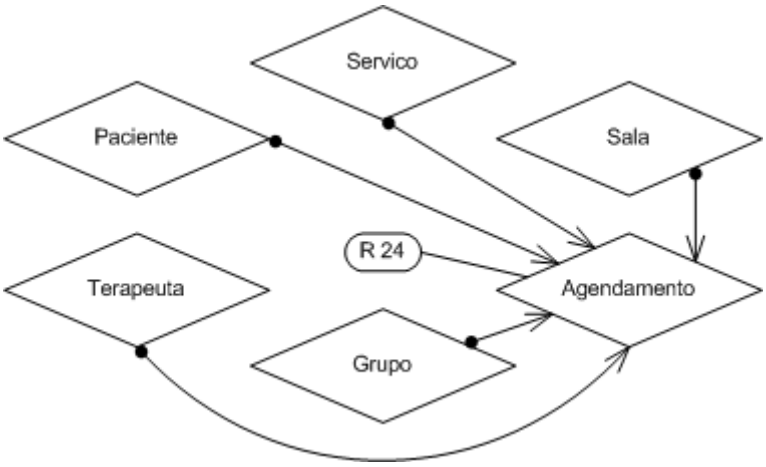


Figura 4.5: Visão de Ação Aparada da *Feature* `Agendamento`

Etapa 1

Tendo decidido que as *features* `Paciente`, `Terapeuta`, `Sala`, `Servico` e `Grupo` irão aumentar e ou modificar a *feature* `Agendamento`, e que `Agendamento` é a principal *feature* do requisito, o requisito é reescrito para mencionar apenas `Agendamento`:

> 24 O sistema deve permitir a inclusão, consulta, alteração e remoção de agendamentos de consultas/atendimentos na clínica com os seguintes dados: dia, horário.

De acordo com as informações do domínio, foi elaborada uma interface para criar os agendamentos, que inclui uma grade horária com os dias e horas disponíveis para agendamento de consulta. A Figura 4.6 contém uma imagem do protótipo feito antes do desenvolvimento da *feature*. Esse protótipo de interface foi feito com base no requisito para `Agendamento` reescrito, livre de influências, por isso aparece apenas

"reservado" no horário do agendamento. O diagrama de classes da *feature* `Agendamento` pode ser visto na Figura 4.7. Todas as classes, exceto `Agendamento`, são classes que compõem a GUI.

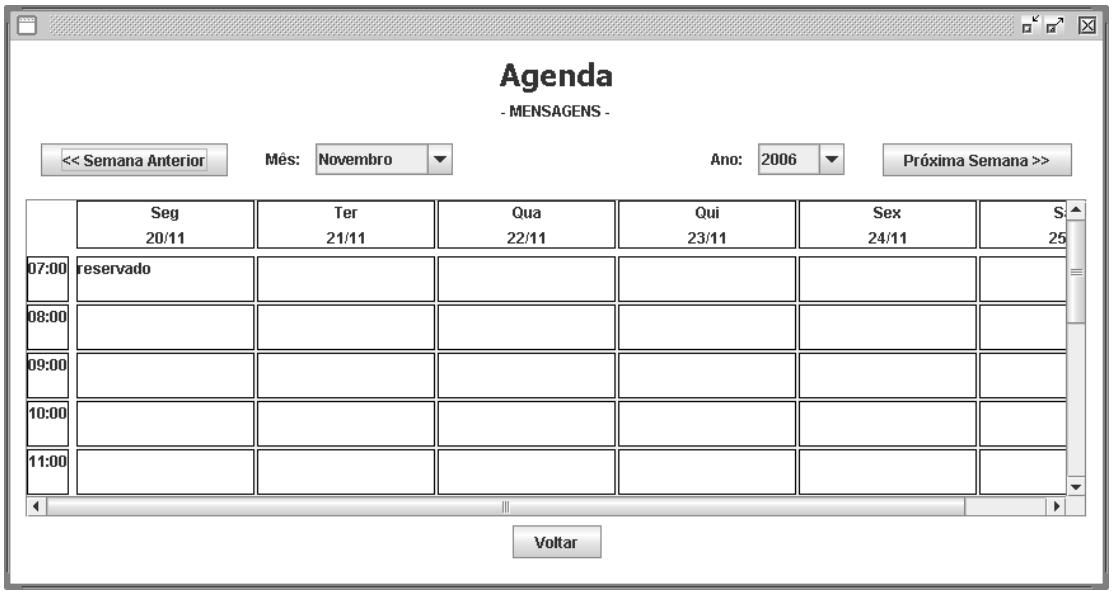


Figura 4.6: Interface do Agendamento – Etapa 1

As classes de `Agendamento` (Figura 4.7) compõem a interface (Figura 4.6) da seguinte maneira: a janela principal mais externa vista na Figura 4.7 é a classe `FrameAgenda`. Ela contém os campos e os botões para selecionar uma semana específica para ser visualizada. A classe `PanelDiaSemana` exibe os dias da semana (de segunda a sábado) e dias do mês correspondentes, que aparecem no topo da grade de horários. A classe `PanelHoras` exibe os pequenos quadrados com as horas na lateral esquerda da grade de horários. O `FrameAgenda` contém um objeto do tipo `PanelInterno`. A classe `PanelInterno` é a grade de horários em si, ou, todo o retângulo grande no meio da tela com os retângulos pequenos dentro. É um objeto de interface que serve para montar a estrutura da grade de horários. `PanelInterno` contém o método que dada uma data busca todos os `Agendamentos` daquela semana e coloca cada `Agendamento` no respectivo `PanelUmHorario` e cada `PanelUmHorario` em seu lugar. `PanelUmHorario` são os pequenos retângulos na grade horária. São responsáveis por exibir as informações do `Agendamento` da sua respectiva data e hora. Cada `PanelUmHorario` possui um objeto de lógica de aplicação `Agendamento`.

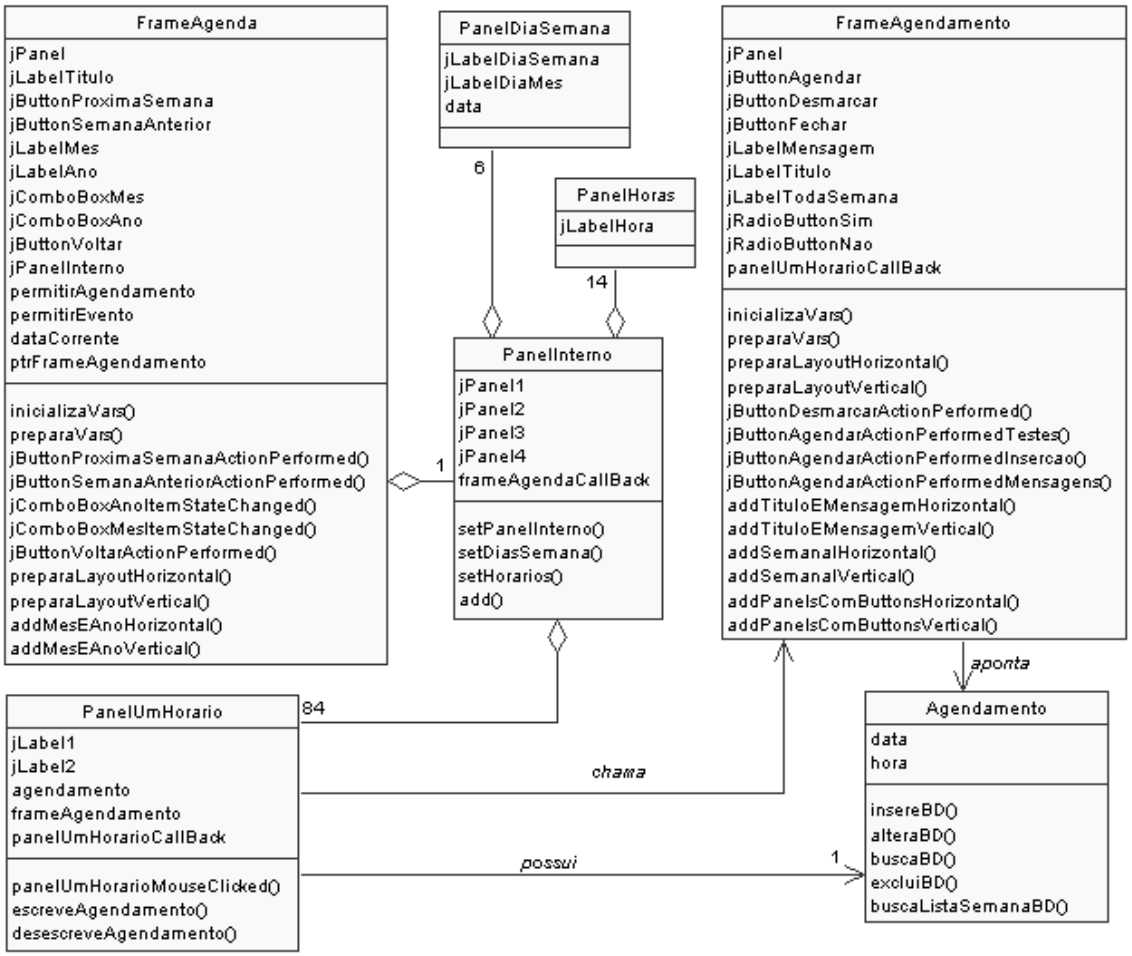


Figura 4.7: Diagrama de classes da *feature* `Agendamento` – Etapa 1

A classe `FrameAgendamento` é uma outra interface da aplicação que não aparece na Figura 4.6. Trata-se de uma janela para obter mais detalhes e confirmar o agendamento.

Os objetos contidos contém ponteiros para os objetos que os contém, chamados de `<alguma coisa>CallBack`. Isso porque eles devem funcionar como uma única estrutura, mas foram divididos para melhor funcionamento do sistema.

Os casos de uso criados foram Consultar Agendamento e Criar Agendamento e são exibidos a seguir. A seqüência de eventos do caso de uso Consultar Agendamento é exibida graficamente na Figura 4.8, e na Figura 4.9 está o diagrama de colaboração do método principal criado para o caso de uso, o método `setPanelInterno()`. Esse método cria e preenche a grade horária com os agendamentos do período.

Na Figura 4.10 é exibida a seqüência de eventos do caso de uso Criar Agendamento e na Figura 4.11 é exibido o diagrama de interação para o método

principal do caso de uso, o método `jButtonAgendarActionPerformedInsercao()`. Este método é um evento disparado quando o usuário clica no botão "Agendar" da interface `FrameAgendamento`, tendo a intenção de confirmar a criação de um agendamento, por isso ele está associado ao caso de uso Criar Agendamento.

Caso de Uso: Consultar Agendamento – Etapa 1

Atores: Secretária.

Propósito: Permitir que os agendamentos marcados sejam consultados.

Visão Geral: A secretária ou pessoa encarregada verifica os agendamentos de um determinado período.

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de consultar agendamentos.	
2. Escolhe um período.	3. Exibe a grade horária com dias e horas preenchida com agendamentos do período.

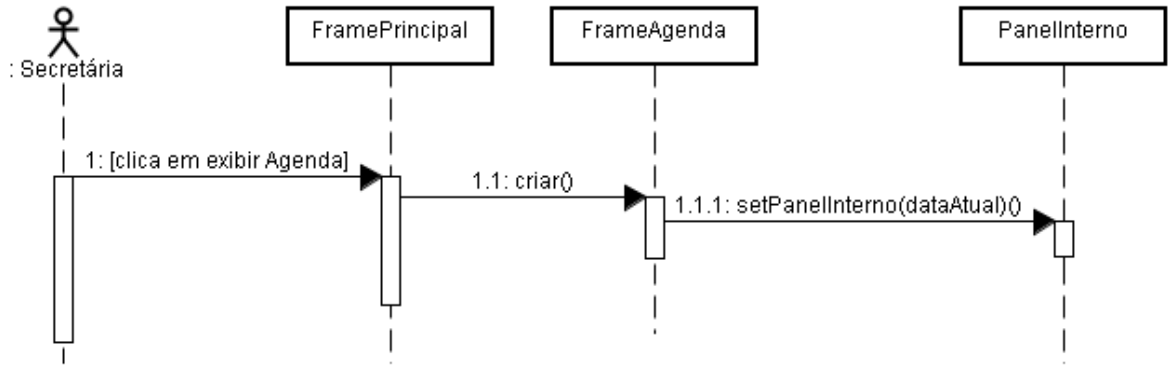


Figura 4.8: Interação do ator na exibição dos agendamentos – Etapa 1

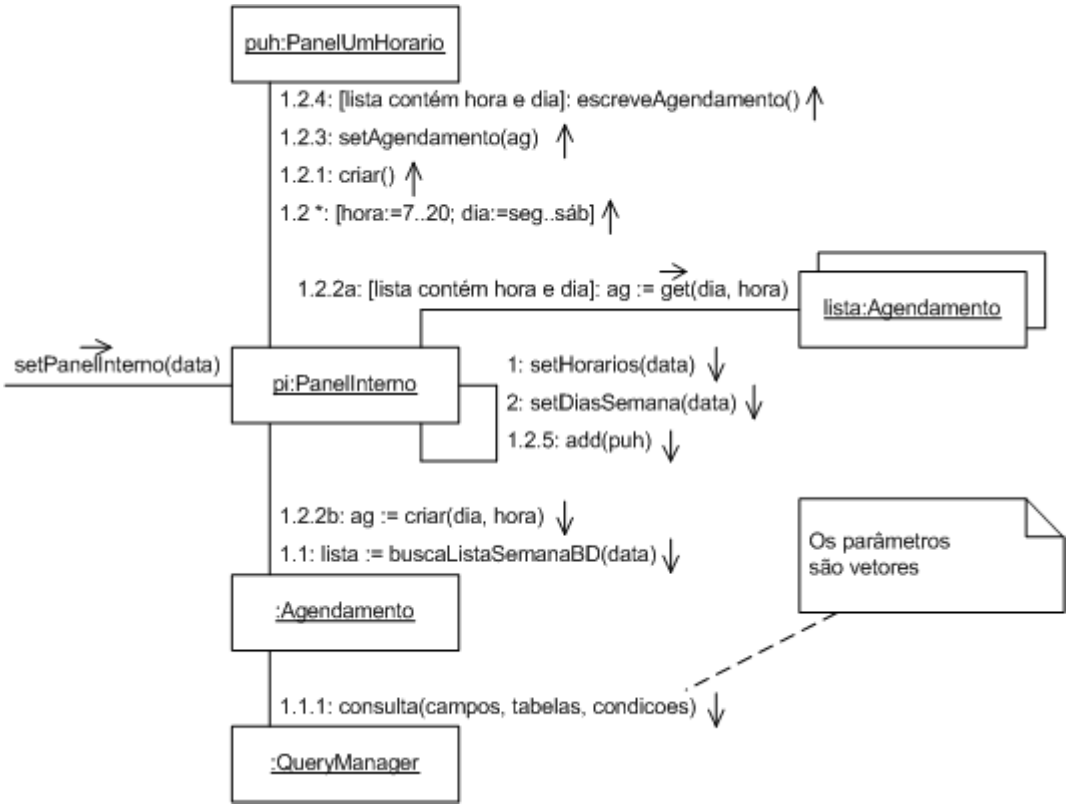


Figura 4.9: Método `setPanelInterno()`[5] – Etapa 1

Caso de Uso: Criar Agendamento – Etapa 1

Atores: Secretária.

Propósito: Permitir que um atendimento seja agendado.

Visão Geral: A secretária ou pessoa encarregada marca um agendamento.

Referência Cruzada: Consultar Agendamento

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de consultar agendamentos.	
2. Escolhe um período.	3. Exibe a grade horária com dias e horas.
4. Clica em uma data e hora da grade.	5. Exibe uma tela para confirmar o agendamento e perguntar se seria toda semana no mesmo horário
6. Escolhe se toda semana ou não.	

[5] A respeito da nota na função consulta(), os argumentos são listas (vetores) que indicam respectivamente quais campos, de quais tabelas e quais condições serão usados nas respectivas cláusulas SELECT, FROM e WHERE de uma consulta SQL. Neste caso, as listas são preparadas anteriormente para buscar todos os campos da tabela agendamento nos registros que tenham a data indicada no passo anterior.

7. Confirma o agendamento.	8. Verifica se já não existe um agendamento na data e hora escolhida.
	9. Registra o agendamento.

