Diretrizes para desenvolvimento de linhas de produtos de software com base em *Domain-Driven Design* e métodos ágeis

Otávio Augusto Cardoso Macedo

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito: 2 de dezembro de 2008

Assinatura: _____

Diretrizes para desenvolvimento de linhas de produtos de software com base em *Domain-Driven Design* e métodos ágeis

Otávio Augusto Cardoso Macedo

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 de Computação e Matemática Computacional.

USP - São Carlos
Fevereiro/2009

Agradecimentos

Agradeço à professora Rosana Teresinha Vaccare Braga, por todas as dicas, conselhos e ensinamentos, e por ter aceitado tão prontamente me orientar nesta investigação.

Agradeço a toda minha família, especialmente ao meu pai, que, pelo exemplo e pela disciplina, despertou em mim o gosto pela Ciência. Obrigado igualmente pela educação baseada em princípios morais sólidos, sem os quais, certamente, não teria sido possível chegar até aqui.

Agradeço ao Joe Martins, pela oportunidade que me ofereceu de aprender sobre desenvolvimento de software, nos tempos da IT GIS. Obrigado, também, pela força nos primeiros meses do Mestrado.

Agradeço à Paula, que esteve sempre ao meu lado, especialmente nos momentos mais críticos desta penosa empreitada. Seu incentivo tem me ajudado muito!

Agradeço aos amigos que estiveram por perto durante esse tempo: Lionis, Julio, Antonielly, Tott, Cristina (Kika), Carlos, Wanderley, Krishna, Padma e Matheus. Esses agradecimentos também se estendem aos amigos de longe, pois a amizade não se rompe facilmente com a distância. São eles: Elcimar, Tiago, Marisa e Glauber.

Por fim, meus agradecimentos institucionais ao ICMC e ao CNPq.

Resumo

Linhas de produtos de software (LPS) são coleções de sistemas que compartilham características comuns, desenvolvidas de forma sistemática a partir de um conjunto comum de ativos centrais. Dentre as técnicas propostas por vários autores para o desenvolvimento de LPS, dois padrões podem ser observados: o processo de desenvolvimento geralmente é sequencial e o foco do projeto (*design*) costuma estar em interesses técnicos, como alocação de componentes e separação em subsistemas, e não em um modelo de domínio. Embora essas práticas sejam reportadas como bem-sucedidas, um outro paradigma de desenvolvimento, baseado em métodos ágeis e em um conjunto de princípios de projeto conhecido como *domain-driven design*, é apresentado neste trabalho e pode produzir resultados mais satisfatórios, comparados aos métodos tradicionais. Essa hipótese é sustentada por comparações entre padrões de modelagem e por um estudo de caso feito neste trabalho.

Abstract

Software product lines (SPL) are collections of systems that have common features. Those systems are systematically developed from a common set of core assets. Two patterns outstand among the various techniques proposed by several authors for developing software product lines: the development process is generally sequential and the primary design concerns deal with technical issues, such as component allocation and system partitioning (into subsystems). The importance of a domain model is often neglected. Although such practices have been reported as successful, this work proposes another paradigm, which can yield more satisfactory results. This paradigm is based on agile methods and a set of design principles known as domain-driven design. This hypothesis is supported by comparing different modeling patterns and by developing a case study.

Sumário

Lista de Figuras

Lista de Tabelas

Listings

CAPÍTULO

1

Introdução

1.1 Contextualização e Motivação

De modo geral, reúso de software é uma área de estudo da Engenharia de Software cujo propósito é a construção e a manutenção de um repositório de ativos reusáveis, visando a minimizar o esforço na criação de novas aplicações (Frakes e Kang, 2005). Ativos reusáveis são partes de software, como bibliotecas, classes, componentes, *frameworks*, etc, que podem ser reaproveitadas em contextos diferentes.

Dentre as técnicas para exploração do reúso de ativos na construção de novos produtos, destacam-se as linhas de produtos de software (LPS). As LPS são coleções de sistemas que compartilham características comuns, definidas inicialmente por Parnas (1979) como uma "família de sistemas". Nos últimos dez anos, o assunto tem sido alvo de interesse de vários autores (Weiss e Lai, 1999; Griss, 2000; Clements e Northrop, 2001; Gomaa, 2004). Por possuírem características (ou funcionalidades) em comum, é desejável que a implementação dessas aplicações seja feita reaproveitando-se ativos construídos previamente e incluídos em algum repositório. O reúso nas LPS pode ser alcançado por duas abordagens: construindo ativos previamente para serem reusados por várias aplicações no futuro ou analisando aplicações existentes, de modo a identificar suas similaridades. A partir dessas similaridades, pode-se construir ativos de uso comum a todas essas aplicações.

Do ponto de vista econômico, o que se pretende com o reúso é a maximização do "retorno do investimento" (comumente conhecido pela sigla em inglês ROI). Em parte, esse retorno é diretamente proporcional à vida útil desses ativos reusáveis. Por outro lado, devido à natureza altamente maleável do software, é normal que haja grandes pressões por mudanças em seu escopo, causadas

pelas novas serventias que os usuários encontram para as funcionalidades já existentes (Brooks, 1987). Assim, quanto mais o tempo passa, maior será a pressão por mudanças nos sistemas já implantados.

Além da vida útil, outro fator maximizador do ROI nas estratégias de reúso é a escalabilidade. Em outras palavras, o reúso trará mais retorno na medida em que for possível sempre acrescentar novos ativos reusáveis ao repositório e gerar novas aplicações a partir desses ativos. Isso necessariamente leva a aumento e alterações de escopo, o que, por sua vez, cria uma tendência de modificações nos ativos reusáveis já existentes, a fim de que estes se adaptem às novas circunstâncias de uso.

Percebe-se, portanto, que o aumento do reúso leva necessariamente a um aumento na pressão por transformações nos sistemas. Essa pressão causada pela reusabilidade soma-se àquela encontrada naturalmente em projetos de software, proveniente de novas realidades de negócio, necessidades que são descobertas durante o desenrolar de um projeto, problemas na comunicação entre *stakeholders* e assim por diante.

Apesar dessa forte correlação entre reusabilidade e mudança, os métodos tradicionais não incorporam o conceito de adaptabilidade a mudanças no seu corpo de princípios. Por não reconhecerem as mudanças como parte natural do desenvolvimento de software, esses métodos tentam evitá-la ao máximo, por meio de especificações prematuras e exaustivas.

Os problemas decorrentes desse cenário de baixa adaptabilidade a mudanças foram criticados por vários autores, o que levou a várias inovações nas técnicas de desenvolvimento. Surgiram, então, a prototipação (Brooks, 1975), o modelo espiral (Boehm, 1988) e diversos métodos iterativos e incrementais (Larman e Basili, 2003).

Mais recentemente, foram criados vários métodos de desenvolvimento, baseados no mesmo conjunto de princípios e valores, como a ênfase em desenvolvimento iterativo (com iterações curtas), *feedback* constante do cliente, a idéia de que código funcionando é a principal medida de progresso, a valorização da comunicação oral, redução de barreiras funcionais, entre outros. Esses métodos ficaram conhecidos como "métodos ágeis", a partir da publicação do manifesto ágil[1]. Alguns dos principais métodos ágeis são o Scrum (Schwaber, 1995), a Programação Extrema (Beck, 2000), *Feature Driven Development* (Palmer e Felsing, 2002) e *Dynamic Systems Development Method* (Voigt *et al.*, 2004).

Os valores propostos pelos métodos ágeis têm provocado em alguns casos reações de ceticismo, tanto por parte da Academia quanto da Indústria de software. Apesar da polêmica, essas práticas ágeis têm demonstrado que é possível desenvolver software com maior garantia de valor de negócios, com melhor atendimento ao escopo e maior capacidade de lidar com mudanças (Cockburn, 2003; Forrester, 2004; Teles, 2005).

Os detalhes técnicos de projeto também influenciam muito na capacidade de lidar com mudanças. Ao contrário de outros produtos de engenharia, a complexidade de um software provém do

[1] http://www.agilemanifesto.org

grande número de elementos distintos que o constituem. Um programa orientado a objetos, por exemplo, é constituído de diversas classes e interfaces, todas idealmente distintas entre si. À medida que novas funcionalidades são introduzidas, mais elementos novos são criados, contribuindo para o crescimento constante dessa complexidade intrínseca dos sistemas.

Embora não seja possível escapar dessa complexidade, pode-se pelo menos restringí-la à própria complexidade do domínio para o qual a aplicação é desenvolvida. Esse é o propósito de uma filosofia de projeto de software conhecida como *Domain-Driven Design* (DDD) (Evans, 2003; Nilsson, 2006), que sugere que os projetos de software devem ser focados em modelos ricos em conhecimento, feitos diretamente no código, a partir de uma linguagem que permeia toda a comunicação sobre o domínio.

Nos últimos anos, diversos trabalhos realizados no ICMC-USP têm focado em estratégias de desenvolvimento de LPS (Pereira e Braga, 2007; BRAGA *et al.*, 2007; Gomes e BRAGA, 2008; Donegan, 2008). Esta dissertação pretende dar uma contribuição a essa investigação, pela exploração de DDD e métodos ágeis no contexto de linhas de produtos.