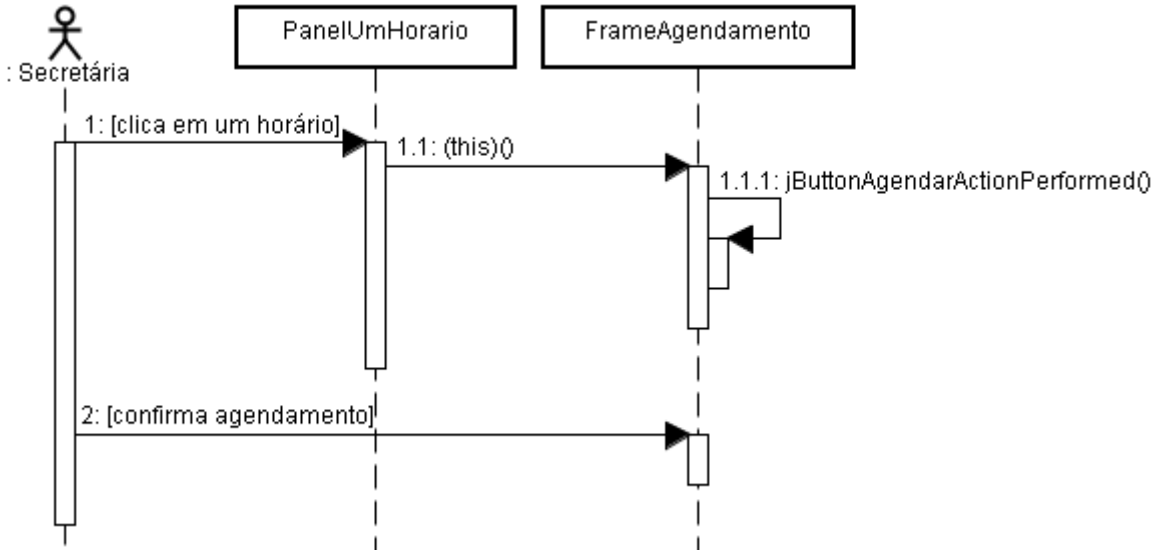


Figura 4.10: Interação do ator na criação dos agendamentos – Etapa 1

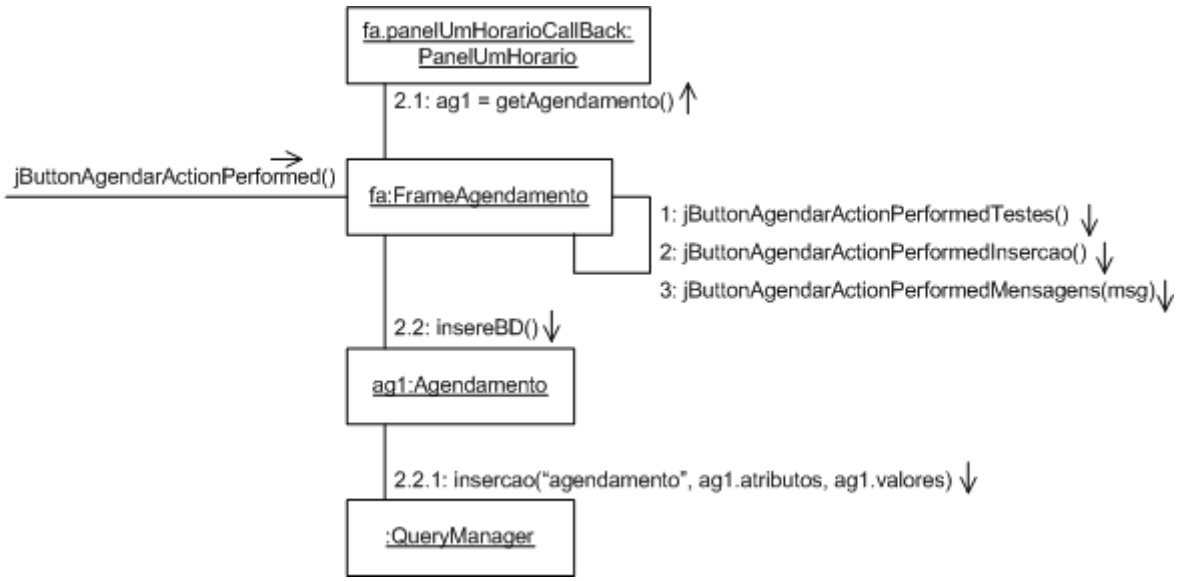


Figura 4.11: Método `jButtonAgendarActionPerformed()` – Etapa 1

Etapa 2

Os artefatos exibidos até agora completaram o projeto parcial da *feature* `Agendamento` para a Etapa 1. Na Etapa 2, deve-se projetar as influências de outras

features na *feature* atual. Como exemplo, será apresentado o desenvolvimento do projeto da influência da *feature* `Paciente`. As *features* `Servico` e `Grupo` não foram levadas em consideração nesta Etapa, pois ainda não haviam sido desenvolvidas. Isso é permitido devido à natureza incremental e iterativa da abordagem.

Para a Etapa 2 foi feito outro projeto parcial com os casos de uso e diagramas desenvolvidos a partir do requisito original. Os casos de uso estão descritos a seguir e a principal diferença com as versões da Etapa 1 é a seqüência de eventos. Na Figura 4.12 é exibido o diagrama de colaboração do método `setPanelInterno()` para a Etapa 2. As diferenças podem ser percebidas na seqüência de eventos do caso de uso. O método `jbuttonAgendarActionPerformed()` para a Etapa 2 tem um diagrama de colaboração igual ao da Etapa 1 (visto na Figura 4.11). Isso porque as diferenças causadas pela influência das outras *features* estão em detalhes omitidos no diagrama de colaboração dentro da execução de alguns dos métodos que compõem a execução do método.

Caso de Uso: Consultar Agendamento – Etapa 2

Atores: Secretária.

Propósito: Permitir que os agendamentos marcados sejam consultados.

Visão Geral: A secretária ou pessoa encarregada verifica os agendamentos de um determinado período.

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de consultar agendamentos.	
2. Escolhe um período.	3. Exibe a grade horária em branco.
4. Digita algumas letras do nome do paciente.	5. Preenche uma lista com os pacientes que tem nomes que começam com o que o usuário digitou.
6. Escolhe um paciente da lista.	7. Exibe a grade horária com dias e horas preenchida com agendamentos do período do paciente.

Caso de Uso: Criar Agendamento – Etapa 2

Atores: Secretária.

Propósito: Permitir que um atendimento seja agendado.

Visão Geral: A secretária ou pessoa encarregada marca um agendamento.

Referência Cruzada: Consultar Agendamento

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de consultar agendamentos.	
2. Escolhe um período.	3. Exibe a grade horária com dias e horas.
4. Digita algumas letras do nome do paciente.	5. Preenche uma lista com os pacientes que tem nomes que começam com o que o usuário digitou.
6. Escolhe um paciente da lista.	7. Exibe a grade horária com dias e horas preenchida com agendamentos do período do paciente.
8. Seleciona uma data e hora da grade.	9. Exibe uma tela para confirmar o agendamento. Pergunta se é toda semana.
	10. Na mesma tela exibe lista de terapeutas.
	11. Na mesma tela exibe lista de tipos de salas.
12. Escolhe se toda semana ou não.	
13. Escolhe um terapeuta.	
14. Escolhe um tipo de sala.	15. Exibe lista com as salas do tipo escolhida.
16. Escolhe uma sala.	
17. Confirma o agendamento.	18. Verifica se o paciente está ocupado na/s data/s do agendamento.
	19. Verifica se o terapeuta está ocupado na/s data/s do agendamento.
	20. Verifica se a sala está ocupada na/s data/s do agendamento.
	21. Anota que existe um agendamento na data e hora indicada.

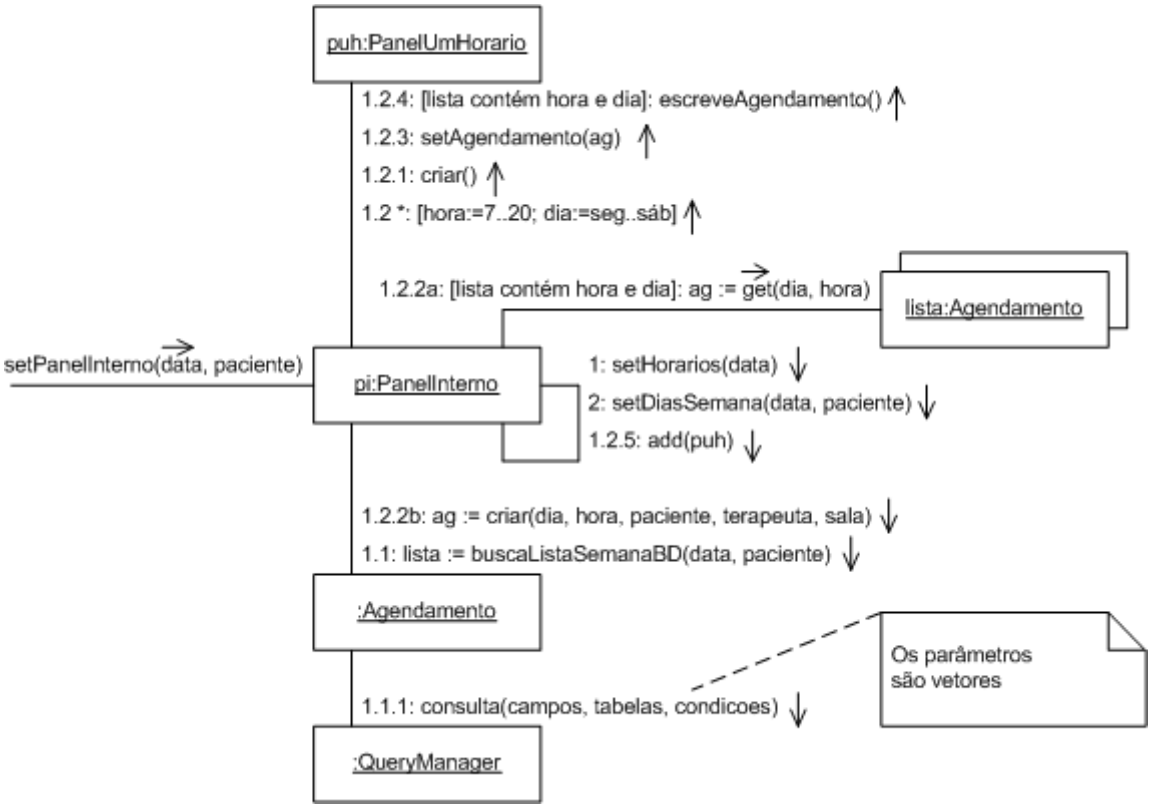


Figura 4.12: Método `setPanelInterno()` – Etapa 2

A estrutura do diagrama de classes da parte de interface se mantém igual. As classes de interface ganharam novos atributos referentes aos novos *widgets*. Esses novos atributos podem ser vistos nas declarações intertipos dos diagramas de colaboração com aspectos criados para mapear a relação entre *features*. O diagrama de classes do projeto parcial da *feature* `Agendamento` da Etapa 2 (Figura 4.13) destaca as novas classes, os novos atributos e métodos com o estereótipo <<new>>. As novas classes, atributos e métodos surgem da influência das outras *features* quando se projeta com os requisitos originais que contém o entrelaçamento de interesses (*features*).

O próximo passo da Etapa 2 requer projetar as diferenças entre os projetos como aspectos. A influência de cada *feature* deve ser projetada separadamente. Os diagramas de colaboração com aspectos mostrados no Capítulo 3 são utilizados. Diagramas auxiliares que esboçam a interação com o usuário mostram como é a nova seqüência de eventos, assim ajudando a elicitar atributos, métodos, adendos e conjuntos de junção. A Figura 4.14 mostra a diferença nessa interação com relação à interação original (Figura 4.8).

Para projetar, deve-se identificar, nos casos de uso, quais partes da mudança foram causadas por essa *feature*. O diagrama de interação com o usuário (Figura 4.14) ajuda nesse processo. Pode-se observar que foi necessário escolher e buscar um paciente para depois verificar os agendamentos exclusivos desse paciente. O diagrama de colaboração com aspectos é exibido na Figura 4.15.

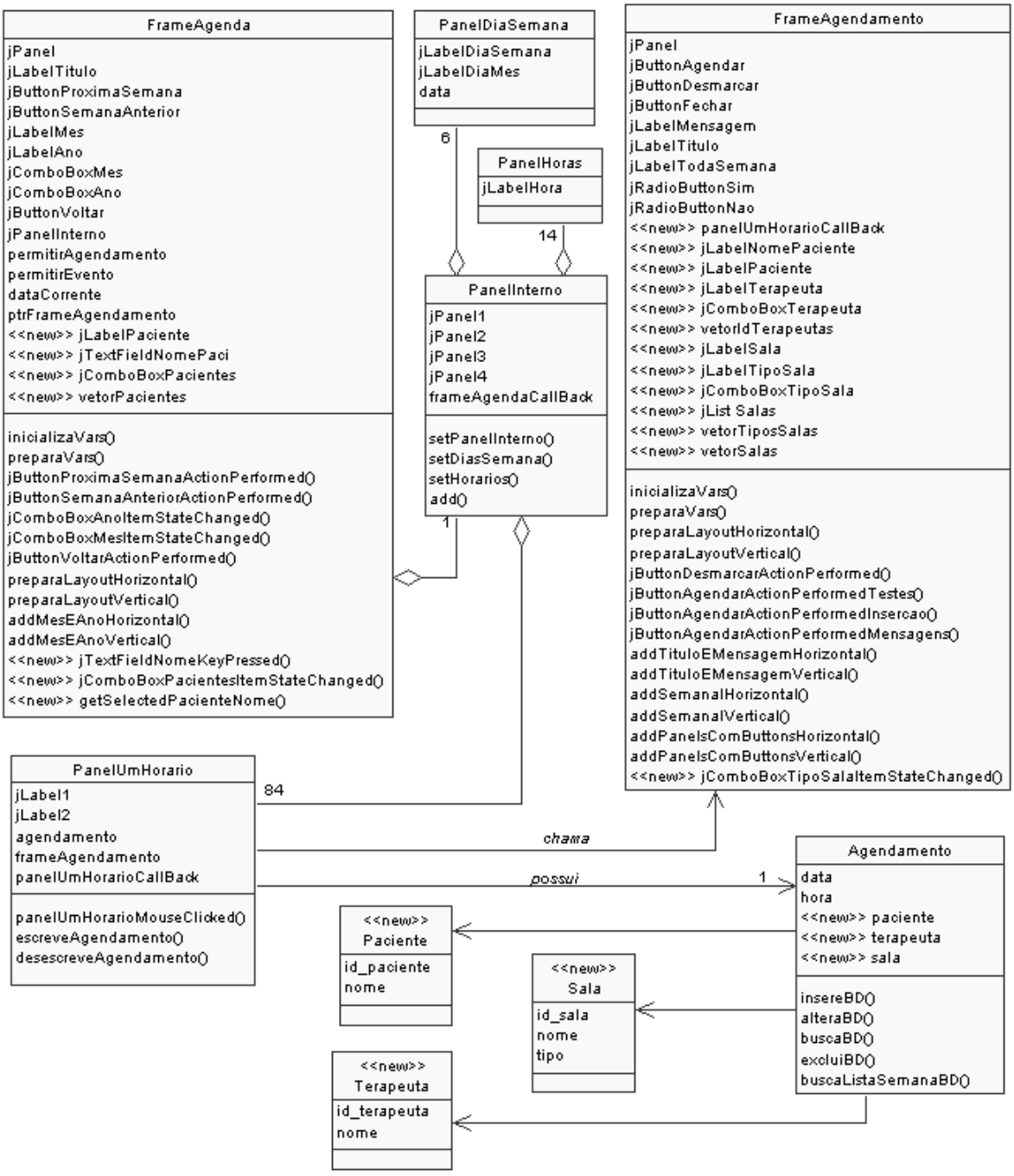


Figura 4.13: Projeto da *feature* `Agendamento` – Etapa 2

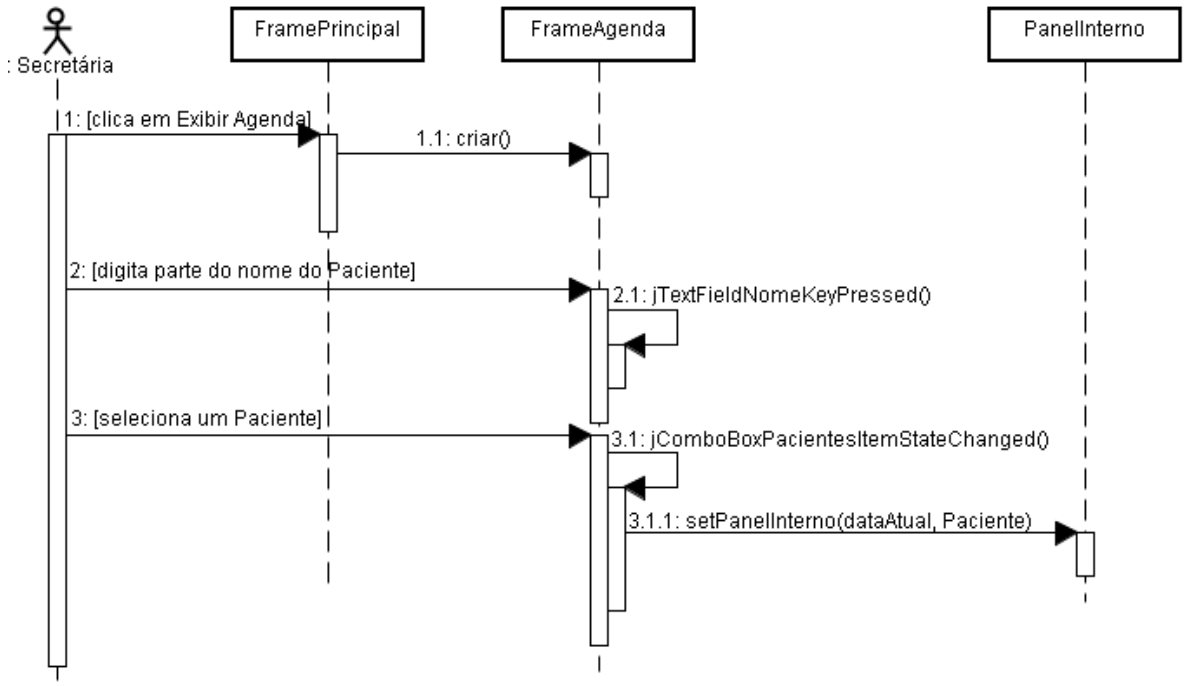


Figura 4.14: Interação do ator na exibição dos agendamentos com alterações de outras *features* – Etapa 2

Para o caso de uso Criar Agendamento, a interação com o usuário é esboçada na Figura 4.16. Utilizando-se essa figura e o caso de uso, pode-se descobrir quais partes do comportamento da funcionalidade serão alteradas pela *feature* `Paciente` e ter uma visão aproximada de como ficariam essas partes. Na interação com o usuário, há pouca influência da *feature* `Paciente`. Isso se deve ao fato de que o paciente já havia sido selecionado em passos anteriores da interação com o usuário quando foi feita a consulta dos agendamentos. Isso não significa que não haja alterações causadas por `Paciente`. Por exemplo, o sistema verifica se o paciente está ocupado no horário escolhido. Além disso, o comportamento de muitos métodos precisa ser alterado, pelo fato de a classe ter ganhado um atributo a mais. O diagrama de colaboração com os aspectos relativos à influência de `Paciente` no método `jButtonAgendarActionPerformed()` é exibida na Figura 4.17. O modo como os aspectos interceptaram os métodos decorrentes da execução do método principal foi uma decisão de projeto escolhida pelo projetista, que faz o projeto da maneira mais apropriada para cada caso.

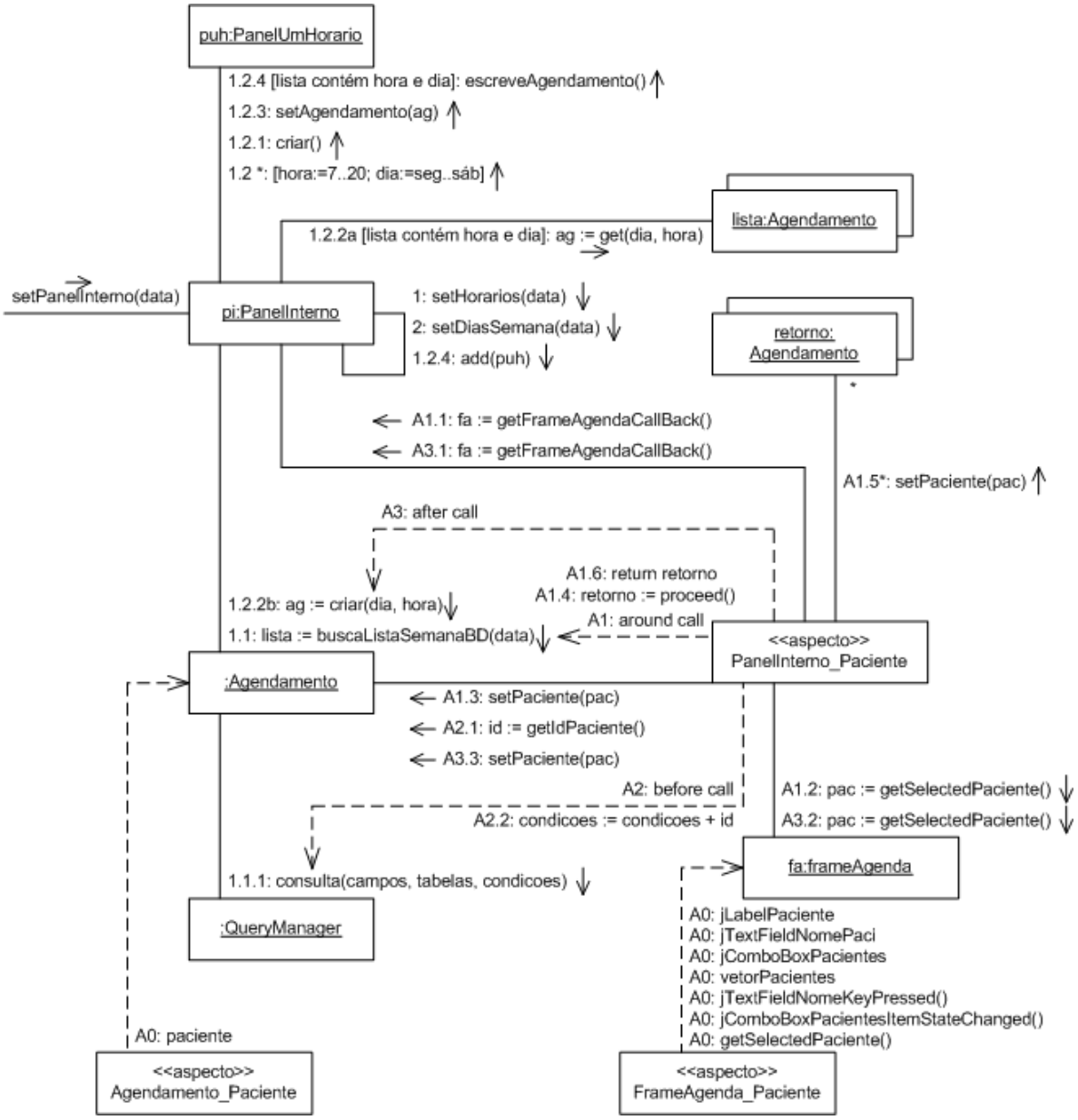


Figura 4.15: Diagrama de colaboração com aspectos – Consultar Agendamentos – Etapa 2

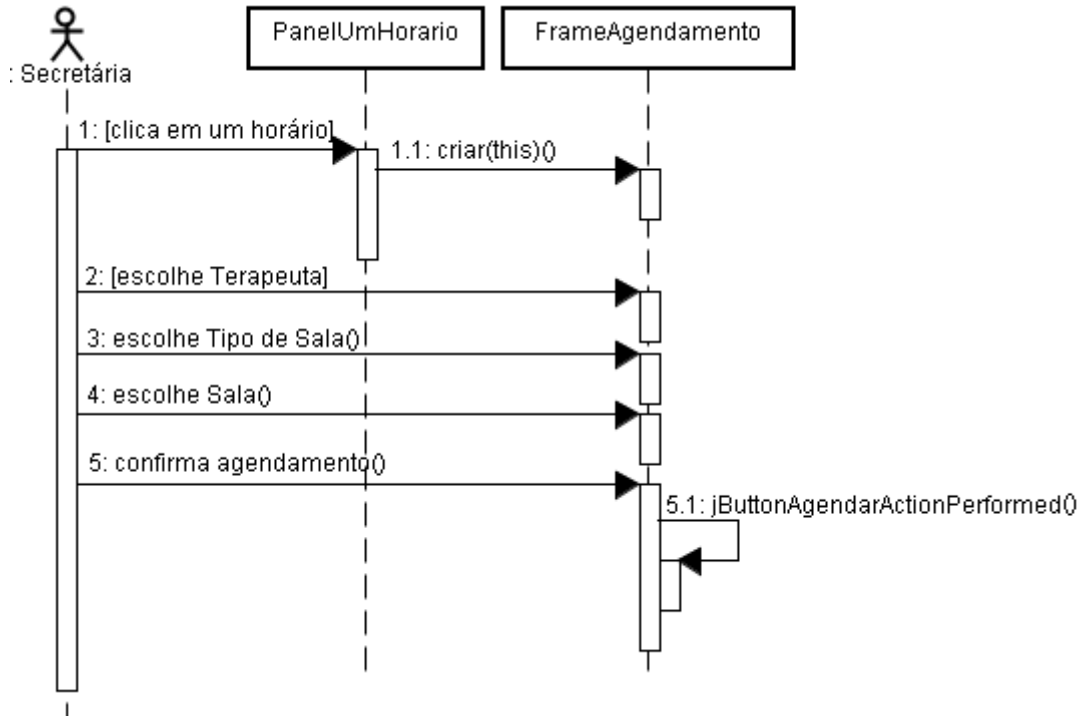


Figura 4.16: Interação do ator na criação dos agendamentos com alterações de outras *features* – Etapa 2

Figura 4.17: Diagrama de colaboração com aspectos – Criar Agendamentos – Etapa 2

Etapa 3

Para a Etapa 3, foram compostas as outras *features* em um só tema denominado "Sistema". As demais *features* eram similares à *feature* `Paciente`, no sentido de que sua funcionalidade principal era permitir a inserção, consulta, alteração e exclusão das informações referentes ao conceito da *feature*, e sendo assim o projeto de cada uma delas também foi similar, tendo duas classes de interface, uma para a busca do objeto e outra para inserir, exibir, alterar ou excluir as informações do objeto. O diagrama de classes de cada uma delas é semelhante ao diagrama da *feature* `Paciente`, visto na Figura 4.3. O diagrama de classes do tema Sistema é exibido na Figura 4.18.

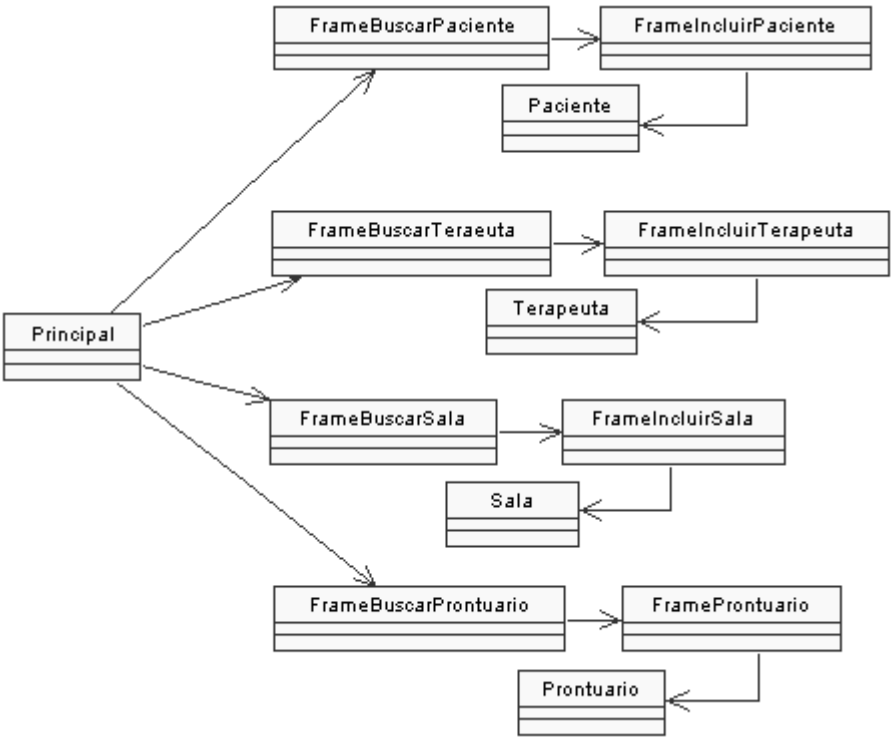


Figura 4.18: Tema Sistema – Etapa 3

Analisando o tema Sistema e o projeto da *feature* `Agendamento` (Figura 4.13), verifica-se que existem na *feature* as novas classes: `Agendamento`, `FrameAgenda`, `PanelInterno`, `PanelUmHorario`, `PanelDiaSemana`, `PanelHoras` e `FrameAgendamento`. Também existem associações com classes de outras *features*, a classe `Agendamento` associa-se com as classes `Paciente`, `Terapeuta` e `Sala`.

Conforme definido na abordagem, as novas classes foram implementadas. As associações percebidas não eram influência de `Agendamento` em outras *features*, mas sim o inverso, e por isso já haviam sido projetadas na Etapa 2.

Finalizando o desenvolvimento da *feature*

Com os artefatos desenvolvidos até então, pôde-se implementar a *feature* `Agendamento` e a influência da *feature* `Paciente`. Os artefatos foram usados da seguinte forma: o diagrama de classes da *feature* `Agendamento` – Etapa 1 (Figura 4.7) e os diagramas de colaboração, incluindo os dos exemplos dados `setPanelInterno()`– Etapa 1 (Figura 4.9) e `jButtonAgendarActionPerformed()` – Etapa 1 (Figura 4.11), foram usados para implementar a *feature* `Agendamento` utilizando programação orientada a objetos.

O diagrama de classes da *feature* `Agendamento` – Etapa 2 (Figura 4.13) e os diagramas de colaboração com aspectos da Etapa 2 (Figuras 4.15 e 4.17), foram usados para implementar os aspectos que trataram do interesse da *feature* `Paciente` na *feature* `Agendamento`, utilizando programação orientada a aspectos.

Para finalizar o projeto da *feature* `Agendamento`, foi necessário projetar as influências das demais *features* que se relacionavam com ela. As influências das *features* `Terapeuta` e `Sala` foram projetadas da mesma maneira como na *feature* `Paciente`, repetindo para cada uma a Etapa 2. O desenvolvimento das influências das *features* `Servico` e `Grupo` continuou postergado até depois que essas *features* fossem desenvolvidas.

4.6. Desenvolvimento de Features Opcionais

Para o prosseguimento do estudo de caso, foram investigadas outras possibilidades de composição de *features*. Para isso foi necessário o desenvolvimento de *features* opcionais. A *feature* `Servico` apresentou possibilidades interessantes que surgiram ao longo do seu desenvolvimento e que serão mostradas aqui.

Os requisitos para a *feature* `Servico` são os requisitos 7 e 42, vistos na Tabela 4.2, além do envolvimento com a *feature* `Agendamento` no requisito 24, que havia sido postergado para o momento de seu desenvolvimento. A partir desses requisitos obtém-se a visão de ação da Figura 4.19. Pode-se observar que as *features* `Paciente` e `Terapeuta` afetam diretamente a *feature* `Servico` pelo requisito 7. Já havia sido decidido (criação da relação *features*-requisitos) que a *feature* `Servico` é dominante nesse requisito, logo as *features* `Paciente` e `Terapeuta` irão afetar a *feature* `Servico`, decisão refletida na visão de ação aparada da Figura 4.20. O requisito 24 é dominado pela *feature* `Agendamento`, o que é coerente pelo fato de que serviço é um detalhe a mais no principal objetivo do requisito, que é o agendamento. O momento da criação da visão de ação aparada é oportuno para rever as decisões de *features* associadas com requisitos. A visão de ação aparada indica que `Servico` irá influenciar `Agendamento`. O relacionamento entre as *features* `Agendamento`, `Paciente` e `Servico` já havia sido estabelecido no desenvolvimento da *feature* `Agendamento` e foi

mantido. A presença da *feature* `Servico` pode ou não alterar essas relações, o que pode ser percebido durante o desenvolvimento.

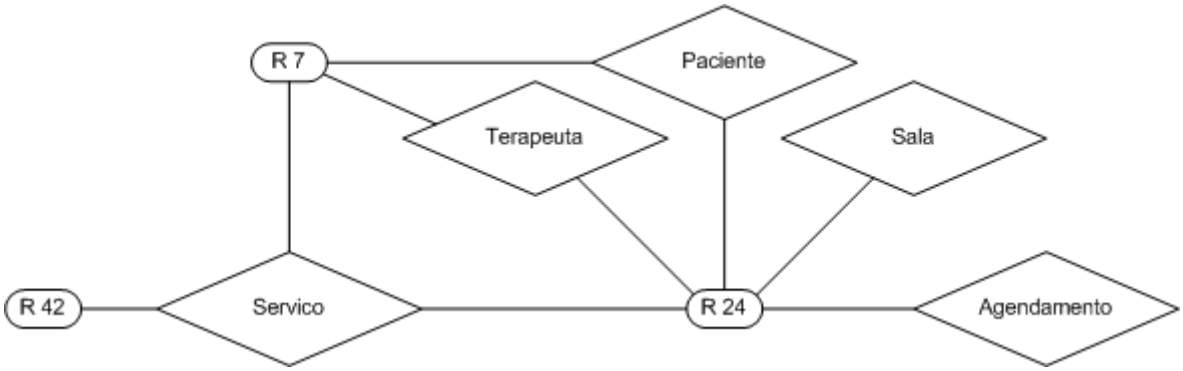


Figura 4.19: Visão de Ação *feature* `Servico`

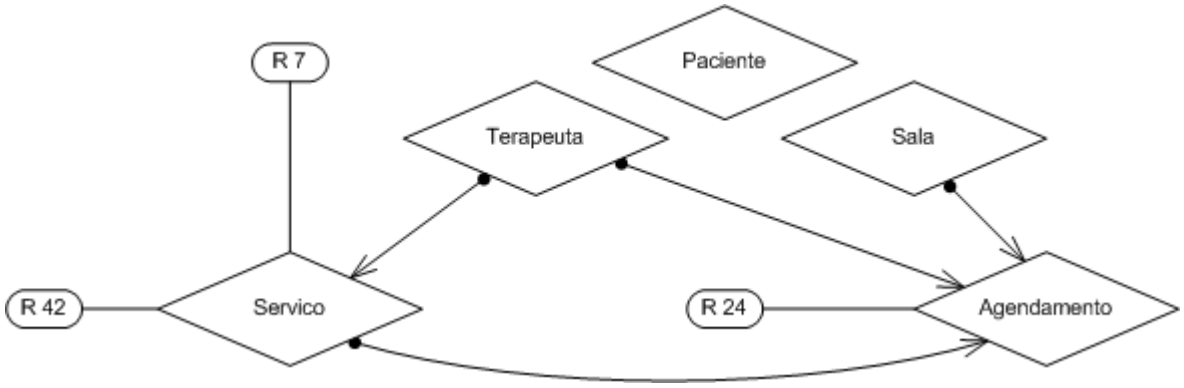


Figura 4.20: Visão de Ação Aparada *feature* `Servico`

Etapa 1

De acordo com a Etapa 1, começou-se reescrevendo os requisitos removendo influências de outras *features* que não a `Servico`:

7 O sistema deve permitir a inclusão, consulta, alteração e remoção de informações sobre o serviço recebido da clínica com os seguintes dados:

Se é encaminhado de outra instituição s/n, nome da instituição e nome do terapeuta.

Serviço (pré-cadastrado).

Horário.