1.2 Objetivos

O principal objetivo deste trabalho é estudar a aplicação de DDD e métodos ágeis ao desenvolvimento de linhas de produtos de software, a fim de analisar as principais vantagens em relação aos métodos tradicionais. Essa análise envolve questões como reúso, configuração de variabilidades, geração de produtos, gerenciamento de requisitos e projeto orientado a objetos.

Para melhor fundamentar a discussão, é feito um estudo de caso, em que se desenvolve uma linha de produtos para sistemas de transporte urbano. Essa implementação, baseada nos princípios aqui sugeridos, é comparada a uma implementação de referência para o mesmo domínio, baseada nas propostas tradicionais. Pela investigação de alguns problemas práticos encontrados no desenvolvimento dessa LPS, é possível avaliar os pontos em que a proposta apresentada neste trabalho difere das propostas encontradas na literatura.

A partir das lições aprendidas no desenvolvimento da linha de produto baseada em DDD e pela comparação com a implementação baseada nos métodos tradicionais, propõe-se um conjunto de diretrizes de desenvolvimento de LPS. Tais diretrizes são regras de caráter prático que visam a guiar o desenvolvimento de novas linhas de produtos. Adaptadas a cada contexto particular, as diretrizes podem ajudar a alcançar maior flexibilidade e extensibilidade nas LPS.

1.3 Organização da dissertação

O Capítulo 2 trata do reúso de software e das várias técnicas propostas na literatura para alcançá-lo. Primeiro são abordadas as engenharias de domínio e de aplicações, que propõem

um conjunto de atividades de desenvolvimento focadas na criação de linhas ou famílias de produtos. Em seguida, são discutidas as principais tecnologias e especificações para criação e reúso de componentes.

O Capítulo 3 é dedicado principalmente a *Domain-Driven Design*. São examinados os princípios gerais de DDD, seguidos dos padrões básicos de desenvolvimento para se construir um modelo de domínio rico em conhecimento, diretamente no código. Vários trabalhos relacionados são apresentados. Os principais métodos ágeis e seus princípios gerais também são discutidos.

O estudo de caso é examinado no Capítulo 4. Após uma breve exposição dos requisitos e das decisões gerais de projeto, são analisadas as quatro primeiras iterações do projeto, até que a LPS alcance um conjunto de funcionalidades que a torne apta para lançamento. Em cada uma das iterações, diferentes problemas e soluções de projeto são analisados. As técnicas de engenharia de aplicações utilizadas também são comentadas. As decisões de projeto são comparadas às de uma implementação de referência para esse mesmo sistema de transporte urbano. Os pontos mais importantes resultantes dessa comparação são ressaltados. Um conjunto de diretrizes é delineado a partir dos *insights* obtidos na implementação baseada em DDD e na comparação desta com a implementação de referência.

No Capítulo 5, as conclusões obtidas a partir deste trabalho são discutidas, apresentando suas principais contribuições e limitações. São sugeridas, por fim, idéias para trabalhos futuros.

CAPÍTULO

2

Reúso de Software

2.1 Considerações iniciais

Desde o surgimento da área de conhecimento conhecida como Engenharia de Software, entre as décadas de 1950 e 1960, existe a preocupação com o reúso de partes de software para a criação de aplicações cada vez mais complexas e com o menor esforço possível. Naquele mesmo momento histórico, a comunidade científica cunhava o termo "crise do software", diante da percepção de que havia sérios problemas nos projetos de desenvolvimento de software, incluindo questões de qualidade, atendimento do escopo proposto e estouro de prazo e orçamento.

Um dos primeiros trabalhos sobre reúso foi o de McIlroy (1968), que sugeriu, pela primeira vez, o compartilhamento de conjuntos de funções, aos quais deu o nome de bibliotecas de software. Segundo Frakes e Kang (2005), reúso de software é definido como o uso de software ou conhecimento sobre software já existente — denominados ativos reusáveis — a fim de construir software novo.

Embora atingir o objetivo de maximização do reúso seja um grande desafio, uma das constatações feitas por desenvolvedores de software ao longo dos últimos 40 anos é a de que a maioria dos sistemas é construída para domínios já conhecidos e para os quais já existem vários outros sistemas. O termo "domínio", neste caso, é definido como uma área de conhecimento ou atividade caracterizada por um conjunto de conceitos e terminologia compreendidos pelos participantes dessa área (Booch, 2000). Portanto, o desenvolvimento de um novo sistema poderia, em tese, se beneficiar dos esforços empreendidos anteriormente por outros desenvolvedores.

Uma das principais técnicas de exploração do reúso é a engenharia de domínio (ou engenharia de linhas de produtos), baseada no reaproveitamento de ativos por meio de processos sistemáticos

de desenvolvimento e documentação de partes de software, de modo que vários produtos relacionados entre si possam utilizar o que foi desenvolvido previamente, facilitando assim, a geração de uma nova aplicação.

Outra técnica de reúso bastante difundida é a do desenvolvimento baseado em componentes. Segundo Szyperski *et al.* (1998), um componente de software é um elemento de sistema que oferece um serviço ou evento pré-determinado e que é capaz de se comunicar com outros componentes. Os seguintes critérios devem ser satisfeitos para que um componente seja classificado como tal:

- Uso múltiplo
- Não específico a um contexto
- Passível de composição com outros componentes
- Encapsulado, isto é, expõe apenas as interfaces
- Uma unidade independente de instalação e versionamento

Este capítulo trata das principais técnicas propostas na literatura orientadas a aumentar a reusabilidade de ativos de software. A seção 2.2 trata das técnicas e conceitos da engenharia de domínio e engenharia de aplicações. Ambas propõem métodos de reúso que podem ser usados na geração de uma nova aplicação. Os dois processos abordados aqui são o ESPLEP e o FAST. A seção 2.3 aborda os principais padrões e tecnologias propostos para o desenvolvimento de aplicações baseadas em componentes. São discutidos os componentes UML, as especificações de arquitetura CORBA e EJB, além das definições de web services.

2.2 Engenharia de domínio e aplicações

2.2.1 ESPLEP/PLUS

O Processo Evolutivo de Engenharia de Linhas de Produtos de Software (ESPLEP, na sigla em inglês) (Gomaa, 2004) é um modelo de desenvolvimento de software que procura unificar engenharia de linhas de produtos e engenharia de aplicações dentro de um modelo coerente de desenvolvimento de software. O processo adapta várias técnicas tradicionais de desenvolvimento de sistemas únicos (*single systems*) para o contexto de linhas de produtos de software. Esse processo é definido dentro de um método mais geral, chamado Engenharia de Software baseada em UML para Linhas de Produtos (PLUS, na sigla em inglês).

Engenharia de linhas de produtos

O ESPLEP define duas estratégias para engenharia de linhas de produtos: engenharia direta (ou avante) e engenharia reversa. No primeiro caso, o foco é desenvolver primeiro o núcleo da linha de produtos, isto é, o conjunto de funcionalidades que deverão estar presentes em todos os produtos da linha. Após a determinação desse núcleo, passa-se ao exame das variabilidades necessárias para a linha.

A estratégia de engenharia reversa é aplicada quando já existe um conjunto de sistemas que compartilham funcionalidades. A identificação do núcleo, neste caso, é feita *a posteriori*, a partir das funcionalidades que são comuns a todos os sistemas analisados. As demais funcionalidades são consideradas variabilidades, as quais são classificadas segundo um padrão que é explicado mais adiante.

Duas características se sobressaem no ESPLEP: a divisão do processo de desenvolvimento de uma linha de produtos em fases e o uso intensivo de diagramas UML, incluindo o uso de estereótipos próprios. Ambas as características são inspiradas no USDP (*Unified Software Development Process*) (Jacobson *et al.*, 1999). Há, contudo, algumas diferenças em relação ao USDP. A principal delas é a relação entre fases de desenvolvimento e fluxos de atividade.

No USDP, as quatro fases (Concepção, Elaboração, Construção e Transição) são ortogonais aos cinco fluxos de atividade (Requisitos, Análise, Projeto, Implementação e Teste). Em outras palavras, cada uma das atividades se estende por todas as fases. Evidentemente, cada atividade pode ser mais ou menos intensa, dependendo da fase. Por exemplo, a atividade de Projeto é bastante intensa ao final da fase de Elaboração e início da fase de Construção, mas não deixa de ser feita nas outras fases, como mostrado na Figura 2.1.

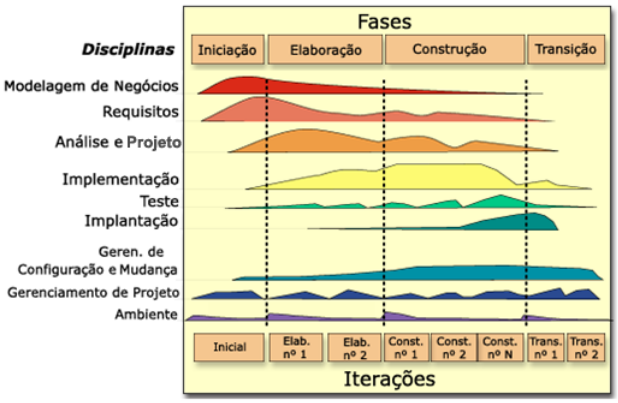


Figura 2.1: Fases e atividades do USDP (retirado de http://wthreex.com/rup).