Disponibilidade de horários do paciente: manhã, tarde, noite.

Situação: AM – (arquivo morto), AG (aguardo), AT (atendimento).

Conclusão: arquivado por faltas, arquivado sem relatório, desistência, encerramento do caso (Alta), falecimento, final do processo, mudança de endereço, não localizado, vencimento da triagem (aguardou muito tempo).

42 O sistema deve permitir a inclusão, consulta, alteração e remoção de informações sobre os Serviços da clínica com as seguintes informações: Nome e descrição.

Com o auxílio do material do domínio e consultas aos psicólogos do GestorPsi, ficou-se entendido o modo como as clínicas trabalham com seus serviços. Serviço é uma descrição genérica das informações referentes a algum tipo de prestação de serviço comum ou trabalho desenvolvido normalmente em clínicas de psicologia, tais como Terapia, Orientação Vocacional ou Tratamento de Dependência Química.

Em uma clínica, um serviço pode ser oferecido de uma ou mais maneiras, com diferenças nas faixas etárias, grupos restritos de pessoas, com diferentes modalidades de atendimento, tratamento ou outras especificações. Quando um serviço oferecido é prestado a um paciente, existem informações específicas a respeito desse atendimento que devem ser registradas. Assim, tendo o entendimento global da *feature*, foram criados três conceitos pertinentes a ela: `Servico`, `Servico Oferecido` e `Servico Recebido`.

Prosseguindo com a Etapa 1, a partir dos requisitos foram criados os casos de uso presentes no diagrama de casos de uso da Figura 4.21. A palavra Gerenciamento nos casos de uso significa incluir, consultar, alterar e remover, o que na verdade resulta em quatro casos de uso cada. Para poupar detalhes desnecessários, apenas os artefatos mais ilustrativos serão exibidos aqui, sendo eles relativos ao desenvolvimento do caso de uso Gerenciar Serviço Recebido.

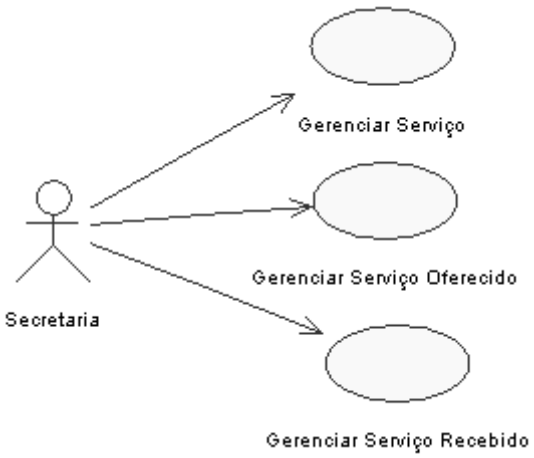


Figura 4.21: Casos de Uso *feature* `Servico` – Etapa 1

Caso de Uso: Incluir Serviço Recebido – Etapa 1

Atores: Secretaria.

Propósito: Permitir que se crie um novo atendimento.

Visão Geral: O funcionário da secretaria inicia um atendimento com um dos serviços que a clínica oferece.

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de iniciar atendimentos.	2. Exibe o formulário de atendimentos em branco.
3. Escolhe um dos Serviços Oferecidos.	
4. Preenche as outras informações necessárias, como pasta, encaminhamento, horários e situação e clica em Iniciar.	5. Registra as informações e dá o atendimento como iniciado.

O desenvolvimento desses casos de uso criou as classes e as funcionalidades básicas da *feature* `Servico`, que podem ser vistas no diagrama de classes da Figura 4.22. A *feature* também utiliza a estrutura de duas classes de interface para cada classe de lógica de aplicação, uma para busca dos objetos e uma para inclusão, consulta, alteração e exclusão.

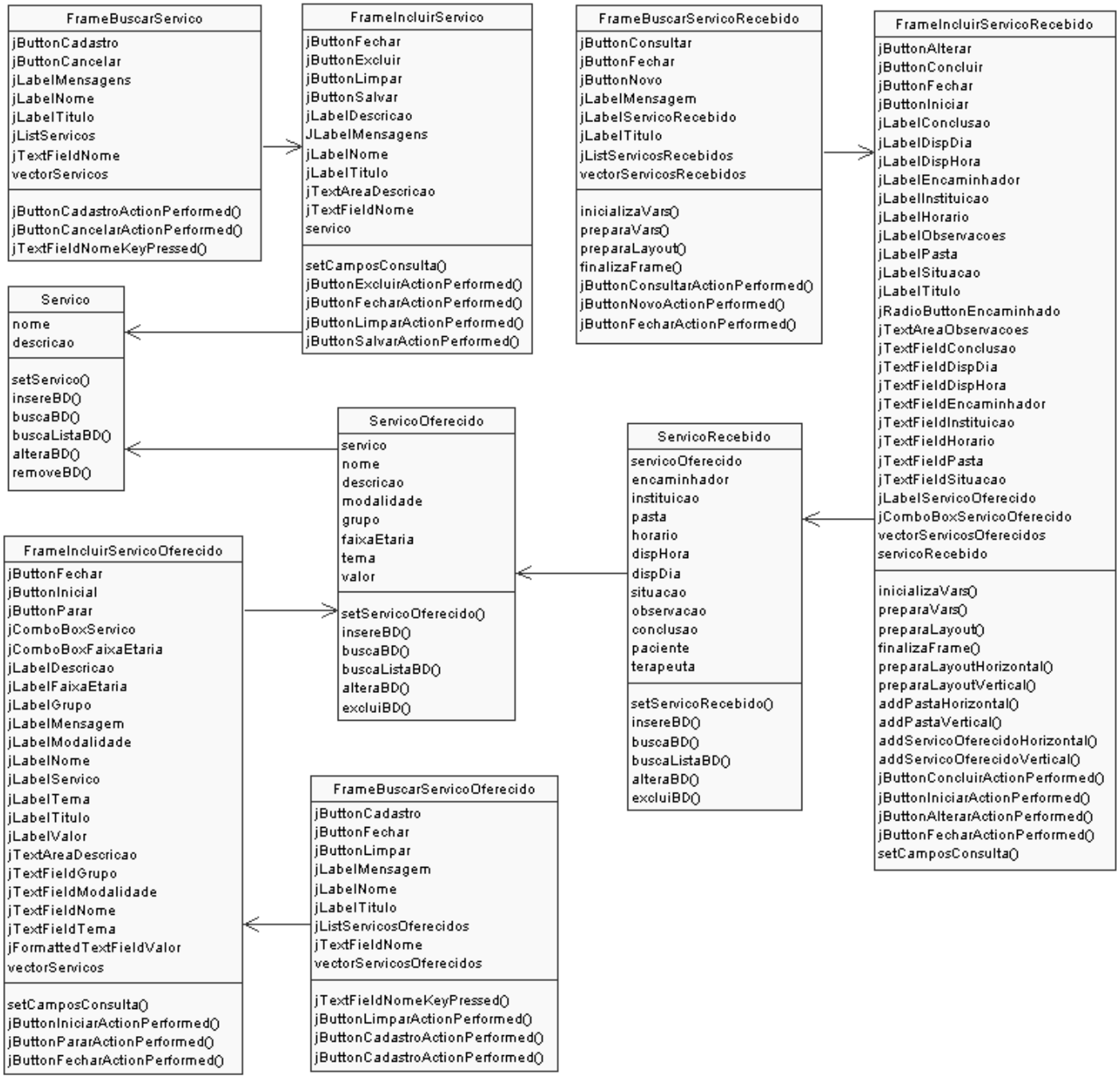


Figura 4.22: Diagrama de classes da *feature* `Servico` – Etapa 1

Etapa 2

Para a Etapa 2, foram projetadas as influências das outras *features*. Pela visão de ação aparada da Figura 4.20 pode-se ver que as *features* `Paciente` e `Terapeuta` influenciam na *feature* `Servico`. Para continuar o exemplo iniciado, será mostrada a influência da *feature* `Paciente`. O caso de uso Incluir Serviço Recebido é desenvolvido com os requisitos originais da Tabela 4.2.

Caso de Uso: Incluir Serviço Recebido – Etapa 2

Atores: Secretaria.

Propósito: Permitir que se crie um novo atendimento.

Visão Geral: O funcionário da secretaria inicia um atendimento para um paciente com um dos serviços que a clínica oferece.

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de iniciar atendimentos.	2. Exibe o formulário de atendimentos em branco.
3. Escolhe um dos Serviços Oferecidos.	
4. Escolhe um Paciente.	
5. Escolhe um Terapeuta.	
6. Preenche as outras informações necessárias, como pasta, encaminhamento, horários e situação e clica em Iniciar.	7. Registra as informações e dá o atendimento como iniciado.

O caso ocorrido é típico e esperado da evolução da linha de produtos incremental. Os objetos de lógica de aplicação como `ServicoRecebido` ganham novos atributos referentes a uma combinação com outras *features*, e assim os objetos que compõem a camada de interface desses objetos também precisam acompanhar as evoluções. Para os objetos de lógica de aplicação, é feita a introdução dos novos atributos em seus vetores `campos` e `valores`. Para os objetos de interface os novos atributos referentes aos novos *widgets* são introduzidos com declarações intertipos, e a preparação desses objetos para a preparação da interface é feita seguindo as técnicas explicadas na seção 3.5.2.2.

O diagrama de colaboração da Figura 4.23 representa a utilização das técnicas para a interface `FrameIncluirServicoRecebido`. A seqüência de mensagens

original é composta pelas mensagens que não são iniciadas pela letra "A". São cumpridas as quatro atividades definidas para os *widgets* de interface: inicialização, preparação de valores, montagem do leiaute e finalização do leiaute (mensagens de 1 a 4). As mensagens iniciadas pela letra "A" mostram as interceptações dos aspectos e a seqüência de eventos dos respectivos adendos.

A0: introdução dos novos *widgets* necessários por Paciente

A1: faz a inicialização dos novos *widgets*

A2: prepara os novos *widgets* com valores e conteúdos

A3: posiciona os novos *widgets* horizontalmente no leiaute

A4: posiciona os novos *widgets* verticalmente no leiaute

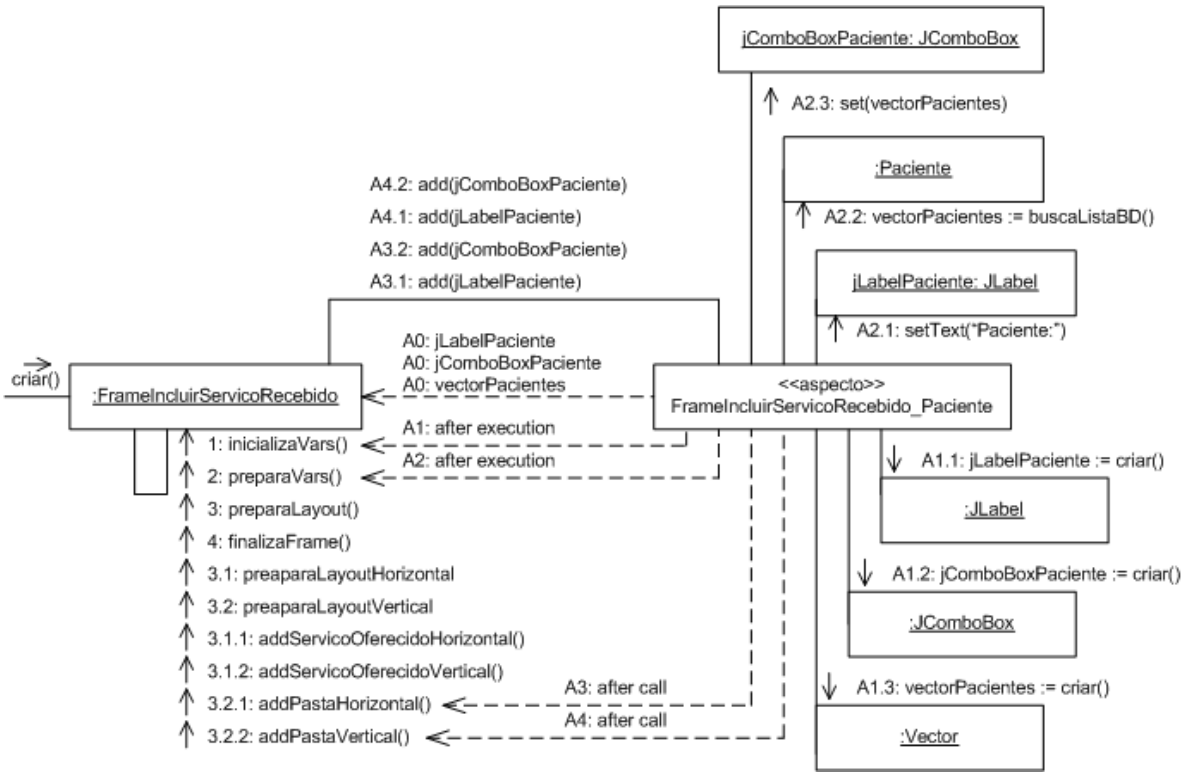


Figura 4.23: Diagrama de colaboração com aspectos – Construtor da interface `FrameIncluirServicoRecebido` – Etapa 2

O diagrama de classes da Figura 4.24 apresenta as alterações sofridas pela *feature* `Servico`. Os novos atributos e métodos são destacados com o estereótipo <<new>>.

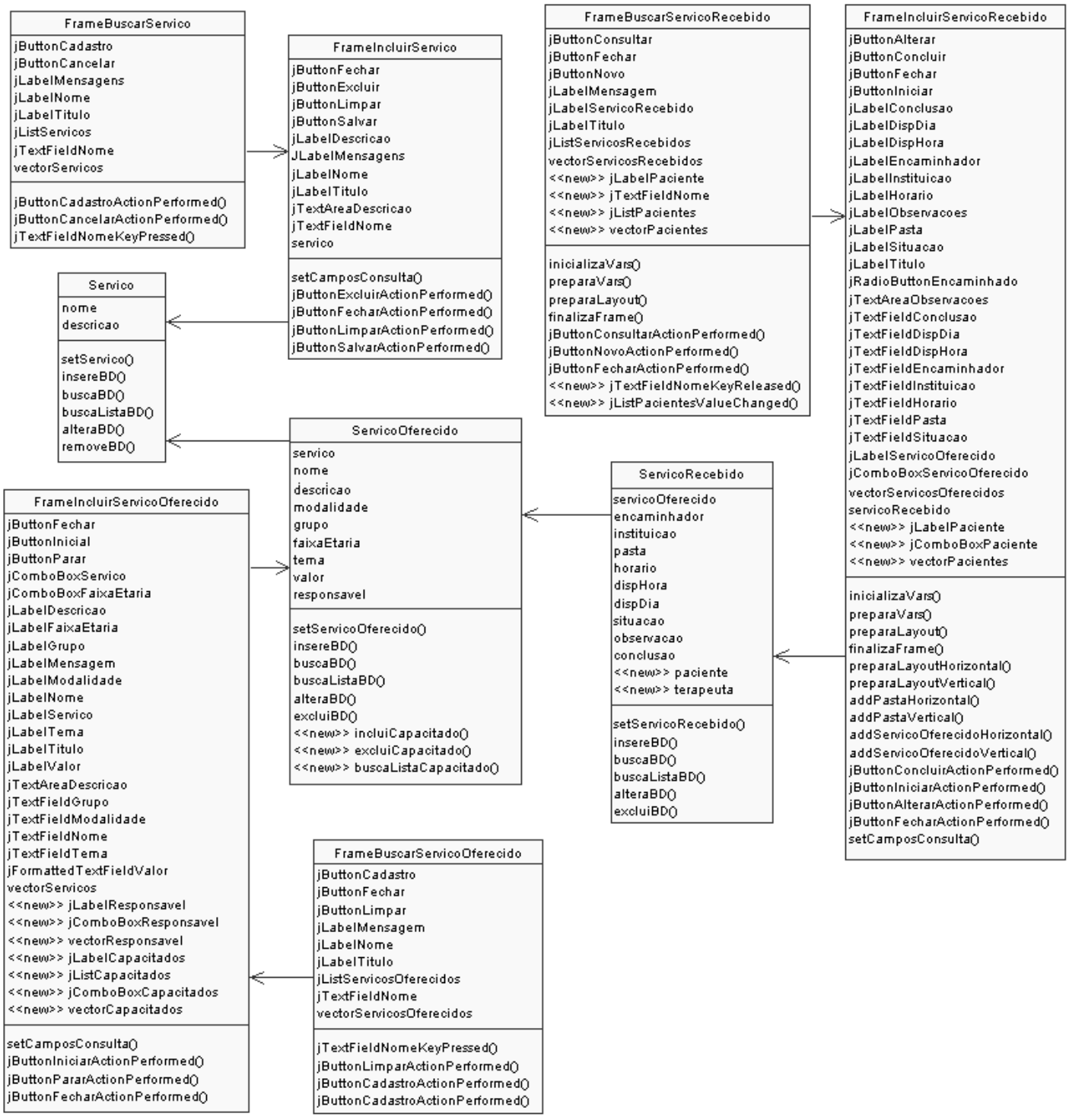


Figura 4.24: Diagrama de classes da *feature* `Servico` – Etapa 2

Etapa 3

O projeto da *feature* `Servico` deve ser comparado com o projeto do restante do sistema para ver se há interseção. Pelo diagrama de classes da Figura 4.24 não há, já que todas as classes presentes são classes novas que não existiam no sistema. Portanto, esta parte é implementada com classes OO.