No ESPLEP, ao contrário, as atividades estão inteiramente contidas dentro das fases. Assim, na fase de Concepção é feito apenas um estudo de escopo e viabilidade do projeto a partir de uma definição preliminar dos requisitos. A fase de Elaboração é dividida em duas partes. A primeira parte dirige a atenção para o núcleo da linha. Neste ponto, pretende-se chegar a uma definição completa dos requisitos (refinando-se os requisitos da fase anterior), que deve ser documentada por casos de

uso e diagramas de características (*features*). Modelos de análise e projeto são determinados também nesta fase, dando origem a diagramas UML, como diagramas de estado, diagramas de classes e diagramas de colaboração. A segunda parte da fase de Elaboração é voltada para a evolução da linha de produtos. O trabalho efetuado na primeira parte é repetido nesta segunda parte, desta vez com foco nas variabilidades.

Na fase de Construção, as funcionalidades do núcleo são codificadas com base nos diagramas produzidos nas fases anteriores. A última fase do ESPLEP — Transição — contempla a execução dos testes funcionais e a entrega de um produto com as funcionalidades consideradas mínimas. No ESPLEP, o conjunto de funcionalidades mínimas inclui todas as funcionalidades do núcleo além das funcionalidades variáveis *default*. Como conseqüência, a primeira entrega do produto é feita somente após um longo tempo de desenvolvimento, executado nas fases anteriores à fase de Transição.

O desenvolvimento das demais funcionalidades (variáveis) é feito de modo iterativo, alternando as fases de Construção e Transição para cada variabilidade, de modo semelhante ao que foi feito para o núcleo. São feitas tantas iterações quanto necessárias, até que o gerente de projeto decida o momento de parar. Caso haja a necessidade de implementações adicionais, elas são postergadas para a fase de engenharia de aplicações.

Todos os artefatos produzidos em cada fase são armazenados em um repositório, para uso posterior. O esquema geral do processo é mostrado na Figura 2.2. Das fases apresentadas na figura, as três primeiras (requisitos, análise e projeto) são discutidas em mais detalhes nas seções subseqüentes. As duas últimas fases (implementação e testes) não são detalhadas no ESPLEP.

O ESPLEP prescreve a criação de dois tipos de artefatos na fase de levantamento de requisitos: modelos de casos de uso e modelos de *features*. Propostos inicialmente por Jacobson (1987), os casos de uso são empregados para descrever o comportamento esperado das aplicações e suas interações com o ambiente (incluindo usuários, dispositivos externos e outras aplicações). As formas mais comuns de representação de casos de uso são as narrativas formais ou semi-formais e os diagramas de caso de uso, que mostram graficamente os relacionamentos entre os diversos casos de uso do sistema. Esta última forma de representação está definida formalmente pela especificação UML 2.0 (OMG, 2007).

No contexto do ESPLEP, o propósito dos modelos de casos de uso é o mesmo. No entanto, algumas extensões foram criadas para lidar com similaridades e variabilidades, conceitos específicos do desenvolvimento de linhas de produtos de software. Assim, cada caso de uso é classificado como pertencente ao núcleo da linha ou como uma variabilidade.

Os casos de uso correspondentes a variabilidades da linha podem ser classificados como opcionais ou alternativos. Casos de uso opcionais são requeridos por alguns, mas não por todos os membros da linha de produtos. Casos de uso alternativos são aqueles que aparecem sob diferentes versões e cada produto da linha pode requerer uma versão diferente. Para denotar a que tipo um determinado caso de uso pertence, são usados estereótipos, uma notação também prevista pela

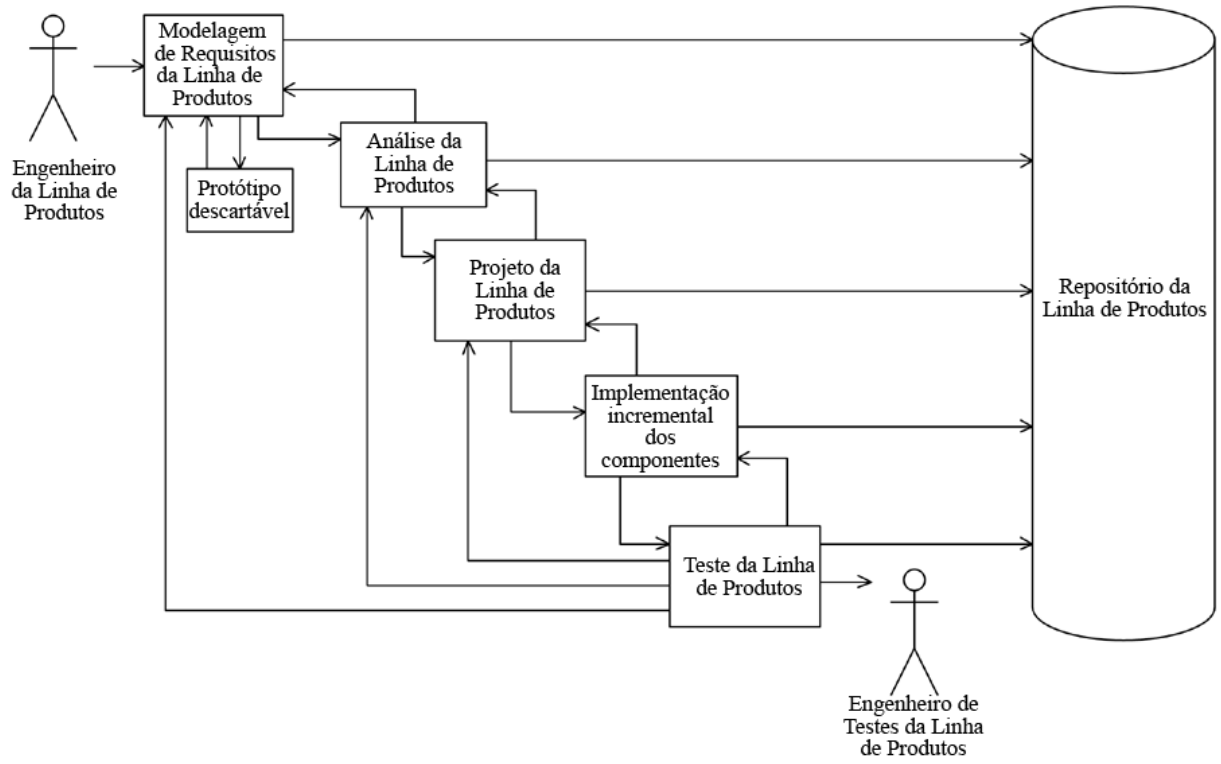


Figura 2.2: Engenharia de Linhas de Produtos, segundo o processo ESPLEP (Gomaa, 2004).

especifiação UML. Os casos de uso do núcleo, os opcionais e os alternativos são denotados pelos estereótipos «`kernel`», «`optional`» e «`alternative`», respectivamente.

De modo geral, o propósito dos casos de uso é representar os requisitos funcionais do sistema considerando cenários específicos. Os requisitos funcionais, entretanto, podem ser analisados sob outra perspectiva: a das *funcionalidades*, que são aspectos do domínio visíveis ao usuário final. Segundo Kang *et al.* (1990), as funcionalidades definem tanto os aspectos do domínio quanto as diferenças entre sistemas relacionados ao mesmo domínio.

Os modelos de *features* são representados por diagramas de classes, em que cada classe (caixa do diagrama) corresponde a uma funcionalidade. Cada funcionalidade de um sistema (ou linha de produtos, neste caso) pode se manifestar em cenários distintos. Por isso, uma funcionalidade pode estar associada a vários casos de uso. Do mesmo modo, um caso de uso pode abranger várias funcionalidades do sistema ou linha de produtos.

As mesmas classificações aplicadas aos casos de uso (“comuns”, “opcionais” e “alternativas”) também são aplicadas às funcionalidades. Adicionalmente, uma funcionalidade pode ser classificada como:

Parametrizada: A variação de valores de determinados parâmetros (em tempo de configuração) altera o comportamento do sistema. O comprimento da senha em uma funcionalidade de autenticação, por exemplo, é um caso de parametrização.

Pré-requisito: Algumas funcionalidades dependem de outras. Neste caso, a funcionalidade da qual se depende é denominada pré-requisito.

Mutuamente inclusiva: Semelhante a pré-requisitos, funcionalidades mutuamente inclusivas devem ser usadas em conjunto. A diferença é que no caso das mutuamente inclusivas, se qualquer uma das funcionalidades for incluída no produto, as demais também o serão.

Funcionalidades relacionadas podem ser agrupadas, de acordo com a maneira com que podem ser combinadas na geração de um produto da linha. O ESPLEP define os seguintes grupos:

Mutuamente exclusivas: Somente uma das funcionalidades pode ser incluída nos produtos finais. Em algumas famílias, uma das funcionalidades deve sempre ser escolhida, ao passo que em outros, pode-se optar por não selecionar nenhuma dessas funcionalidades.