Resta projetar a influência da *feature* `Servico` na *feature* `Agendamento`, evidenciada pela visão de ação aparada da Figura 4.20, que se deu no requisito 7. Esse requisito deu origem aos casos de uso Consultar Agendamentos e Criar Agendamentos. O procedimento é então desenvolver novamente os casos de uso considerando o requisito da maneira 100% original e perceber as influências da *feature* `Servico` a partir do que surgir de diferente nesse projeto.

Caso de Uso: Consultar Agendamento – (reescrito. Etapa 3 – *feature* Servico)

Atores: Secretária.

Propósito: Permitir que os agendamentos marcados sejam consultados.

Visão Geral: A secretária, ou pessoa encarregada verifica os agendamentos de um determinado período.

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de consultar agendamentos.	
2. Escolhe um período.	3. Exibe a grade horária em branco.
4. Digita algumas letras do nome do paciente.	5. Preenche uma lista com os pacientes que tem nomes que começam com o que o usuário digitou.
6. Escolhe um paciente da lista.	7. Preenche uma lista com os serviços que o paciente está recebendo.
8. Escolhe um dos serviços que o paciente está recebendo.	9. Exibe a grade horária com dias e horas preenchida com agendamentos do período daquele serviço do paciente.

Caso de Uso: Criar Agendamento – (reescrito. Etapa 3 – *feature* Servico)

Atores: Secretária.

Propósito: Permitir que um atendimento seja agendado.

Visão Geral: A secretária ou pessoa encarregada marca um agendamento.

Referência Cruzada: Consultar Agendamento

Seqüência de Eventos:

Ação do Ator	Resposta do Sistema
1. Seleciona a opção de consultar agendamentos.	
2. Escolhe um período.	3. Exibe a grade horária com dias e horas.
4. Digita algumas letras do nome do paciente.	5. Preenche uma lista com os pacientes que tem nomes que começam com o que o usuário digitou.
6. Escolhe um paciente da lista.	7. Preenche uma lista com os serviços que o paciente está recebendo.
8. Escolhe um dos serviços que o paciente está recebendo.	9. Exibe a grade horária com dias e horas preenchida com agendamentos do período daquele serviço do paciente.
10. Seleciona uma data e hora da grade.	11. Exibe uma tela para confirmar o agendamento. Pergunta se é toda semana.
	12. Na mesma tela exibe lista de tipos de salas.
13. Escolhe se toda semana ou não.	
14. Escolhe um tipo de sala.	15. Exibe lista com as salas do tipo escolhida.
16. Escolhe uma sala.	
17. Confirma o agendamento.	18. Verifica se o paciente está ocupado na/s data/s do agendamento.
	19. Verifica se o terapeuta está ocupado na/s data/s do agendamento.
	20. Verifica se a sala está ocupada na/s data/s do agendamento.
	21. Anota que existe um agendamento na data e hora indicada.

A partir da presença da *feature* `Servico`, o funcionamento da *feature* `Agendamento` é diferente. Isso porque o paciente passou a se relacionar com a clínica por um serviço que ele está recebendo, e o terapeuta se relaciona com o paciente por um serviço que ele está prestando. Os agendamentos serão referentes ao serviço sendo prestado. No modelo conceitual da Figura 4.25a pode-se observar como era a estrutura do `Agendamento` e na Figura 4.25b como ela passa a ser.

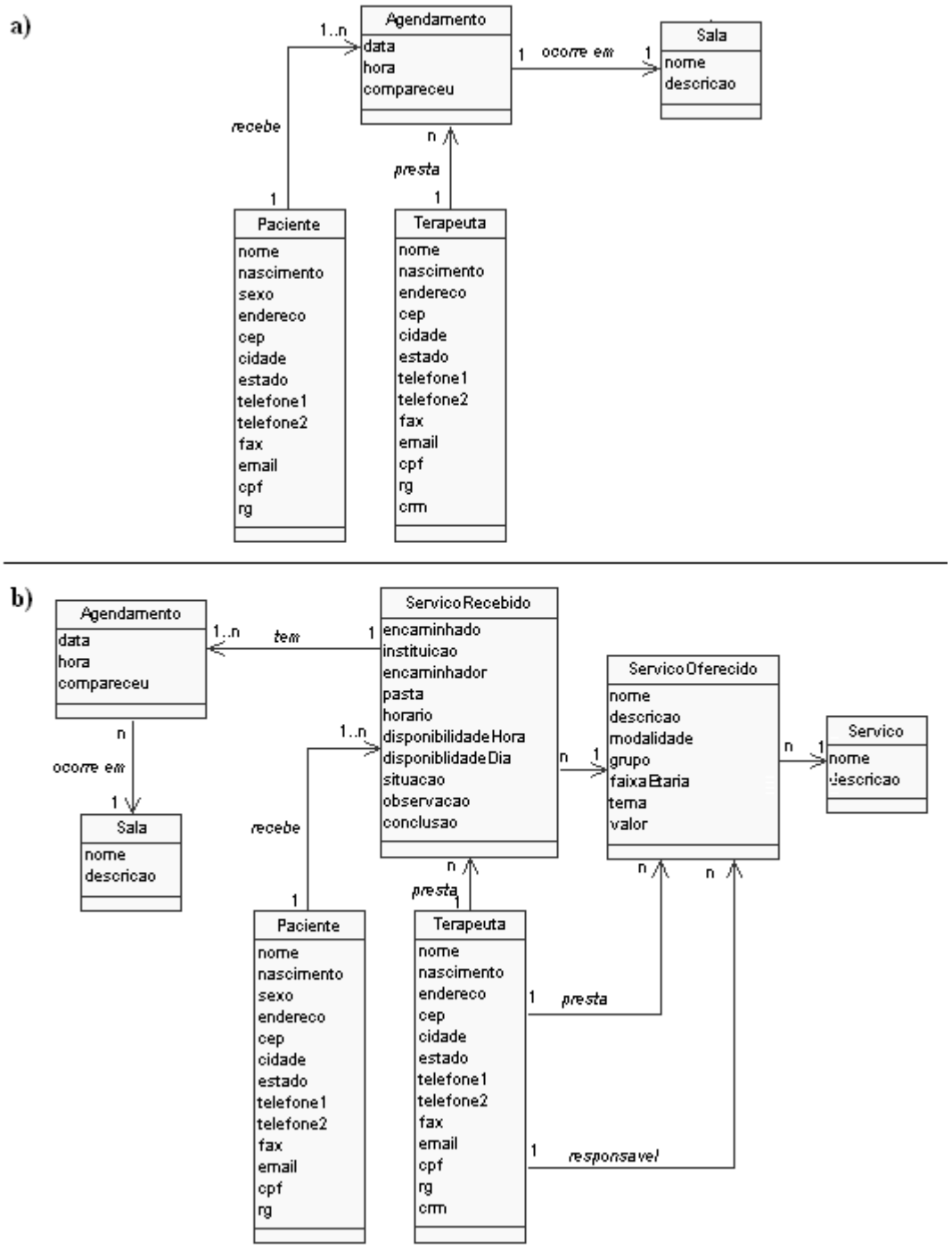


Figura 4.25: Modelo Conceitual *feature* `Servico`

A partir dos casos de uso, existem duas conclusões. A primeira é a associação evidente entre os objetos `Servico` e `Agendamento`. `Agendamento` terá uma referência a um `Servico` que poderá ser um `id` ou um objeto completo.

A segunda é que se pode ver que o objeto `Agendamento` não tem mais associações diretas com os objetos `Paciente` e `Terapeuta`. Isso não significa que as *features* `Paciente` e `Terapeuta` não influenciem a *feature* `Agendamento`.

Indiretamente, pelo serviço, ainda é preciso saber se o paciente ou terapeuta estão ocupados naquela data e hora do agendamento, pois o paciente pode estar recebendo mais de um serviço e o terapeuta pode estar prestando mais de um serviço.

A segunda conclusão representa um caso interessante para o estudo: dadas duas *features*, existia um projeto e implementação da influência de uma na outra. Uma terceira *feature* surge e altera o funcionamento do sistema de tal maneira que a influência entre as duas primeiras é alterada e deve ser reprojetada.

No exemplo utilizado, a *feature* `Paciente` influenciava a *feature* `Agendamento` de tal maneira que o objeto `Agendamento` continha um objeto `Paciente`. Com a presença da *feature* `Servico`, o objeto `Agendamento` não conterá mais um `Paciente`, mas certos comportamentos relativos ao `Paciente` ainda estarão presentes na *feature* `Agendamento`.

Este caso explica, por exemplo, a terceira parte do nome dos pacotes que contém os objetos de implementação da influência de duas *features* (ver Seção 3.5.2.3). Uma mesma *feature* recebe influência de uma outra de duas formas distintas, que são os casos em que não há uma terceira *feature* ou em que há uma terceira *feature*. No primeiro caso, a terceira parte do nome do pacote é o nome da *feature* que influi na primeira, pois foi ela mesma quem causou a influência, e o nome do pacote fica `Agendamento_Paciente_Paciente`. No segundo caso, o nome do pacote fica `Agendamento_Paciente_Servico`, pois foi `Servico` que determinou o modo como `Paciente` influenciaria em `Agendamento`.

O projeto da influência de `Servico` em `Agendamento` pode ser visto nos diagramas de colaboração com aspectos nas Figuras 4.26 e 4.27. Na seqüência, o projeto da influência de `Paciente` em `Agendamento` com a presença de `Servico` é feito também baseado nos casos de uso reescritos da Etapa 3. No caso de uso Consultar Agendamento, a *feature* `Paciente` não intercepta o método `setPanelInterno()` em momento algum. A influência se dá antes – um paciente é escolhido na interface `FrameAgenda` para que depois se possa escolher um dos serviços que ele está recebendo. A partir disso os agendamentos da grade horária são buscados com influência do `Servico`. A pouca influência de `Paciente` no agendamento se deve ao fato de que nesse contexto, `Paciente` já havia influenciado `Servico`. No caso de uso Criar `Agendamento`, existe a necessidade de checar se o paciente está ocupado, e nesse momento há uma interceptação do método `Agendamento.insereBD()`.

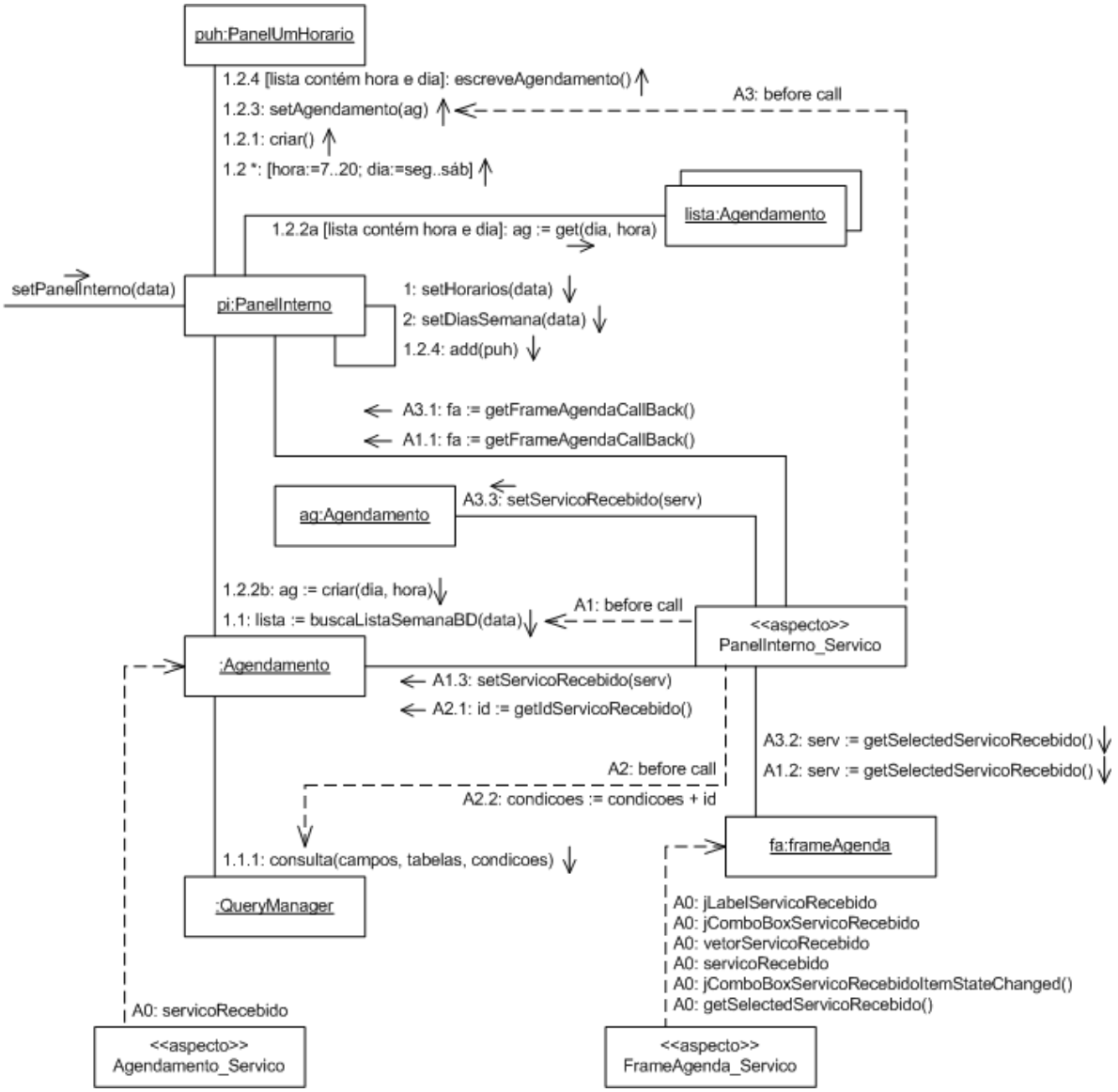


Figura 4.26: Diagrama de colaboração com aspectos – Influência de `Servico` em `Agendamento` - Consultar Agendamentos – Etapa 3

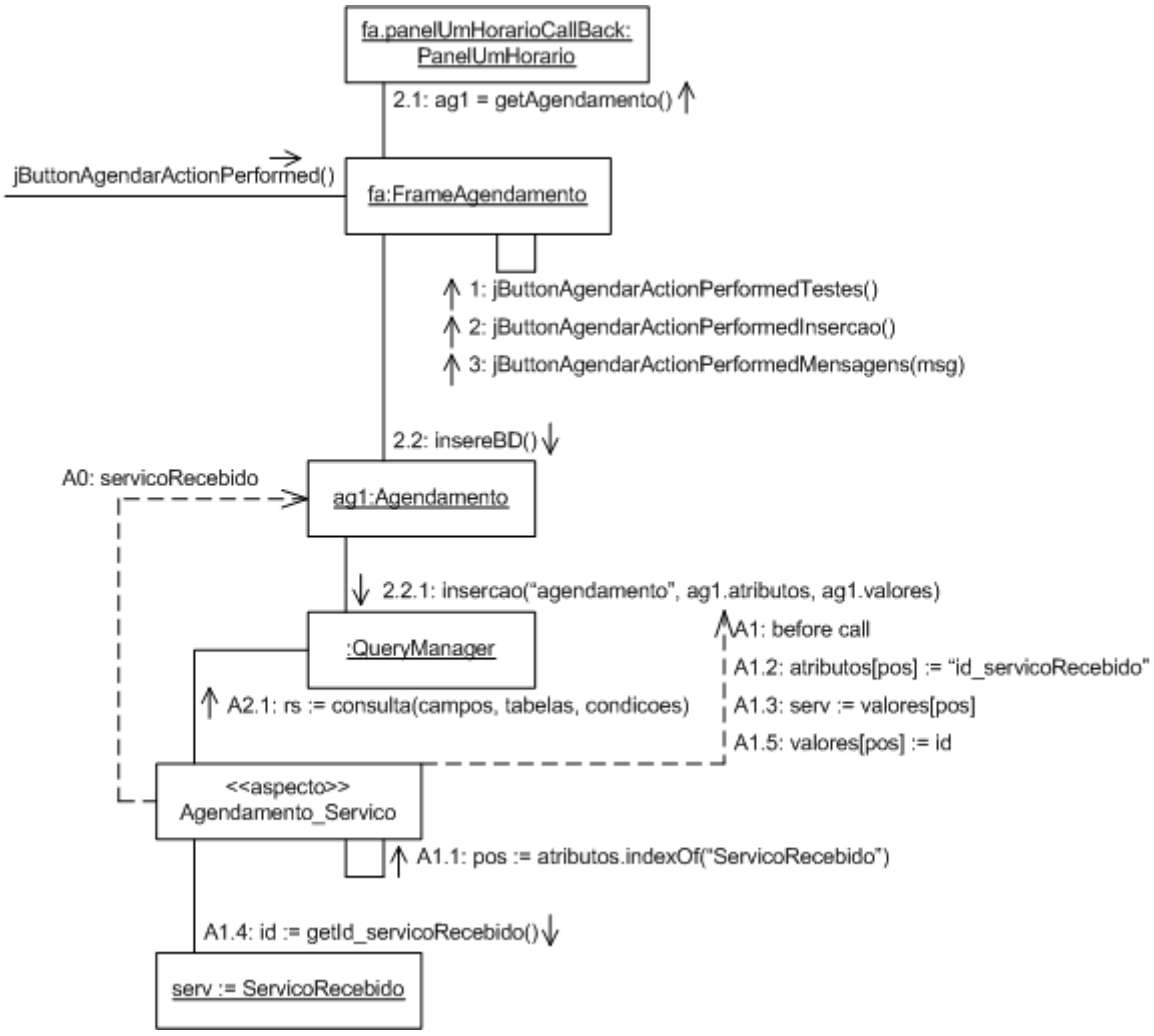


Figura 4.27: Diagrama de colaboração com aspectos – Influência de `Servico` em `Agendamento` - Criar Agendamentos – Etapa 3

Assim, com os diagramas de colaboração pôde-se implementar a influências da *feature* `Paciente` na feature `Agendamento` e concluir o projeto da *feature* `Agendamento`.