Grupos de exatamente uma funcionalidade: Semelhante ao grupo de funcionalidades mutuamente exclusivas, com a diferença de que a escolha de uma dessas funcionalidades é obrigatória.

Grupos de pelo menos uma funcionalidade: Grupo de funcionalidades opcionais, das quais pelo menos uma deve ser escolhida na geração de um produto.

Grupos de zero ou mais funcionalidades: Em grupos desse tipo, as funcionalidades podem ser escolhidas em qualquer número e combinação. Apesar de parecer uma classificação inútil, ela serve para explicitar que tais funcionalidades estão fortemente relacionadas.

A atividade de análise, segundo a prescrição do ESPLEP, deve ser realizada sobre o espaço do problema, de modo a produzir um modelo coerente do domínio, identificando os principais conceitos, regras e interações entre os elementos. Como resultado dessa atividade, dois tipos de artefatos devem ser criados: um modelo estático, que trata da estrutura do domínio e um modelo dinâmico, que trata dos relacionamentos entre os objetos.

Diagramas de classes são utilizados como notação para representar a modelagem estática do domínio. Mais uma vez, a categorização das classes é feita de acordo com o tipo de variabilidade: do núcleo, opcional ou variante, utilizando estereótipos que foram usados para os casos de uso para acrescentar essa semântica específica aos conceitos do domínio.

Além da classificação segundo a variabilidade, o autor apresenta uma outra hierarquia de categorias para as classes do domínio, conforme apresentado na Figura 2.3. A cada categoria corresponde um estereótipo. Dentro dessa categorização, vale destacar a separação entre entidade e lógica da aplicação, no primeiro nível da hierarquia, logo abaixo da categoria raiz (`«aplicação»`).

Vale lembrar que, embora o autor proponha uma abordagem orientada a objetos para o desenvolvimento de linhas de produtos de software, essa separação entre classes que contêm dados e classes que contêm lógica para manipular os dados vai contra os princípios fundamentais de

orientação a objetos. Esse padrão arquitetural de separação entre dados e comportamento é conhecido como *script* de transação (Fowler, 2003b), o qual sugere a organização da lógica de negócios em procedimentos, de tal modo que cada procedimento manipula uma única requisição do usuário. Geralmente esses procedimentos interagem diretamente com o banco de dados ou com uma camada fina de abstração do banco.

O efeito colateral do uso de *scripts* de transação é a falta de expressividade do código. Como a aplicação é implementada como um conjunto de procedimentos razoavelmente independentes, é difícil fazer abstrações e generalizações no modelo, o que o torna um *modelo anêmico* (Fowler, 2003a). Assim como um paciente é considerado anêmico quando há uma deficiência de hemoglobina no sangue, por analogia um modelo anêmico de domínio é aquele em que os conceitos e dados estão presentes, mas há uma carência de comportamento (ou lógica de negócios), o qual fica fora do modelo, em camadas de código procedimental.

Diversos autores, como Wirfs-Brock e Wilkerson (1989), Evans (2003), Richardson (2006) e Nilsson (2006), além do próprio Fowler (2003b), apresentam objeções a essa forma de modelagem e sugerem padrões melhores, que enfatizam um modelo rico em conhecimento.

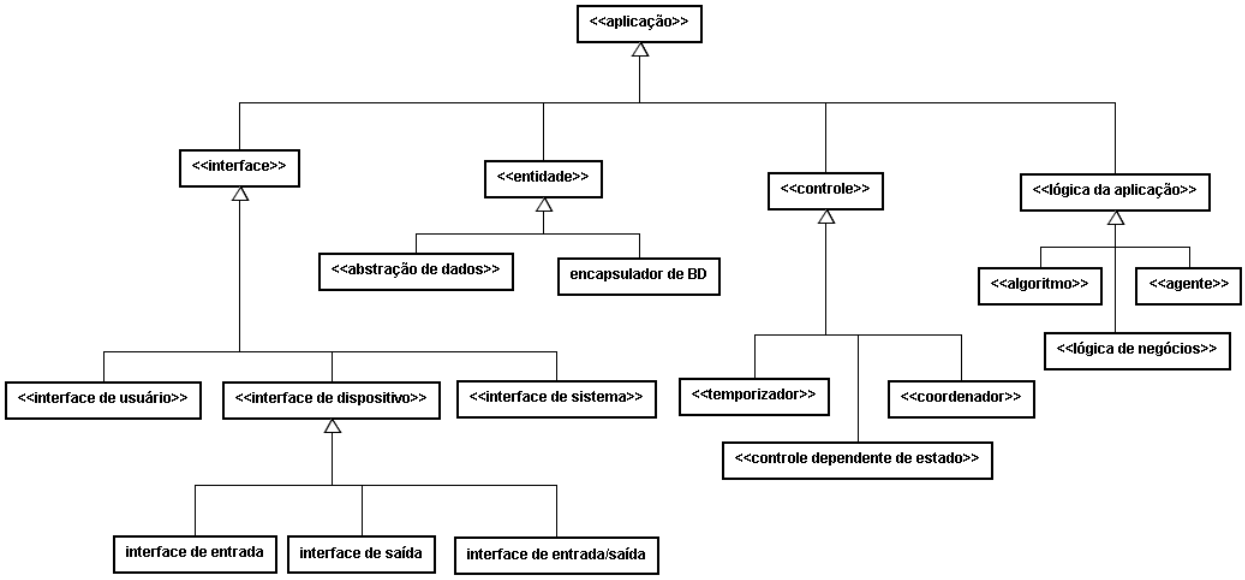


Figura 2.3: Categorização das classes da aplicação (Gomaa, 2004).

Além da modelagem estática do domínio, o processo engloba também a modelagem dinâmica, isto é, as trocas de mensagens entre os objetos, incluindo ordem, parâmetros, nomes das mensagens e tipos de dados transacionados. Para a documentação dessas interações entre os objetos, propõe-se o uso de diagramas de colaboração.

Ao interagirem entre si, trocando mensagens, os objetos podem alterar seus estados ou provocar a alteração de algum estado global do sistema. Segundo proposto no ESPLEP, é preciso também modelar essas mudanças de estado, usando, para tal, os diagramas de estado, também especificados na UML. Em particular, diagramas de estado são bastante adequados para a modela-

gem de sistemas de controle, como uma aplicação embarcada de um forno de microondas ou um controlador de elevadores.

O princípio geral de projeto, segundo proposto no ESPLEP, é o de dividir o sistema em vários subsistemas, cada qual fracamente acoplado aos demais e altamente encapsulado, visando a obter o máximo de ocultação de informação. Tais subsistemas são organizados em torno dos casos de uso, definidos na fase de levantamento de requisitos.

Uma vez que o modelo do sistema é particionado, cada subsistema pode ser implementado como um componente auto-contido, configurável e baseado em mensagens. Um componente é uma unidade de software reusável, que encapsula dados e comportamento — assim como objetos, mas em nível mais alto de abstração — e cuja comunicação com outros componentes é feita por interfaces. Uma discussão mais aprofundada sobre identificação e especificação de componentes é feita na seção 2.3.

Assim, os componentes podem funcionar de modo distribuído, de tal forma que cada componente pode ser instalado em um nó diferente. Por nó, entenda-se qualquer dispositivo de computação, como um servidor, uma estação de trabalho, um dispositivo móvel ou um micro-controlador.

O principal objetivo que se pretende atingir com o uso de componentes é a maximização do reúso. Por isso, a componentização deve ser feita de modo que grupos de objetos fortemente relacionados entre si sejam incluídos dentro de um mesmo componente. O agrupamento de componentes a fim de formar componentes maiores também é previsto, originando os chamados "componentes compostos" (*composite components*), em oposição aos "componentes simples" (*simple components*).

Segundo a descrição abstrata de um componente, as mensagens são enviadas e recebidas por meio de portas, de modo que cada porta está associada a uma ou mais interfaces. Uma interface é um conjunto de operações definidas sobre um componente. As operações que um componente provê para outros são agrupadas em interfaces, denominadas interfaces fornecidas. Em oposição, são definidas as interfaces requeridas, que representam as dependências dos componentes. Dois componentes somente podem trocar mensagens se um deles oferecer uma interface que for requerida pelo outro.

Além da definição dos componentes, suas interfaces e portas, na fase de projeto também é definida a arquitetura da linha de produtos, em que se planeja como esses componentes devem se comunicar e de que modo estarão dispostos no sistema. Para tanto, sugere-se o uso de padrões (*patterns*), que são soluções reusáveis para problemas recorrentes em desenvolvimento de software. Padrões de desenvolvimento de software podem ocorrer em vários níveis de abstração diferentes: padrões arquiteturais, padrões de projeto e "expressões idiomáticas" (padrões de codificação específicos a uma linguagem de programação) (Gamma *et al.*, 1995; Buschmann *et al.*, 1996). Destes três, os padrões arquiteturais são tratados com mais ênfase no ESPLEP. Os padrões arquiteturais são divididos em três grupos principais:

Padrões estruturais tratam da estrutura estática da arquitetura. São de especial importância em sistemas distribuídos, em que se precisa definir quais componentes devem estar hospedados em quais nós.

Padrões de comunicação tratam dos modos de troca de mensagens entre os componentes. Nesses padrões, as principais preocupações são relacionadas à sincronia das mensagens, número de componentes comunicantes, direção das mensagens e formas de descobertas de pares para comunicação.

Padrões de transação especialmente úteis para sistemas que lidam com grandes volumes de dados, como aplicações corporativas, os padrões de transação tratam das formas de se efetuar alterações atômicas, consistentes, isoladas e duráveis (ACID) no estado do sistema, levando em conta as dificuldades dessas operações em um ambiente distribuído.

Engenharia de aplicações

O esquema geral do ESPLEP é mostrado na Figura 2.4. No diagrama, há três atores: o engenheiro de linhas de produtos, o engenheiro de aplicações e o cliente. O papel do engenheiro de linhas de produtos é desenvolver, a partir dos requisitos do domínio, todos os artefatos discutidos anteriormente: descrição e diagrama de casos de uso, diagramas de características, modelos estático e dinâmico e modelo de componentes. Cada um desses artefatos é armazenado em um repositório de linhas de produtos de software.

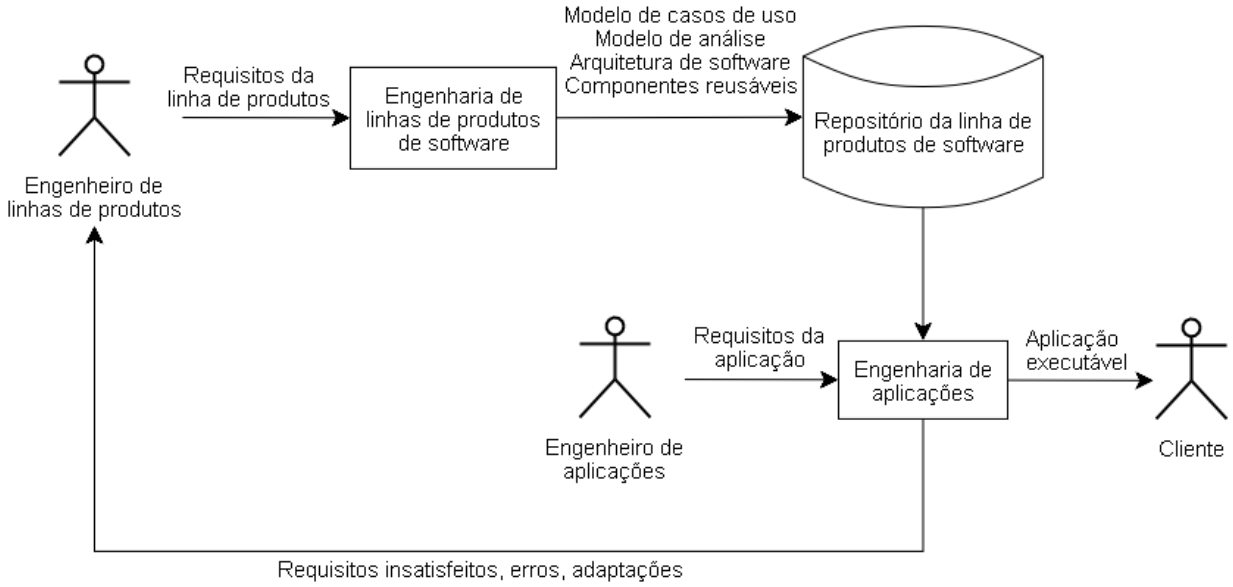


Figura 2.4: Engenharia de Linhas de Produtos, segundo o processo ESPLEP (Gomaa, 2004).

De posse desses artefatos, o engenheiro de aplicações é capaz de gerar produtos específicos, de acordo com as necessidades de cada cliente. O processo de engenharia de aplicações, portanto, consiste basicamente em combinar os artefatos, dando origem a uma aplicação executável, levando

em conta requisitos específicos para cada caso. O mesmo modelo em cascata da engenharia de linhas de produtos é empregado na engenharia de aplicações, ou seja, o desenvolvimento passa seqüencialmente pelas fases de requisitos, análise, projeto, implementação e testes.

Em alguns casos, porém, há alguns requisitos não satisfeitos pelo conjunto de artefatos do repositório. Em outros casos, são detectados erros nos artefatos ou descobre-se que eles não atendem o requisito da aplicação, sendo necessário algum tipo de adaptação. Para esses casos excepcionais[1] há duas abordagens possíveis de solução: sistemática e pragmática.

Na abordagem sistemática, o processo de desenvolvimento volta à atividade de engenharia de linhas de produtos. O objetivo é gerar novos artefatos (ou alterar os já existentes), depositá-los no repositório e voltar à engenharia de aplicações com todas as pré-condições satisfeitas. Na abordagem pragmática, a adaptação é feita somente no produto que se pretende gerar, sem alterar a linha de produtos ou acrescentar novos artefatos ao repositório. A escolha da abordagem depende de cada caso, em que os custos de cada uma devem ser analisados.

2.2.2 FAST

O FAST (Weiss e Lai, 1999) — sigla em inglês para Abstração, Especificação e Tradução Orientadas a Famílias — é um processo de desenvolvimento de linhas de produtos de software introduzido inicialmente na AT&T e melhorado posteriormente pela Lucent Technologies. O FAST é uma descrição em alto nível do processo de desenvolvimento, que estabelece subprocessos, atividades, papéis e artefatos a serem adotados.

Nas próximas subseções, as particularidades do processo são examinadas, passando por uma revisão de seus subprocessos e por uma descrição resumida do modelo PASTA, que descreve precisamente como implantar o processo.

Subprocessos

O FAST é um processo bastante abrangente. Além das questões tradicionais de gerenciamento de projeto e dos detalhes técnicos de desenvolvimento, há também uma preocupação com viabilidade econômica e com questões de *marketing*, como a previsão das necessidades futuras dos usuários. Para dar conta de todo esse espectro de atuação, foram criados três subprocessos.

O primeiro deles é a qualificação do domínio, em que é feita uma análise de viabilidade da criação de uma linha de produtos para aquele domínio. O resultado desse subprocesso é um modelo econômico com informações sobre os custos estimados, os quais encerram tanto o custo inicial de preparação do ambiente para a linha de produtos, quanto o custo de desenvolvimento de cada produto da linha, isoladamente.

A qualificação do domínio fundamenta-se sobre a chamada "hipótese do oráculo", segundo a qual é possível prever as mudanças que irão afetar os sistemas de software. Essa previsão é

[1] Considerando que todas as etapas do processo visam a produzir componentes com o mínimo possível de erro, esses casos são considerados excepcionais.

feita com base em dados históricos de sistemas semelhantes. Supõe-se, ainda, que as mudanças previstas possam ser implementadas de modo independente umas das outras, o que poderia ser alcançado por uma organização correta dos módulos do sistema. Esta última afirmação recebeu o nome de "hipótese organizacional".

O segundo subprocesso é a engenharia de domínio, responsável por produzir uma plataforma (ou ambiente) para a linha de produtos. Sobre essa plataforma podem ser construídos os produtos individuais da linha. A engenharia de domínio é dividida em duas partes: análise e implementação. Durante a análise é construído um modelo de domínio que será usado como base para a implementação da linha.

Uma parte dessa análise é direcionada para a identificação de similaridades e variabilidades existentes no domínio. O produto desta análise é o chamado "documento de similaridades", no qual se registram, além das próprias similaridades e variabilidades, os parâmetros de variação para refinamentos das variabilidades, cenários e decisões importantes que devam ser tomadas.

A outra parte da análise volta-se para a criação de uma linguagem de modelagem da aplicação (AML, na sigla em inglês). A AML é uma linguagem específica de domínio cujo propósito é abstrair os detalhes das aplicações. Idealmente, uma AML permite instanciar um novo produto da linha apenas pela manipulação de seus elementos, sem que seja necessário conhecer os detalhes internos que dão origem a cada aplicação em particular. É uma espécie de linguagem de programação simplificada (por ser específica ao domínio), de altíssimo nível.

O terceiro e último subprocesso é a engenharia de aplicações, que consiste em criar aplicações a partir da plataforma desenvolvida pela engenharia de domínio. A criação dessas aplicações pode ser feita utilizando-se um dos seguintes métodos:

1. Geração de código, em que um compilador específico, desenvolvido pelos engenheiros de domínio, gera código em uma linguagem de alto nível (como Java e C++) ou em uma linguagem de máquina. A geração desse código é feita a partir do que foi especificado por meio da AML.

2. Composição de módulos, por meio da qual módulos previamente construídos são unidos de modo a compor uma aplicação funcional, de acordo com a especificação fornecida pelo engenheiro de aplicações. Novamente, a ferramenta do engenheiro de aplicações é a AML.

Seja qual for o método de instanciação de produtos, é importante que ele esteja abstraído e escondido do engenheiro de aplicações. Isso favorece uma separação clara de interesses e permite que ambos trabalhem em paralelo, aumentando a produtividade do processo. Apesar dessa paralelização, há uma dependência mútua entre ambos os papéis. O engenheiro de aplicações depende do arcabouço desenvolvido pelo engenheiro de domínio, e este depende de respostas e informações do primeiro, como necessidade de desenvolvimento de funcionalidades que ainda não estão presentes na linha, defeitos encontrados, problemas de integração etc.