Além das *features*, foi necessário implementar a interface principal, que não está associada a nenhuma *feature*, mas a partir dela se acessam as funcionalidades das *features*. Foi necessário implementar uma camada de persistência para acesso à base de dados que é utilizada por diversas *features*.

Usando-se as *features*, pode ocorrer que mais de uma delas adicione *widgets* em uma interface, ou comportamentos em métodos existentes. Assim, é necessário controlar a ordem em que cada aspecto age. Para isso deve haver um aspecto que controle as precedências dos outros aspectos. Esse aspecto de precedência precisa ser preparado de acordo com as *features* utilizadas.

4.7. Composição e Instanciação de Produtos

As seguintes *features* foram desenvolvidas para o sistema GestorPsi: `Paciente`, `Prontuario`, `Terapeuta`, `Sala`, `Agendamento` e `Servico`. A relação *features*-pacotes obtida pode ser vista na Tabela 4.4:

Tabela 4.4: Relação *Features* Pacotes para a Linha de Produtos de Clínicas de Psicologia

Núm. *Feature*	**Nome *Feature***	**Requer *Feature***	**Pacotes a Incluir (p.i.)**	**Pacotes a Excluir (p.e.)**
1	Paciente		Paciente	
2	Prontuario	1	Prontuario	
3	Terapeuta		Terapeuta	
4	Sala		Sala	
5	Agendamento		agendamento	
6	Servico		Serviço	
5 e 1	Agendamento Paciente		agendamento_paciente_agendamento	
5 e 3	Agendamento Terapeuta		agendamento_terapeuta_agendamento	
5 e 4	Agendamento Sala		agendamento_sala_agendamento	
6 e 1	Servico Paciente		servico_paciente_servico	
6 e 3	Servico Terapeuta		servico_terapeuta_servico	
6 e 5	Servico Agendamento		agendamento_servico_servico	
6, 5 e 1	Servico Agendamento Paciente		agendamento_paciente_servico	agendamento_paciente_agendamento
6, 5 e 3	Servico Agendamento Terapeuta		agendamento_terapeuta_servico	agendamento_terapeuta_agendamento

A relação esclarece quais os pacotes a serem incluídos nos produtos em duas ou mais determinadas *features* estiverem presentes, mostrando o nome de todas as *features* no campo Nome Feature.

Foram testadas duas instanciações de produtos. Primeiro com as *features* `Paciente`, `Prontuario`, `Terapeuta`, `Sala` e `Agendamento`. Depois acrescentou-se a *feature* `Servico` ao produto.

O processo de combinação foi feito utilizando o combinador AJC do AspectJ, que estava instalado como *plug-in* no ambiente de desenvolvimento Eclipse (Eclipse, 2006).

Esse ambiente de desenvolvimento com o *plug-in* de AspectJ permitia fazer a composição dos pacotes escolhidos e gerar a aplicação resultante. No Apêndice A podem ser vistas algumas telas dos produtos gerados.

4.8. Considerações Finais

Neste capítulo foi apresentado um estudo de caso que serviu como experimentação para o desenvolvimento da abordagem incremental para criação de linhas de produtos.

A partir de uma idéia geral inicial de usar a abordagem Tema para projetar *features* de uma linha de produtos e implementá-las utilizando aspectos, iniciou-se o desenvolvimento da linha de produtos. A idéia inicial foi então experimentada na prática, além das demais idéias que surgiram ao longo do desenvolvimento da linha de produtos.

Durante o decorrer do estudo de caso, surgiu a necessidade de várias atividades para possibilitar que se pudesse ter um processo que conduzisse passo a passo da análise ao projeto e implementação. Cada atividade apresentou peculiaridades e foi refinada, substituída ou abandonada para se encaixar e compor a abordagem.

Além de servir como experiência para o desenvolvimento, o estudo de caso também serviu para validar grande parte do que é proposto pela abordagem, já que só eram incorporadas definitivamente à abordagem as práticas que se mostravam eficientes ou adequadas.

A seguir, resumem-se os eventos ocorridos durante o estudo de caso que tiveram influência nas definições da abordagem.

Inicialmente o desenvolvimento procurou seguir a o processo definido na abordagem Tema para fazer a análise e projetar as *features* da linha de produtos como temas, e com isso investigar se a abordagem Tema seria adequada para análise e projeto de linhas de produtos orientadas a aspectos.

Como não haviam os requisitos definidos, foram utilizadas técnicas para análise de domínio para fazer o levantamento dos requisitos. As técnicas para análise de domínio utilizadas, trouxeram um conhecimento abrangente do domínio, o qual pode ser aproveitado não apenas na criação dos requisitos, mas de um modo mais abrangente no contexto da criação de linhas de produtos.

As técnicas para análise de domínio foram conciliadas com técnicas para desenvolvimento de linhas de produtos, como a abordagem FODA ou o método ESPLEP. As técnicas para desenvolvimento de linhas de produtos têm necessidade e dão ênfase à análise de domínio. Por isso foi dado ênfase no processo de domínio da abordagem. A análise de domínio é essencial e intrínseca ao desenvolvimento de linhas de produtos.

Essas metodologias ditavam um processo diferente do da abordagem Tema e tinham práticas, conceitos e artefatos diferentes. Elas consideravam o conceito de *features* como componentes a serem armazenados em um repositório para reúso.

Existem vários papéis nesses processos, o analista de domínio, o projetista de software, desenvolvedor de aplicações que é usuário das *features* do repositório entre outros. Esses papéis ditam atividades específicas em fases específicas do desenvolvimento de linhas de produtos, os quais foram em parte incorporados ao processo.

Foi feito o levantamento dos requisitos e com isso era possível utilizar a abordagem Tema. A parte de análise da abordagem Tema orientava a separação em temas de todo comportamento entrecortante. Seguindo a abordagem à risca, obtinha-se temas de granularidade pequena e que não eram compatíveis com as intenções da linha de produtos de obter *features* inteiras projetadas como temas para permitir sua composição para a instanciação de sistemas. Além disso, os temas entrecortantes não mapeavam diretamente para os aspectos do AspectJ.

As técnicas de linhas de produtos permitiram a separação e mesmo a divisão de artefatos por *features*. A partir disso, foi utilizada análise e projeto OO para desenvolver cada *feature* separada, no contexto do processo unificado, desenvolvendo casos de uso, diagramas de colaboração e diagramas de classes.

Tendo o projeto de *features* separadas, percebeu-se que elas poderiam ser compostas utilizando a composição de temas base da abordagem tema. Esse processo de composição foi ligeiramente alterado para que ele também identificasse os pontos de interceptação de uma *feature* na outra e o que ocorria nesses pontos. Isso então era mapeado diretamente para os conjuntos de junções e adendos.

Por fim, durante a implementação foram reconhecidas maneiras otimizadas de implementar as classes que recebiam interceptações e os aspectos que interceptavam, e assim foram definidas como práticas recomendadas.

5. Conclusões

5.1. Contribuições deste Trabalho

Este trabalho apresentou uma abordagem incremental para desenvolvimento de linhas de produtos orientadas a aspectos (Pacios et al., 2006). A abordagem é constituída de técnicas para APOA que conduzem o desenvolvedor desde a análise do domínio, possibilitando a criação de um projeto para software orientado a aspectos. As técnicas de análise de domínio empregadas fazem com que não seja necessário ter os requisitos para o software de antemão, pois eles são obtidos pela própria análise de domínio.

A abordagem é constituída de uma seqüência de passos curtos, bem definidos e reprodutíveis. Ela é dividida em três fases: (1) Análise de Domínio, (2) Desenvolvimento da Base e (3) Desenvolvimento dos Produtos. Cada uma possui atividades e artefatos bem definidos.

A abordagem considera as *features* da linha de produtos como os interesses principais da aplicação, sendo as *features* isoladas, encapsuladas e projetadas com a orientação a aspectos.

As técnicas de DSOA beneficiaram a criação das *features* da linha de produtos. Com o encapsulamento proporcionado pela orientação a aspectos, elas tornam-se mais coesas, mais fáceis de se combinarem e de serem reutilizadas. A própria natureza da programação orientada a aspectos foi responsável por facilitar a combinação das *features*. As técnicas para APOA foram combinadas com técnicas de APOO para conseguir otimizar o projeto com separação de interesses.

O trabalho também fornece uma contribuição com o estudo da utilização de aspectos para interesses funcionais. As *features* da linha de produtos são criadas de maneira a serem compostas por um ou mais interesses funcionais. A abordagem apresentada para desenvolvimento de *features* pode então ser aplicada para o desenvolvimento de interesses funcionais.

Apesar da abordagem ser própria para a criação de linhas de produtos, muitas das técnicas apresentadas são aplicáveis ao desenvolvimento de software tradicional. Várias das atividades presentes ao longo da abordagem não são específicas de desenvolvimento de linhas de produtos e, portanto, são comuns ao desenvolvimento de

software. Assim como ocorre com certos processos de desenvolvimento de software, o desenvolvedor é livre para ter o discernimento de utilizar as atividades e artefatos que forem mais apropriados para a situação de projeto.

A abordagem integrou APOA com implementação OA, ou seja, a abordagem fornece o embasamento para que se crie todo o projeto da linha de produtos e fornece também técnicas de implementação OA (diretrizes) que podem ser usadas em benefício de um bom código com alta coesão e baixo acoplamento, tornando-o assim mais fácil de reutilizar. As diretrizes de implementação OA também podem ser utilizadas independentemente da utilização das demais atividades da abordagem, porém, é mais garantido um bom código tendo um bom projeto.

Isso também significa que a abordagem trata o problema dos aspectos nos estágios iniciais do desenvolvimento. Por exemplo, os requisitos já são agrupados por *features*. Esse problema tem sido bastante discutido atualmente na comunidade de software (Pearce e Noble, 2006; Apel et al., 2006; Griswold et al., 2006; Baniassad et al., 2006), busca desenvolver processos para análise e projeto de software orientado a aspectos. Atualmente existem muitas pesquisas para solucionar problemas específicos das várias fases do desenvolvimento.

Para o desenvolvimento da abordagem foram consideradas várias técnicas recentes e pesquisas com práticas para APOA e desenvolvimento de LPs, além de contribuições do autor. Dentre as técnicas para APOA, a abordagem Tema se destacou por ter fornecido uma grande contribuição, pelo fato de ser uma das técnicas mais elaboradas com pesquisas bastante atuais.

Como resultado, a abordagem resultante permite o desenvolvimento incremental de linha de produtos. Tradicionalmente, linhas de produtos precisam ser especificadas e projetadas por inteiro antes do desenvolvimento. Isso porque os encaixes (*hot spots*) para as *features* precisam ser projetados conhecendo-se de antemão todas as *features* que virão a ser utilizadas na linha de produtos. No caso da abordagem incremental aqui proposta, a natureza dos aspectos permitiu (pelo menos em parte) que o novo código passasse a interagir com o código existente, sem a necessidade de que o código existente precisasse conhecer o código das *features* a serem desenvolvidas. Houve apenas a necessidade de uma preparação genérica no código existente e que é simples. Essa preparação é pertinente às diretrizes de implementação. Com isso, possibilitou-se reduzir uma das grandes desvantagens de linhas de produtos que é o alto custo inicial, e também foi facilitada a integração de novas *features*.

5.2. Limitações

No trabalho realizado não foram empregadas todas as possibilidades de utilização de aspectos, pois não era o objetivo deste estudo. Por exemplo, dentro do projeto das *features*, há entrelaçamento e espalhamento, porque o trabalho desenvolvido preocupou-se apenas com a aplicação dos aspectos em um nível mais elevado que é o da *feature* inteira. Porém, o uso da abordagem não descarta a separação de interesses em granularidades menores, até mesmo dentro do projeto de cada *feature*.

Além disso, é importante considerar também que o estudo de caso realizado desenvolveu apenas uma parte de uma linha de produtos de tamanho reduzido. Assim, somente uma porcentagem pequena das possibilidades foram cobertas pelo estudo. Existem muitas outras possibilidades de tipos de requisitos, tipos de sistemas de informação, domínios, dificuldades, peculiaridades, situações no desenvolvimento entre outras coisas que podem causar a necessidade de aprimoramento na abordagem ou até possam inviabilizar sua utilização. Faltou averiguar o desempenho da abordagem quando a quantidade de *features* desenvolvida torna-se muito elevada e também quando a complexidade da linha de produtos é alta por algum motivo.

As diretrizes de implementação foram desenvolvidas considerando o uso da arquitetura de três camadas, as linguagens Java e AspectJ. Isso não significa que as diretrizes não sejam válidas fora desse escopo, mas seriam necessários mais estudos para verificar como as técnicas poderiam ser adaptadas a outras condições de desenvolvimento.

Apesar da intenção de facilitar o desenvolvimento de linhas de produtos orientadas a aspectos, as técnicas e diretrizes propostas neste trabalho podem ocasionar no aumento na complexidade do software e dificultar a compreensão do mesmo. Essa possibilidade precisaria ser averiguada com estudos empíricos.

Além disso, todo o desenvolvimento da abordagem foi feito tendo em consideração um tipo de sistema, o de sistemas de informação. Acredita-se que as técnicas para projeto, desenvolvimento e integração de interesses funcionais possam ser aplicadas a outros tipos de sistemas, mas isso precisaria ser confirmado com estudos específicos. Sistemas web, por exemplo, podem apresentar particularidades que alterem fatores do uso da abordagem, como por exemplo, as diretrizes de implementação.

5.3. Trabalhos Futuros

O presente trabalho envolveu muitos conceitos e tecnologias diferentes. Muitos deles são por si campos de pesquisas. Este trabalho serve como base e ponto de partida para vários tópicos que podem ser investigados.

Toda a abordagem pode, e é justamente uma das intenções do trabalho, ser aprimorada com técnicas específicas, para evoluir rumo a um processo completo. A abordagem é constituída de muitas técnicas e atividades que podem, isoladamente ou em conjunto, ser aperfeiçoadas para melhoria geral do processo. Um exemplo é o modo como são feitos os conjuntos de junção para alterar o comportamento dos métodos. Atualmente, eles são escritos especificamente para os pontos de junção. Poderiam ser padronizados para que conjuntos de junção mais específicos herdassem deles. Existem trabalhos relacionados sendo feitos atualmente nessa área (Pearce e Nobel, 2006). Outro exemplo são as linguagens de modelagens e extensões de linguagens de modelagens sendo pesquisadas para OA, como, por exemplo, os trabalhos de Ziadi et al. (2004), Aldawud et al. (2003) e Rodrigues e Masiero (2005).

O desenvolvimento de padrões de projeto OA pode ser feito a partir das diretrizes apresentadas, com o objetivo de melhorar o reúso. Por outro lado, o uso de padrões de projeto já existentes, tanto OA como OO, dentro do projeto das *features*, certamente seria benéfico ao projeto.

Os ganhos reais no desenvolvimento de linhas de produtos com a utilização da abordagem proposta teriam de ser investigados e mensurados por meio de experimentos para verificação de seus resultados.

Para ajudar na geração automática de produtos, existe a possibilidade de construção de geradores de sistemas, devido à possibilidade de mapear os artefatos de projeto para os artefatos de código. Atualmente, um trabalho de mestrado está sendo desenvolvido nesse sentido. A ferramenta Captor é um gerador de aplicações configurável (Shimabukuro Junior et al., 2006), que gera aplicações a partir de especificações em alto nível persistidas em um repositório. Ela está sendo estendida para poder incorporar código orientado a aspectos (Pereira Junior, 2006). Assim, essa ferramenta seria beneficiada por um código orientado a aspectos que utilizasse as diretrizes deste trabalho e que fosse bem projetado com as técnicas para análise e projeto orientadas a aspectos aqui apresentadas.

A abordagem desenvolvida no presente trabalho foi apresentada para a comunidade de psicologia (Braga et. al., 2006), a qual se mostrou interessada principalmente na informatização de seus processos, utilizando sistemas de informação, recurso para o qual há demanda devido à carência de sistemas nessa área. Isso dá margem ao desenvolvimento de novas pesquisas relativas a geradores de aplicação e linhas de produtos para essa área.

Referências Bibliográficas

Aldawud, O.; Elrad T.; Bader, A. UML Profile for Aspect-Oriented Software Development. In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003.

Alves, V. R. Implementing Software Product Line Variability. Proposta de Tese de Doutorado apresentada ao Centro de Informática da Universidade Federal de Pernambuco, Pernambuco – BA, 2005.

Apel, S.; Leich, T.; Saake, G. Aspectual Mixin Layers: Aspects and Features. In Concert. In: Proceedings of International Conference on Software Engineering. (2006)

Aragon, C. R. Processo de Desenvolvimento de uma Linha de Produtos para Sistemas de Gestão de Bibliotecas. Dissertação de Mestrado apresentada ao Departamento de Computação e Estatística da Fundação Universidade Federal do Mato Grosso do Sul. Campo Grande – MS, 2006.

Arango, G. Domain Engineering for Software Reuse. Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1988.

Arkin, A. Business Process Modeling Language (BPML), Version 1.0. BPML.org, 2002.

AspectJ. The AspectJ Project. Disponível para acesso na URL: http://eclipse.org/aspectj/, em 10/11/2006.

Baniassad, E. L. A.; Clarke, S. Finding Aspects in Requirements with Theme/Doc. Workshop on Early Aspects, held as part of the International Conference on Aspect-oriented Software Development (AOSD) 2004.