Observe-se que nessa interação entre as duas áreas de engenharia, há uma troca de informações bastante intensa sobre questões relacionadas ao negócio e às necessidades dos usuários. Por esse motivo, a participação do cliente nesse processo é importante: ele deve dar um retorno rápido ao engenheiro de aplicações sobre as prioridades do projeto.

O modelo PASTA

O processo FAST apresenta apenas as diretrizes gerais de como deve ser feito o desenvolvimento de uma linha de produtos. Os detalhes de implantação do processo em circunstâncias particulares não são discutidos. Para descrever precisamente o processo, foi criado um modelo especial chamado PASTA, acrônimo em inglês para Abstração de Transições de Estado de Processos e Artefatos.

No modelo PASTA, as atividades de tomada de decisão são representadas como máquinas de estado. Cada uma das cinco atividades do FAST (qualificar o domínio, desenvolver o domínio, desenvolver as aplicações, gerenciar o projeto e mudar a família) são representados como estados dessa máquina e são conhecidos como *estados de processo*. Cada um desses estados, por sua vez, é composto de outros sub-estados.

Os artefatos constituem outro elemento dessa máquina de estados. No FAST existe uma hierarquia de *artefatos*, que é apresentada na Figura 2.5. Esses artefatos correspondem às entradas e saídas dos estados de processo. Em cada estado de processo são realizadas uma ou mais *operações*, que são definidas como atividades realizadas sobre os artefatos. Conseqüentemente, toda vez que um artefato passa por um estado de processo, o seu próprio estado é alterado.

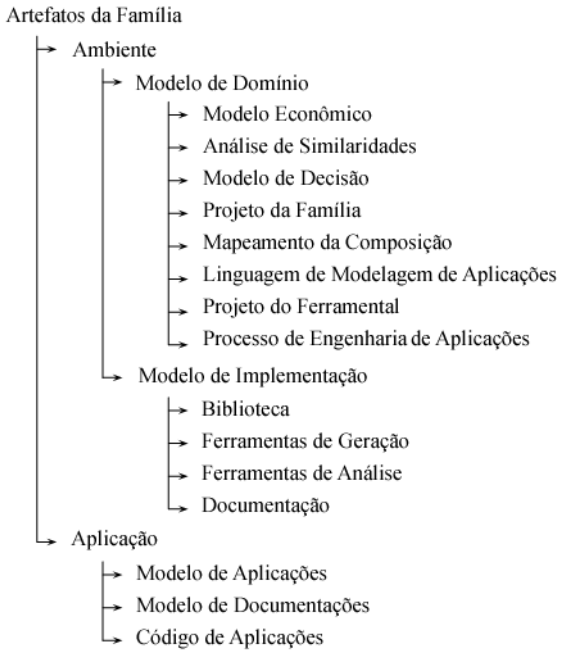


Figura 2.5: Hierarquia de artefatos do FAST (Weiss e Lai, 1999).

Considere-se como exemplo a atividade de qualificar o domínio, representada no diagrama de transições da Figura 2.6. O estado de processo representado na figura recebeu o nome de *Qualificar_Dominio* e possui quatro sub-estados: *Colher_Dados*, *Analisar_Dados*, *Rejeitar* e *Aceitar*.

O artefato de entrada para o estado de processo *Qualificar_Dominio* é o *Modelo_Economico*, inicialmente no estado *Referenciado*, indicando que o artefato é usado em outros pontos do modelo. Se ele ainda não existir, precisa ser criado. Uma vez colhidos os dados necessários para a análise da viabilidade da linha de produtos, considera-se que o modelo econômico mudou de estado, estando agora *Iniciado*. Após a análise, o artefato muda novamente de estado, desta vez para *Revisado*. Dependendo do que foi decidido na análise, o artefato passa por um dos sub-estados *Aceitar* ou *Rejeitar* e o resultado é um documento de aprovação ou reprovação para o início do desenvolvimento da família de produtos.

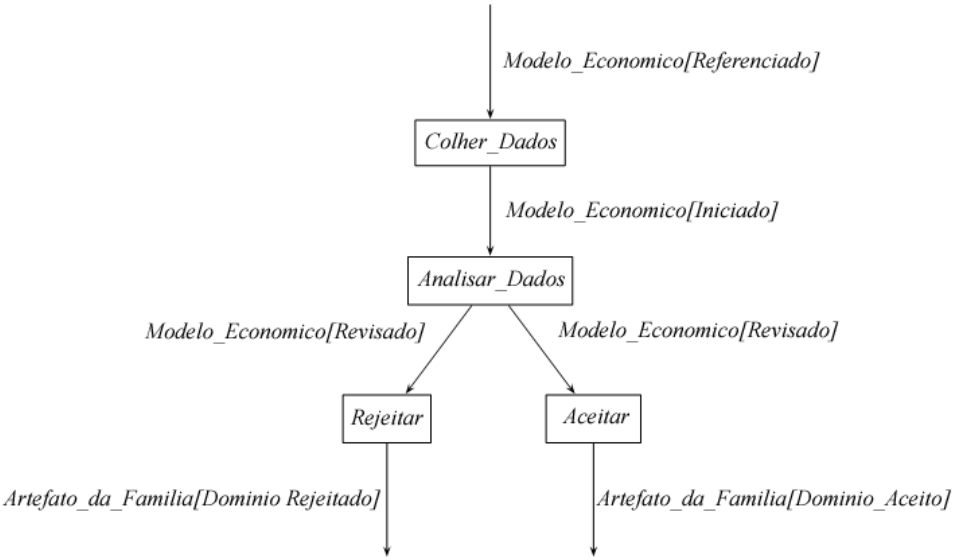


Figura 2.6: Diagrama de transições do estado de processo *Qualificar_Dominio* (Weiss e Lai, 1999).

Ao modelar as atividades como estados de processos (contendo seus respectivos sub-estados), os vários papéis que atuam no desenvolvimento da família de produtos conseguem mais facilmente manter o controle sobre o status do projeto. Esses papéis incluem, além dos engenheiros de domínio e aplicação, o gerente do projeto, o gerente de aplicações e os analistas de suporte e manutenção dos sistemas.

Do mesmo modo, é possível conhecer o estado de cada artefato da linha em um dado momento e em quais estados de processo eles estão sendo operados. Além disso, a natureza concorrente da máquina de estados simplifica o gerenciamento de atividades distintas, mas que se inter-relacionam, como a engenharia de domínio e a engenharia de aplicações, conforme comentado na seção 2.2.2.

2.3 Desenvolvimento baseado em componentes

O principal objetivo que se procura alcançar no desenvolvimento de uma linha de produtos de software é a maximização do reúso, de modo que o desenvolvimento de um novo produto da linha requeira o menor esforço possível. Conforme comentado na seção 2.2.1, uma possível solução para esse problema do reúso é o desenvolvimento de componentes. Várias técnicas e especificações para a construção de componentes já foram propostas. As principais técnicas são analisadas nas próximas seções.

2.3.1 Componentes UML

Cheesman e Daniels (2000) propõem um processo de desenvolvimento de software totalmente baseado em componentes, cuja identificação e especificação são fortemente embasadas no uso de UML. Assim como no ESPLEP, o desenvolvimento prescrito por este processo é seqüencial (embora proponha entregas mais curtas), de modo que a cada fase são produzidos artefatos UML que apóiem as fases subseqüentes.

As atividades propostas pelo processo de componentes UML são baseadas no processo RUP (Kroll e Kruchten, 2003): requisitos, especificação, provisionamento, montagem, teste e implantação. As atividades de especificação, provisionamento e montagem substituem, respectivamente, as atividades de análise, projeto e implementação do RUP. O processo prescreve a geração de vários artefatos, além de técnicas de identificação e especificação de componentes e interfaces.

O primeiro artefato a ser criado, na atividade de requisitos, é o modelo conceitual do domínio, em que os principais conceitos de negócio são identificados, bem como os relacionamentos entre esses conceitos. A representação desse modelo é feita usando-se um diagrama de classes. Nesse diagrama, não são representadas operações e somente os atributos relevantes são mostrados. O objetivo deste modelo é apresentar uma visão geral do domínio, independente do software a ser desenvolvido.

O modelo conceitual deve ser refinado, a fim de produzir uma descrição mais detalhada do negócio. Por isso, é proposta a criação de um modelo de tipos de negócio. Esse artefato contém todos os conceitos definidos no modelo conceitual. Os conceitos, porém, são vistos nesse diagrama como *tipos*, os quais contêm todos os atributos previstos. Os relacionamentos entre os conceitos também são refinados e tratados como associações. Assim, definem-se as multiplicidades e direções dessas associações. O diagrama de tipos de negócio proporciona uma representação do sistema orientada a dados.

O próximo passo é escrever um documento de visão do sistema. Esse documento é uma especificação em alto-nível, curta, contendo apenas os objetivos gerais que o sistema deve alcançar. Recursos adicionais, como *storyboards*, podem ser usados para ajudar a esclarecer a visão geral do sistema. Os principais envolvidos no uso do sistema também são citados nesse documento.