Baniassad, E. L. A., Clements, P., Araújo, J., Moreira, A., Rashid, A.; Tekinerdogan, B. (2006). Discovering Early Aspects. In IEEE Software, vol. 23, nº 1, pp. 61-70.

Bayer, J., Flege, O.; Gacek, C. Creating Product Line Architectures. Proceedings of the Third International Workshop on Software Architectures for Product Families (IWSAPF-3), March, 2000.

Bergey, J.; Cohen, S.; Fisher, M.; Campbell, G.; Jones L.; Krut, R.; Northrop, L.; O'Brien, W.; Smith, D.; Soule, A. Fourth DoD Product Line Practice Workshop Report (CMU/SEI-2001-TR-017, ADA399205). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001.

Bosch, J. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach. Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, 2000.

Braga, R. T. V.; Lima, R. C.; Pacios, S. F. Abordagem para desenvolvimento de linha de produtos de software na área de psicologia. In: III Psicoinfo Seminário Brasileiro de

Psicologia e Informática, 2006, São Paulo. Caderno Temático do III Psicoinfo e II Jornada do NPPI, 2006.

Pereira Junior, C. A. F. Composição e Geração de Aplicações usando Aspectos, projeto de mestrado com bolsa Fapesp proc no 06/54467-3, sob orientação da Profa Dra Rosana T. V. Braga, 2006.

Clarke, S. Aspect-Oriented Design with Theme/UML. UPGRADE, the European Journal for the Informatics Professional. Vol. V, No. 2: 14-20, April, 2004.

Clarke, S.; Baniassad, E. L. A. Aspect Oriented Analysis and Design. Addison-Wesley Professional, ISBN 0321246748, April, 2005.

Czarnecki, K.; Eisenecker, U. W. Generative Programming: Methods, Tools, and Applications. Boston: Addison-Wesley, 2000.

Dijkstra, E. W. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.

Eclipse Project. Disponível para acesso em: http://www.eclipse.org/eclipse/, em 02/12/2006.

Elrad, T.; Filman, R. E.; Bader, A. Aspect Oriented Programming. Communications of the ACM, 44(10), October 2001a.

Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H. Discussing Aspects of AOP. Communications of the ACM 44(10), pp. 33–38, October 2001b.

Gomaa, H. Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, 2004.

Griswold, W. G.; Shonle, M.; Sullivan, K.; Song, Y.; Tewari, N.; Cai, Y.; Rajan, H. Modular Software Design with Crosscutting Interfaces. IEEE Software, vol. 23, no. 1, pp 51-60. 2006.

Griss, M. L. Implementing Product-Line Features with Component Reuse, in Proceedings of 6th International Conference on Software Reuse, Springer-Verlag, Vienna, Austria, June, 2000.

Griss, M. L. "Product-line architectures". In G. T. Heineman and W. T. Councill, editors, Component-Based Software Engineering, chapter 22, pages 405–420. Addison-Wesley, 2001.

IEEE Std. 830. IEEE Guide to Software Requirement Specification. The Institute of Electrical and Electronics Engineers. New York, 1984.

Jacobson, I.; Booch, G.; Rumbaugh, J. The Unified Process. IEEE Software (May/June 1999).

John, I.; Muthig, D. Product Line Modeling with Generic Use Cases. In Proceedings of the Second Software Product Line Conference, August 2002.

Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software, Engineering Institute, Carnegie Mellon University, 1990.

Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.- M.; Irwin, J. Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 1997, pp 220–242.

Larman, C. Utilizando UML e padrões: uma introdução à análise e ao projeto Orientada a Objeto. Porto Alegre: Bookman, 2000.

Lee, K.; Kang, K.; Lee, J. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Gacek, C. (eds.): Software Reuse: Methods, Techniques, and Tools. Lecture Notes in Computer Science, Vol. 2319. Springer-Verlag, Berlin Heidelberg (2002) 62-77.

McCain, R. Reusable Software Component Construction: A Product-Oriented Paradigm. In Proceedings of the 5th AIAA/ACM/NASA/IEEE Computers in Aerospace Conference, Long Beach, CA, pp:125-135, October 21-23, 1985.

Neighbors, J. Software Construction Using Components. Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine, 1981.

Ossher, H.; Tarr, P. Using Multi-Dimensional Separation of Concerns to (Re)Shape Evolving Software. CACM 44(10): 43-50, October 2001.

Pacios, S. F.; Masiero, P. C.; Braga, R. T. V. Guidelines for Using Aspects to Evolve Product Lines. In: III Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, 2006, Florianópolis. Proceedings do WASP - In: SBES 2006, 2006a.

Pearce, D. J.; Noble, J. (2006) Relationship aspects. In: Proc. of the 5th international conference on Aspect-oriented software development table of contents, p. 75-86.

Prieto-Diaz, R. Domain Analysis for Reusability. In Proceedings of COMPSAC 87:The Eleventh Annual International Computer Software & Applications Conference, pages 23-29. IEEE Computer Society, Washington, DC, October, 1987.

Prieto-Díaz, R. Domain Analysis: an Introduction. In ACM SIGSOFT Software Engineering Notes, eel. 15, no. 2 (47-54), April 1990.

Rational, C. Unified Modeling Language. Disponíıvel na URL: http://www.rational.com/uml/references, 2000.

Rodrigues, A. G.; Masiero, P. C. Converting Theme/UML into a Fully Symmetric Design Modelling Approach. In: II Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, 2005, Uberlândia. Anais do II Workshop Brasileiro de

Desenvolvimento de Software Orientado a Aspectos. São Carlos: Valter V. de Camargo/Instituto de Ciências Matemáticas e de Computação-USP, 2005. v. 2. p. 43-50.

SEI/CMU. A Framework for Software Product Line Practice. Versão 4.2. Software Engineering Institute, Carnegie Mellon University. Disponível para visualização na URL: http://www.sei.cmu.edu/plp/framework.html#framework_toc, em 12/11/2006.

SEI. Three Tier Software Architectures. Disponível para acesso em: http://www.sei.cmu.edu/str/descriptions/threetier_body.html, em Agosto de 2006.

Shimabukuro Junior, E. K.; Masiero, P. C.; Braga, R. T. V. Captor: Um Gerador de Aplicações Configurável. In: XIII SESSÃO DE FERRAMENTAS DO SBES, 2006, Florianópolis - SC. Anais da XIII Sessão de Ferramentas do SBES. Florianópolis: SBC, 2006. v. 1, p. 121-128.

Silva, R. P.; Price, R. T. Suporte ao desenvolvimento e uso de componentes flexíveis. In: Proceedings of XIII Simpósio Brasileiro de Engenharia de Software. pp 13-28. Florianópolis, SC – Outubro de 1999.

Travassos, G. H.; Gurov, D. TABA Workstation: Towards a Product Line for the Developing of Software Engineering Environments. Publicações Técnicas do Projeto TABA, RT-21/02. Programa de Engenharia de Sistemas e Computação. COPPE/UFRJ, 2002.

Turine, M.A.S.; Masiero, P.C. Especificação de Requisitos: uma introdução. Relatório Técnico, n.39, ICMC - USP, São Carlos - SP, 1996.

Van Deursen, A.; Klint P. Domain-specific language design requires feature descriptions. In Journal of Computing. Informatics. Tech. 10, 1, pp. 1–17. 2002.

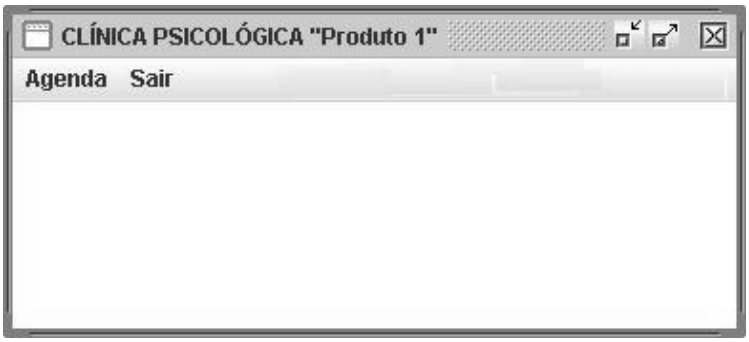
Ziadi, T.; Hélouët, L.; Jézéquel, J. M. Towards a UML Profile for Software Product Lines. In: Software Product-Family Engineering: 5th International Workshop. Lecture Notes in Computer Science, Vol. 3014. Springer-Verlag, Berlin Heidelberg (2004) 129-139.

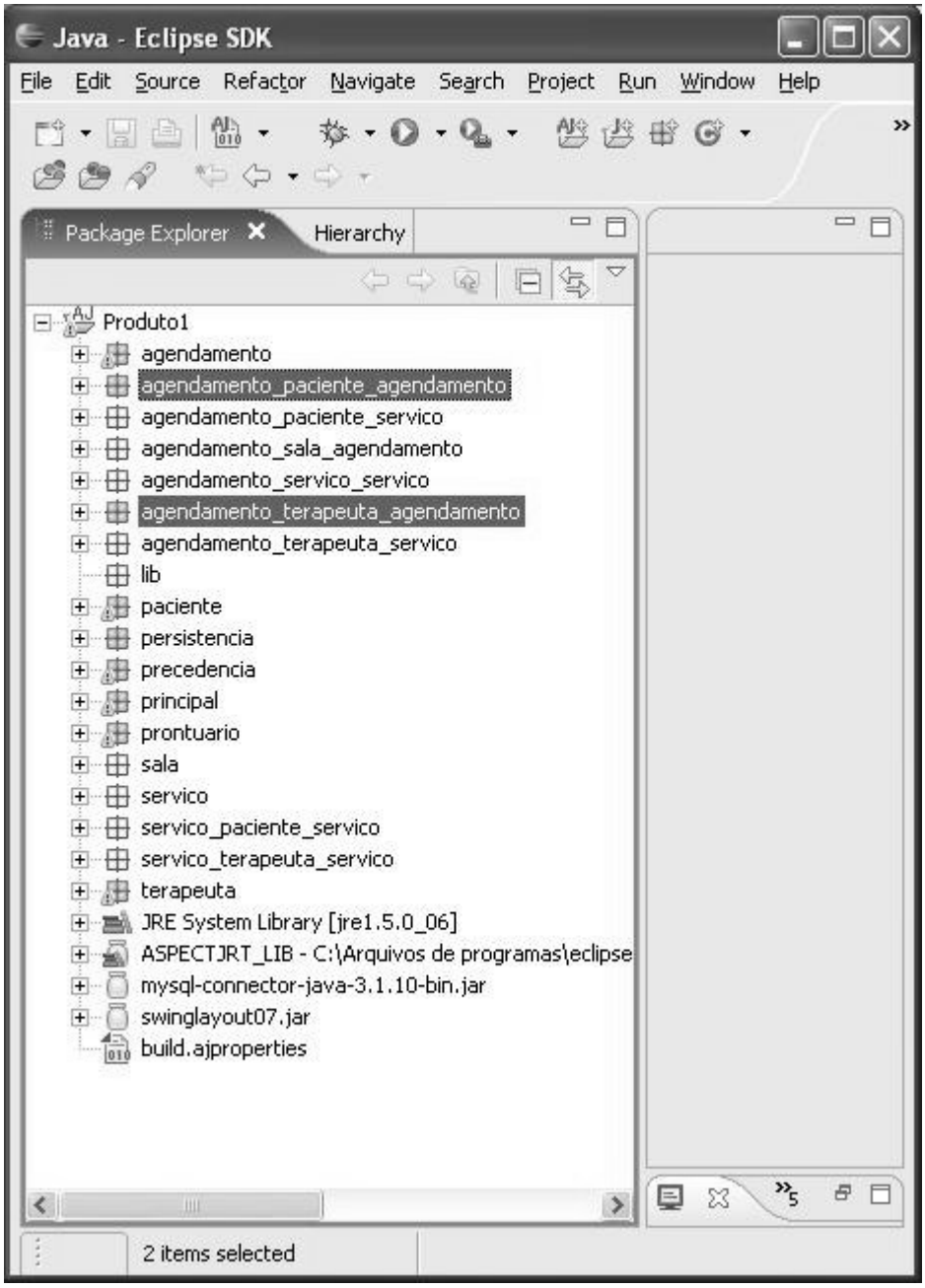
Yoder, J.W.; Balaguer, F.; Johnson, R. Architecture and Design of Adaptive Object Models. SIGPLAN Not. 36, pages 50-60. 2001

Weiss, D.; Lai, C. T. R. Software Product-Line Engineering. A Family-Based Software Development Process, Addison-Wesley, 1999.

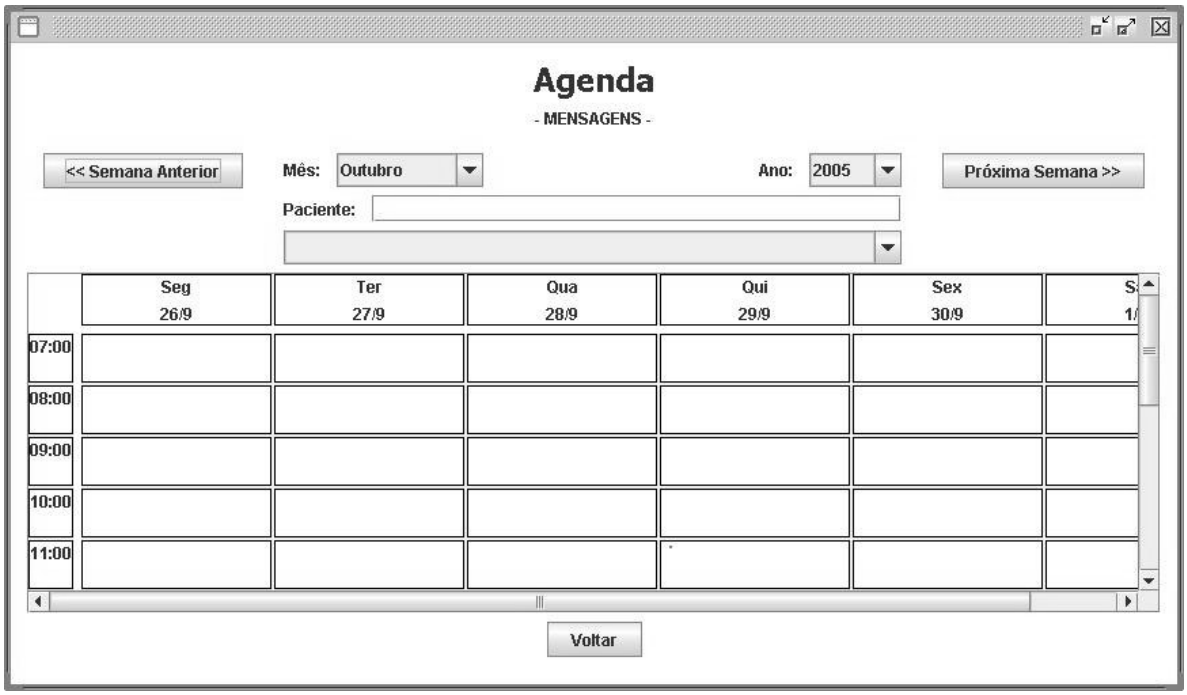
Apêndice A. Screenshots da instanciação de produtos

Este apêndice apresenta *screenshots* da instanciação de produto. A instanciação foi feita no ambiente eclipse configurado com o *plug-in* do AspectJ. Na primeira Figura pode-se ver o ambiente, e é exibido um projeto "Projeto 1" que é formado pelos pacotes pertinentes às *features*. Os produtos são montados com uma facilidade do ambiente que permite escolher os pacotes para compilação. Os demais *screenshots* são referentes às telas do sistema e outras escolhas de pacotes no projeto "Projeto 1".