Com a visão geral esclarecida, passa-se à especificação de casos de uso, tanto na forma escrita, quanto em diagramas, que mostram o relacionamento entre os diversos casos de uso. Durante a definição dos casos de uso, são feitas descrições detalhadas do comportamento do sistema, do ponto de vista dos atores que irão interagir com o sistema. Nenhum detalhe de arquitetura ou implementação deve ser citado nesses documentos.

Uma das regras centrais do processo é a separação entre especificação e implementação. Uma especificação define os contratos e restrições gerais que as implementações devem observar. Além dessa divisão, há uma outra, igualmente importante: entre componentes e interfaces. Componentes são unidades de software que encapsulam dados e comportamento. A comunicação entre os componentes se dá por meio de interfaces, que são conjuntos de operações que podem ser suportadas por componentes. As duas classificações são ortogonais, isto é, tanto as interfaces quanto os componentes possuem especificação e implementação.

A especificação de uma interface envolve as operações que a interface deve ter, incluindo os contratos, seu modelo de informação (os tipos com os quais essa interface trabalha), invariantes e restrições adicionais. A especificação de um componente define quais interfaces o componente deve oferecer, quais são as restrições em termos de comunicação com outros componentes (representadas como "interfaces requeridas"), além das suas responsabilidades, em termos de comportamento.

A arquitetura proposta no processo de componentes UML segue o princípio de divisão do sistema em camadas. Nesse tipo de arquitetura, o sistema é organizado na forma de uma pilha de camadas, de tal modo que uma camada dependa somente das camadas inferiores. A comunicação com a camada superior deve ser feita somente na forma de retorno de chamadas a métodos. A separação das responsabilidades em diferentes camadas já provou ser bem sucedida em outros campos, como a pilha de protocolos de rede TCP/IP e OSI (Piscitello e Chapin, 1993), por exemplo, que segue a mesma filosofia de projeto.

O autor propõe quatro camadas, de acordo com o diagrama da Figura 2.7. As duas camadas do topo correspondem a interesses de apresentação de informação e interação com o usuário. Embora seja bastante comum o uso de componentes nessa camada, como botões, menus e tabelas (no caso de interfaces gráficas), a arquitetura proposta concentra-se nas duas camadas de baixo, em que reside a lógica do sistema, propriamente dita.

A terceira camada, de cima para baixo, é a camada de sistema. Essa camada não guarda estado sobre a interação com o usuário. Os componentes dessa camada, denominados componentes de sistema, correspondem diretamente aos casos de uso definidos previamente. As interfaces oferecidas por esses componentes correspondem aos passos dos casos de uso.

A camada inferior é a camada de negócios. Todas as classes de negócio estão encapsuladas dentro desses componentes. As interfaces oferecidas por esses componentes são chamadas interfaces de negócio. As operações dessas interfaces oferecem abstrações dos dados que o componente encapsula. Isso significa que, geralmente, dados de vários objetos podem ser coletados e agrupados na forma de um *transfer object* (Alur *et al.*, 2003) para a camada superior, quando requisitado.

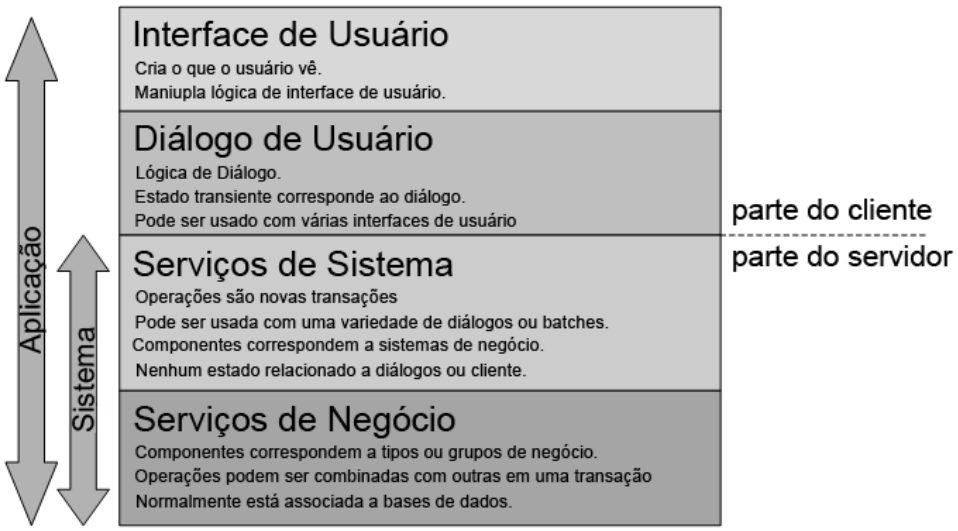


Figura 2.7: Arquitetura em camadas do modelo de componentes UML. Traduzido a partir de Cheesman e Daniels (2000).

De modo geral, os componentes de negócio são responsáveis apenas pelas quatro operações básicas que podem ser aplicadas sobre os dados (criação, leitura, atualização e remoção), enquanto os componentes de sistema carregam a lógica relativa ao domínio. Esse modo de divisão de responsabilidades entre os componentes é mais uma manifestação dos modelos anêmicos de domínio. É importante ressaltar, entretanto, que não é a divisão em camadas que torna o modelo anêmico. Ao contrário, a divisão em camadas é desejável e é aplicada também em modelos ricos, conforme discutido mais adiante. O que torna o modelo anêmico, neste caso, é a representação do domínio de modo pouco expressivo. Boa parte do esforço empregado na criação do modelo de domínio (o primeiro diagrama produzido) é perdida ao se espalhar a lógica do domínio pelos componentes de sistema.

2.3.2 Enterprise JavaBeans

A especificação da tecnologia Enterprise JavaBeans (EJB) (Sun, 2006) define uma arquitetura de componentes do lado do servidor para construção de aplicações corporativas. O objetivo é isolar a lógica de negócios (o *back-end* das aplicações) da interface de usuário (o *front-end*). A tecnologia EJB é baseada na plataforma Java. Segundo a especificação, deve haver um servidor de aplicações, responsável por prover diversos serviços de infra-estrutura, como segurança, controle distribuído de transações, persistência, troca de mensagens, controle de concorrência, serviço de nomes e diretórios e chamadas remotas de procedimentos.

Um dos subsistemas do servidor de aplicações é o *container* EJB, responsável por gerenciar os componentes nos quais a lógica de negócios é implementada. Estes componentes, que recebem o nome de EJBs, são classes que atendem a determinadas convenções de nomenclatura, extensão de

classes e implementação de interfaces. A versão mais recente da especificação introduziu o uso de anotações e simplificou a implementação dos componentes.

Há dois tipos de componentes: os *session beans*, que expõem uma interface de negócios para os clientes, e os *message driven beans*, que são acessados somente pelo servidor de aplicações e cuja responsabilidade é tratar mensagens provenientes de outros EJBs. A comunicação entre um EJB qualquer e um *message driven bean* é feita por filas de mensagens ou por tópicos de mensagens. As duas técnicas garantem um fraco acoplamento entre os componentes.

Os *session beans*, por sua vez, podem ser classificados em duas categorias: *stateless session beans* e *stateful session beans*. Os primeiros são usados para operações que podem ser efetuadas em apenas um passo, como a submissão, pelo cliente, de um objeto a ser persistido pelo servidor. Por esse motivo, tais beans não precisam guardar estado e o servidor de aplicações pode manter uma única instância em memória para cada um desses beans.

Os *stateful session beans*, ao contrário, são usados para operações divididas em vários passos, como o preenchimento de um formulário de compra em aplicações de comércio eletrônico, por exemplo. Como a operação é efetuada em várias etapas, é preciso manter o seu estado entre essas etapas. Isso implica que é preciso instanciar um *bean* para cada operação, a fim de evitar problemas de concorrência, o que torna os *stateful session beans* mais custosos, em termos de recursos do servidor.

Até a versão 2.1, havia também o conceito de *entity beans*, que eram usados para a criação de objetos persistentes. Esses objetos gerenciavam a própria persistência (*entity managed persistence*) ou delegavam ao *container* esse gerenciamento (*container managed persistence*). Em qualquer dos casos, o uso do serviço de invocação remota de métodos era intenso, pois para cada *entity bean* era criada uma cópia remota no cliente (conhecida como *stub*) e qualquer chamada a um método do *stub* era remetida à sua contraparte do lado do servidor.

Na versão 3.0 os *entity beans* foram substituídos pela JPA (*Java Persistence API*), que é capaz de gerenciar a persistência de objetos comuns, sem a necessidade de *stubs* remotos. A JPA é implementada pelos *containers* EJB por meio de provedores de persistência (*persistence providers*), também conhecidos como *EntityManagers*. Todos esses elementos e o modo como eles se relacionam são mostrados na Figura 2.8.

Para lidar com a complexidade inerente aos EJBs, foram criados os padrões J2EE (Alur *et al.*, 2003), que propõem um conjunto de práticas de implementação voltadas especificamente à tecnologia EJB. Os padrões J2EE, por outro lado, recomendam práticas de projeto que violam os princípios básicos de orientação a objetos, como o encapsulamento de dados e comportamento no mesmo objeto, alta coesão e baixo acoplamento.