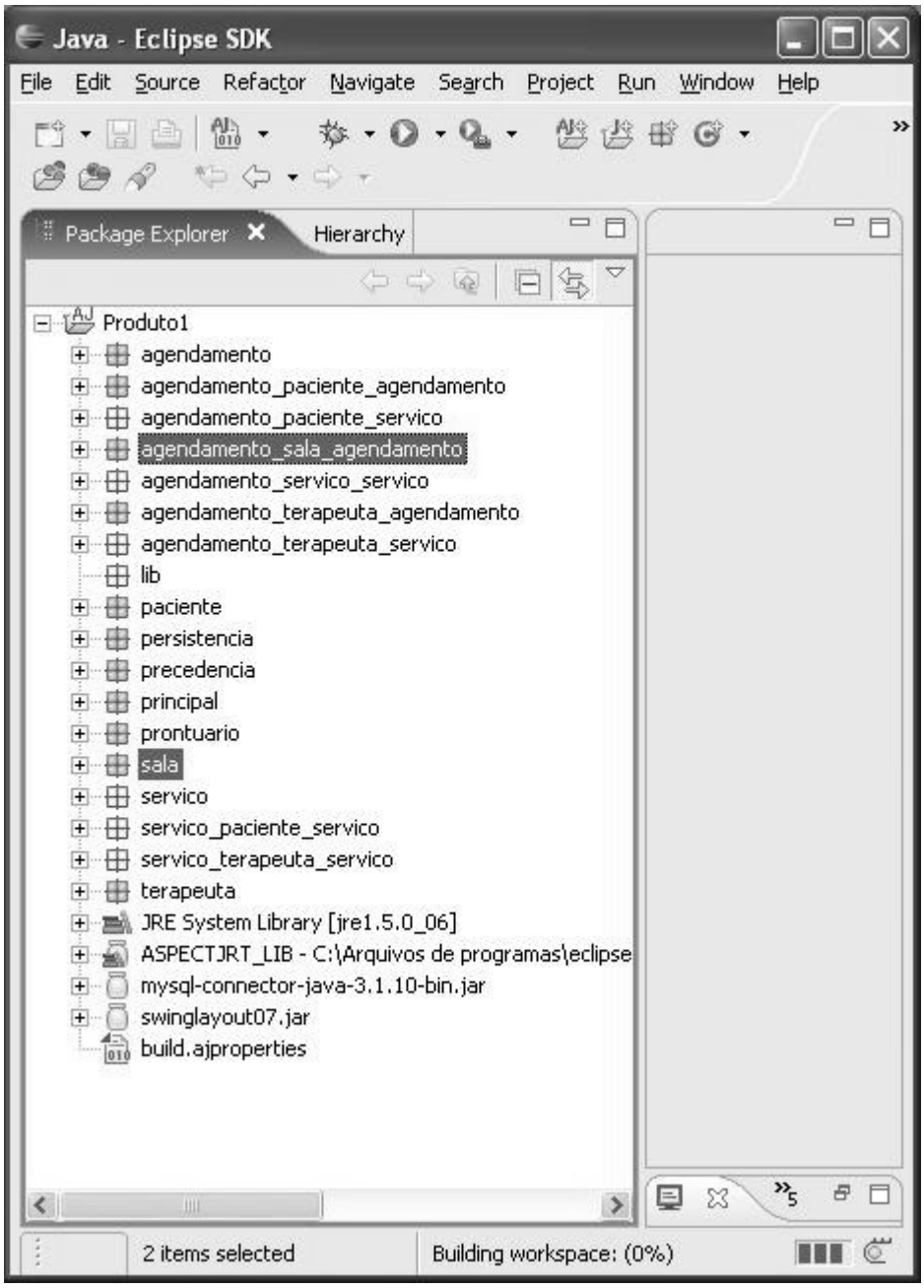

CLÍNICA PSICOLÓGICA "Produto 1"
Agenda
Sair


Java - Eclipse SDK
File
Edit
Source
Refactor
Navigate
Search
Project
Run
Window
Help
Package Explorer
Hierarchy
Produto1
agendamento
agendamento_paciente_agendamento
agendamento_paciente_servico
agendamento_sala_agendamento
agendamento_servico_servico
agendamento_terapeuta_agendamento
agendamento_terapeuta_servico
lib
paciente
persistencia
precedencia
principal
prontuario
sala
servico
servico_paciente_servico
servico_terapeuta_servico
terapeuta
JRE System Library [jre1.5.0_06]
ASPECTJRT_LIB - C:\Arquivos de programas\eclipse
mysql-connector-java-3.1.10-bin.jar
swinglayout07.jar
build.ajproperties
2 items selected


CLÍNICA PSICOLÓGICA "Produto 1"
Pacientes
Terapeutas
Agenda
Sair


Agenda
- MENSAGENS -
<< Semana Anterior
Mês:
Outubro
Ano:
2005
Próxima Semana >>
Paciente:
Seg
26/9
Ter
27/9
Qua
28/9
Qui
29/9
Sex
30/9
07:00
08:00
09:00
10:00
11:00
Voltar


Agendamento
- MENSAGENS 1-
- MENSAGENS 2 -
Toda Semana?
Sim
Não
Paciente: Stanley Fabrizio Pacios
Terapeuta:
Agendar
Desmarcar
Fechar

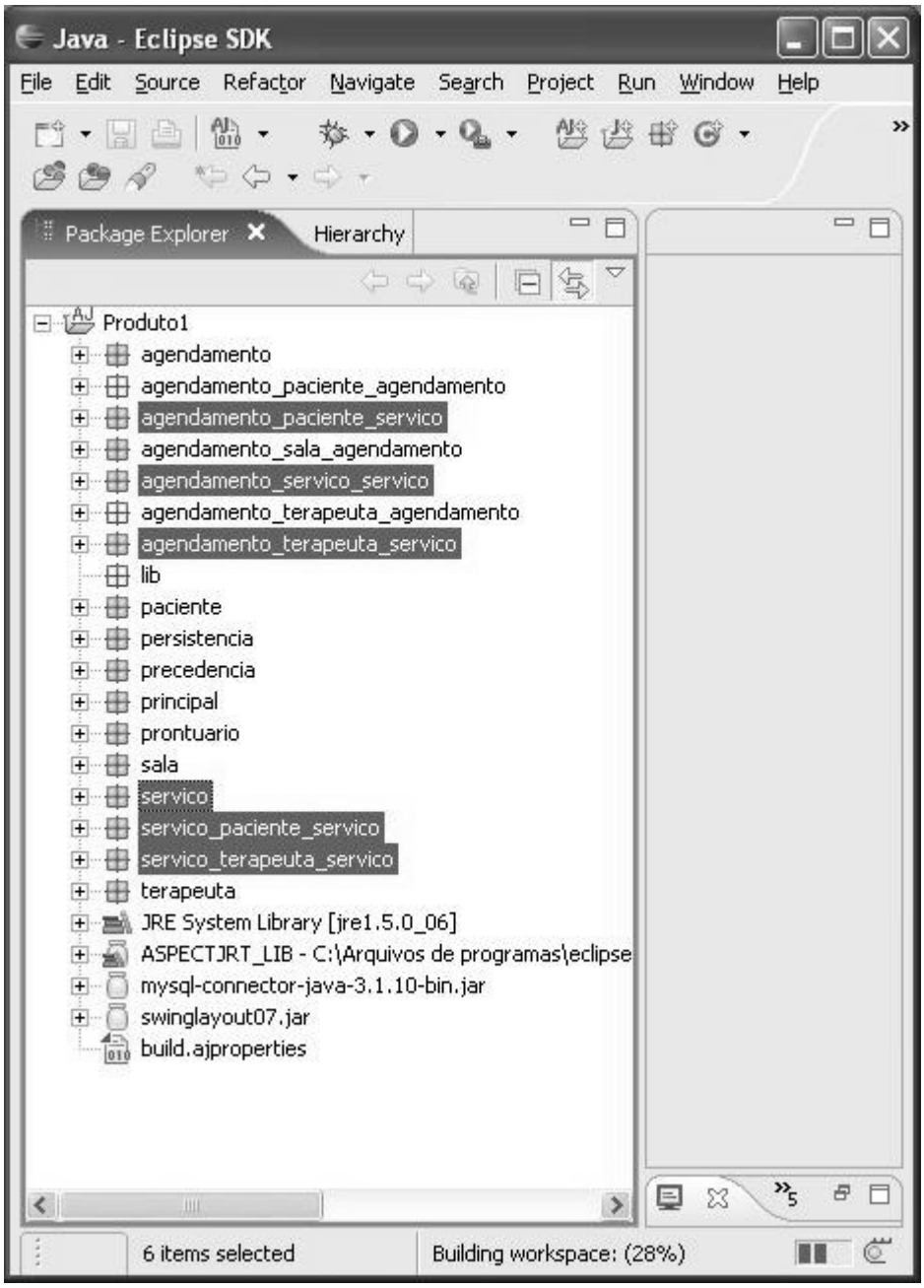

Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
Produto1
agendamento
agendamento_paciente_agendamento
agendamento_paciente_servico
agendamento_sala_agendamento
agendamento_servico_servico
agendamento_terapeuta_agendamento
agendamento_terapeuta_servico
lib
paciente
persistencia
precedencia
principal
prontuario
sala
servico
servico_paciente_servico
servico_terapeuta_servico
terapeuta
JRE System Library [jre1.5.0_06]
ASPECTJRT_LIB - C:\Arquivos de programas\eclipse
mysql-connector-java-3.1.10-bin.jar
swinglayout07.jar
build.ajproperties
2 items selected
Building workspace: (0%)

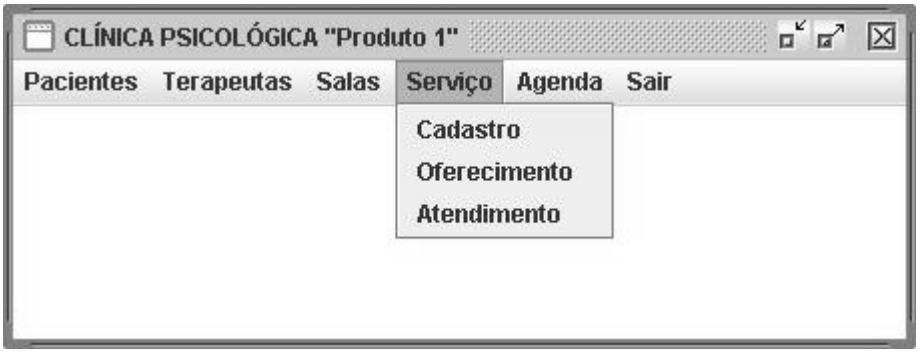

CLÍNICA PSICOLÓGICA "Produto 1"
Pacientes
Terapeutas
Salas
Agenda
Sair
Cadastro


Salas
- MENSAGENS -
Tipo:
Terapia
Salas:
Sala1
Sala2
Cadastro
Fechar


Sala
- MENSAGENS -
Nome:
Descrição:
Tipo:
Novo Tipo:
Salvar
Limpar
Excluir
Fechar

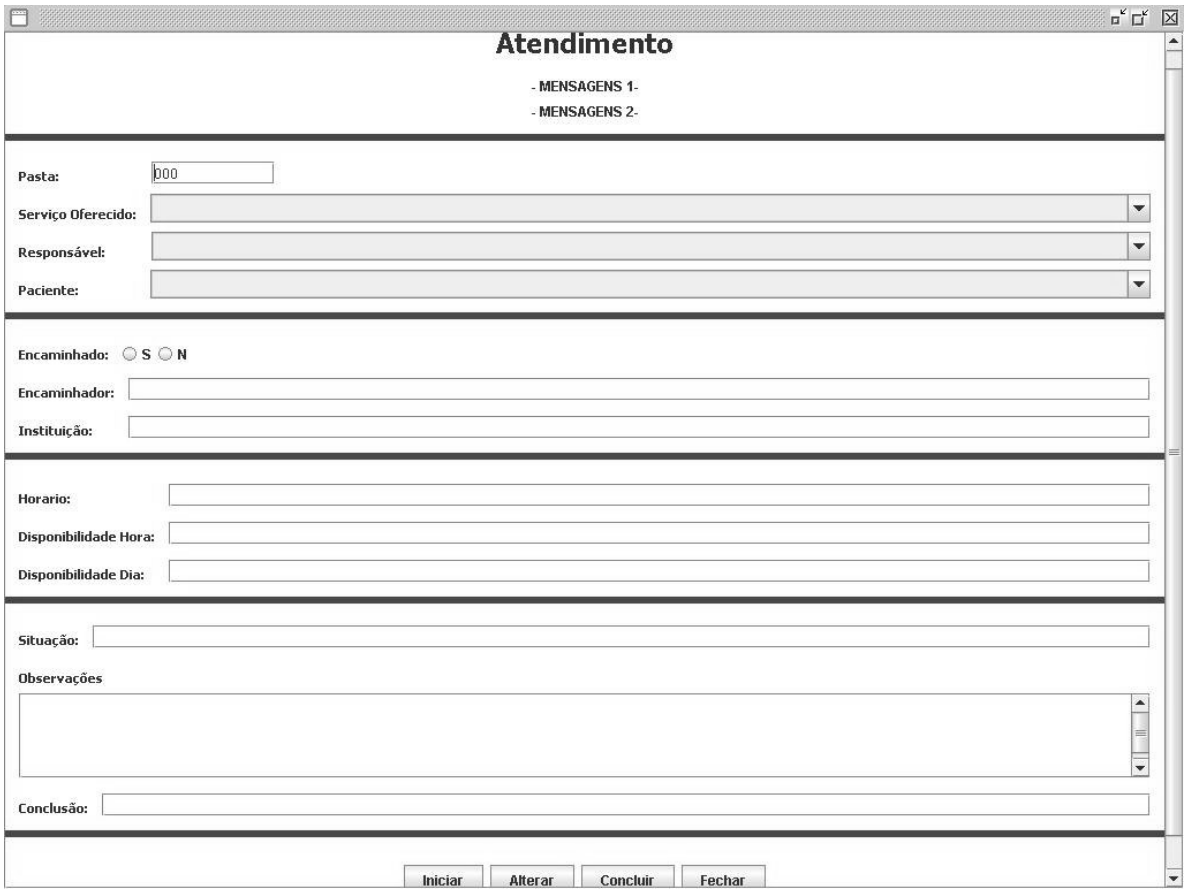

Agendamento
- MENSAGENS 1-
- MENSAGENS 2 -
Toda Semana?
Sim
Não
Paciente: Stanley Fabrizio Pacios
Terapeuta:
Sala:
Tipo:
Terapia
Sala1
Sala2
Agendar
Desmarcar
Fechar

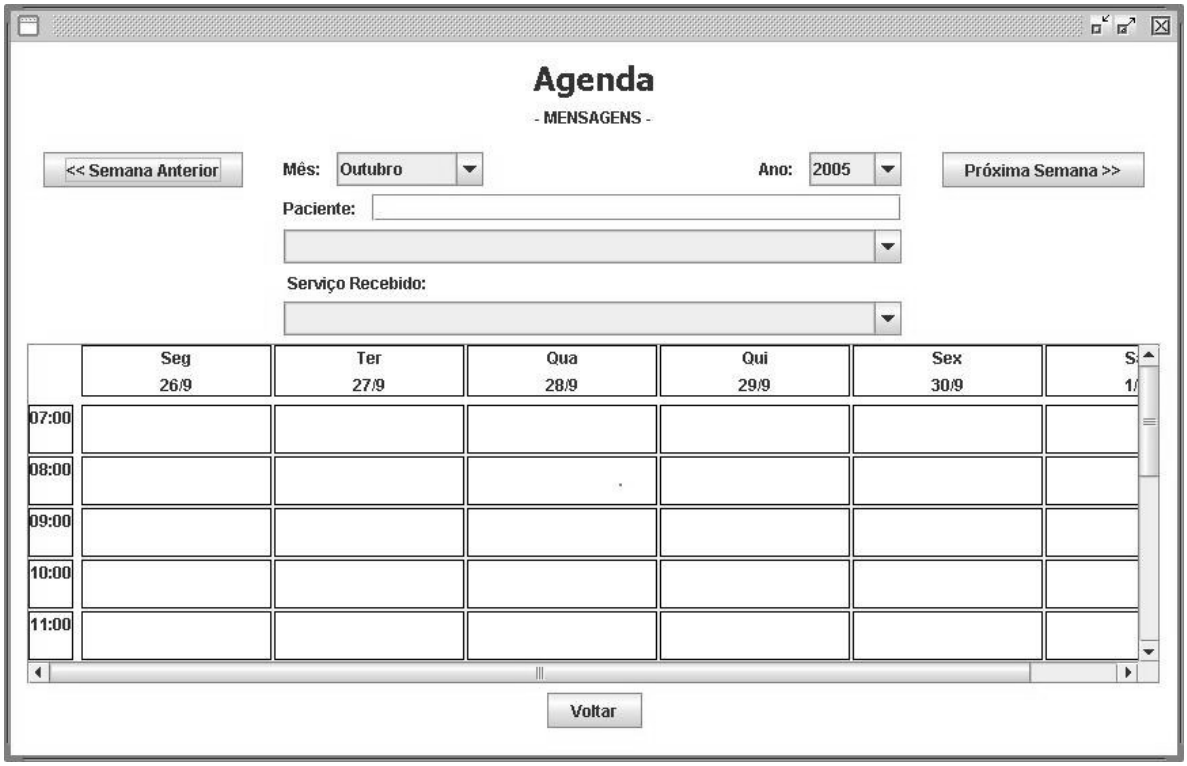

Java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
Produto1
agendamento
agendamento_paciente_agendamento
agendamento_paciente_servico
agendamento_sala_agendamento
agendamento_servico_servico
agendamento_terapeuta_agendamento
agendamento_terapeuta_servico
lib
paciente
persistencia
precedencia
principal
prontuario
sala
servico
servico_paciente_servico
servico_terapeuta_servico
terapeuta
JRE System Library [jre1.5.0_06]
ASPECTJRT_LIB - C:\Arquivos de programas\eclipse
mysql-connector-java-3.1.10-bin.jar
swinglayout07.jar
build.ajproperties
6 items selected
Building workspace: (28%)


CLÍNICA PSICOLÓGICA "Produto 1"
Pacientes
Terapeutas
Salas
Serviço
Agenda
Sair
Cadastro
Oferecimento
Atendimento


Serviço
- MENSAGENS -
Nome:
Descrição:
Salvar
Limpar
Fechar
Excluir


Iniciar Oferecimento de Serviço
- MENSAGENS 1-
- MENSAGENS 2-
Nome:
Serviço:
Responsável:
Descrição:
Modalidade:
Grupo:
Faixa Etária:
Tema:
Valor:
000000.00
Iniciar
Parar
Fechar


Atendimento
- MENSAGENS 1-
- MENSAGENS 2-
Pasta:
000
Serviço Oferecido:
Responsável:
Paciente:
Encaminhado: S N
Encaminhador:
Instituição:
Horario:
Disponibilidade Hora:
Disponibilidade Dia:
Situação:
Observações
Conclusão:
Iniciar
Alterar
Concluir
Fechar


Agenda
- MENSAGENS -
<< Semana Anterior
Mês: Outubro
Ano: 2005
Próxima Semana >>
Paciente:
Serviço Recebido:
Seg 26/9
Ter 27/9
Qua 28/9
Qui 29/9
Sex 30/9
07:00
08:00
09:00
10:00
11:00
Voltar