A tecnologia EJB foi criada com o propósito principal de fornecer um ambiente para a criação de componentes distribuídos. Desse modo, os componentes podem fazer chamadas de métodos de outros componentes remotamente, um serviço que é abstraído pelo servidor de aplicações. Ocorre que o custo de chamadas remotas é ordens de magnitude maior que o de chamadas locais, feitas no

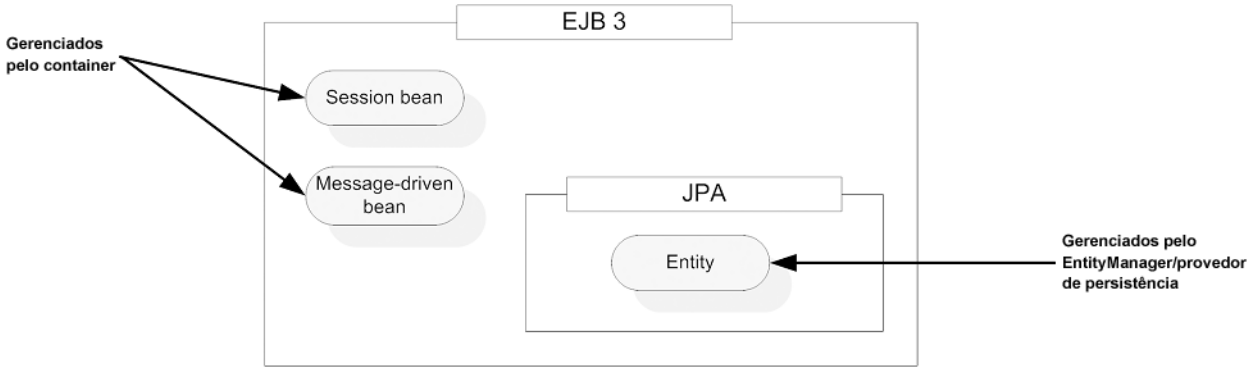


Figura 2.8: Principais elementos da arquitetura de componentes EJB (Panda *et al.*, 2007).

mesmo processo. Uma das soluções possíveis para contornar esse problema é implementar componentes que exponham interfaces de baixa granularidade, reduzindo, asim, o número de chamadas remotas. Uma discussão mais aprofundada sobre interfaces remotas é feita por Fowler (2003b).

2.3.3 CORBA

Semelhante à especificação EJB, CORBA (*Common Object Request Broker Architecture*) é uma arquitetura aberta definida pelo Object Management Group (OMG, 1995) para interoperabilidade de aplicações, comunicando-se via rede. Por ser um padrão aberto, diferentes implementações da arquitetura CORBA podem ser fornecidas por diferentes vendedores, sem que haja problemas na comunicação entre eles.

As aplicações se comunicam por chamadas remotas de métodos, tal qual no padrão EJB. A diferença é que a comunicação é feita por interfaces que escondem totalmente os detalhes de implementação, incluindo a linguagem de programação utilizada. Essas interfaces são definidas pela linguagem IDL (*Interface Definition Language*) e são usadas para encapsular a implementação. Assim, ao receber uma chamada de método, a interface despacha a chamada do modo apropriado, dependendo da linguagem utilizada, sistema operacional, etc. As interfaces CORBA são orientadas a objeto, sendo análogas às interfaces nativas de linguagens OO, como Java e C++.

Um ponto importante que vale destacar é que por esconder completamente quaisquer detalhes de implementação, a arquitetura CORBA pode ser usada até mesmo para a interoperabilidade de sistemas legados, ainda que estes não sejam orientados a objeto. A interface funciona como uma camada de abstração, que é capaz de despachar a invocação de métodos para qualquer linguagem. Contudo, há algumas limitações naturais para essa adaptabilidade: programas desenvolvidos em linguagens procedimentais, como C e Fortran, não podem oferecer várias instâncias de um mesmo tipo, como é feito nas linguagens orientadas a objeto. Neste caso, para cada interface haverá uma única instância.

Para que a comunicação entre as aplicações funcione, é preciso que haja um ORB (*Object Request Broker*), um *middleware* que despacha as chamadas para os objetos corretos nos servidores

corretos. Logo, é preciso que os objetos e suas respectivas interfaces estejam publicados corretamente nesses ORBs. Os métodos das interfaces são invocados por meio de *stubs*, representantes dos objetos hospedados no servidor (portanto com a mesma interface) cujo objetivo é esconder a chamada remota. Assim, do ponto de vista do cliente, a chamada é sempre local. Se o objeto que recebeu a chamada for um *stub*, ele irá redirecionar a chamada ao ORB, sem que o cliente precise ter ciência desse detalhe.

2.3.4 Web Services

Os web services são um padrão de interoperabilidade de componentes, independente de plataforma e de linguagem de programação, proposto pelo World Wide Web Consortium, ou W3C (Booth *et al.*, 2004). Cada componente expõe suas funcionalidades por uma interface de serviço, que pode ser acessada via rede por qualquer outro componente. Os protocolos de comunicação entre esses componentes são todos baseados em formatos XML definidos pelo W3C. Assim, toda troca de mensagem entre os componentes é feita por envio e recebimento de documentos XML. As mensagens trocadas pelos componentes podem ser desde chamadas remotas de procedimentos até mensagens em nível mais alto de abstração, e portanto mais desacoplados da implementação, como os web services baseados no estilo REST (Richardson e Ruby, 2007).

A Figura 2.9 mostra esquematicamente os principais elementos da arquitetura de web services. O primeiro deles é o corretor (*broker*) de serviços. Ele funciona como um catálogo no qual os serviços podem ser publicados e consultados. A publicação e descoberta de serviços é feita pelo protocolo UDDI (*Universal Description Discovery and Integration*), que consiste de três componentes: as páginas brancas, que contêm informações gerais de negócio, como nome e endeço do provedor do serviço; as páginas amarelas, que classificam os serviços de acordo com uma taxonomia padrão; as páginas verdes, com informações técnicas sobre como acessar o serviço. Os corretores de serviço podem ser públicos ou privados.

Os serviços publicados no corretor são oferecidos pelos chamados provedores de serviço, que são os servidores que efetivamente oferecem funcionalidades reusáveis no contexto de web services. Todo provedor deve oferecer uma descrição dos serviços que oferece, o que é feito por meio da WSDL (*Web Services Description Language*). Esta linguagem define a interface pública de um web service, ao especificar uma ligação (*binding*) aos quais os consumidores do serviço podem se conectar. Uma ligação consiste do formato das mensagens (uma abstração dos dados enviados e recebidos por um web service), uma porta (uma abstração das operações que podem ser efetuadas), além do protocolo de rede a ser usado.

As informações técnicas oferecidas pelas páginas verdes dos corretores de serviço podem ser acessadas utilizando-se a linguagem WSDL. Por esse motivo, sempre que um serviço é publicado em um corretor, a descrição desse serviço também deve ser publicada.

Os consumidores dos serviços oferecidos pelos provedores são chamados de requisitantes de serviços. Um requisitante de serviço pode acessar um corretor a fim de descobrir um serviço que

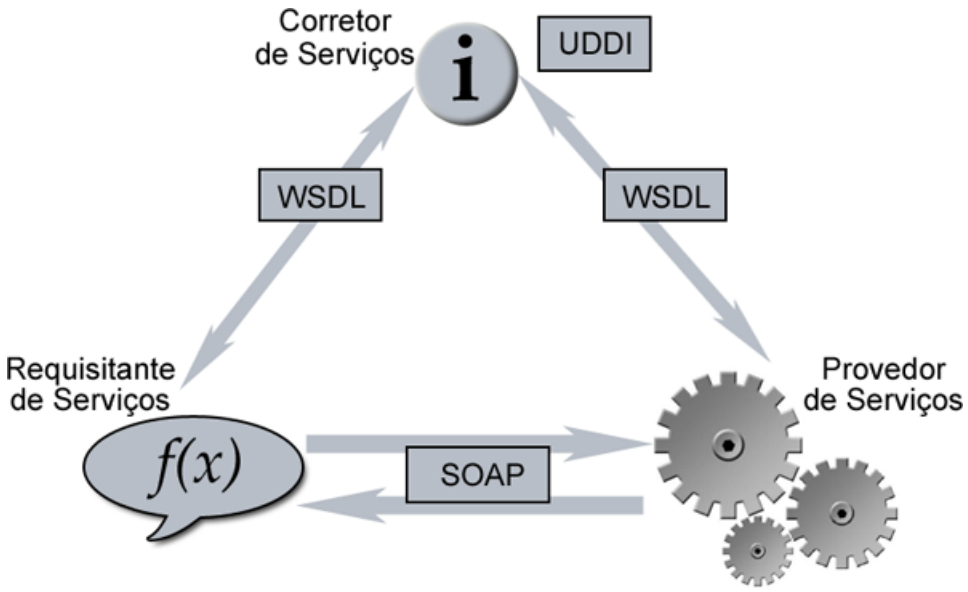


Figura 2.9: Estrutura geral da arquitetura de Web Services.

lhe seja útil ou acessar diretamente o provedor, caso já tenha a descrição formal (codificada em WSDL) do serviço oferecido. Em qualquer dos casos, o requisitante e o provedor estabelecem uma conexão e trocam mensagens utilizando o protocolo SOAP. Este protocolo de comunicação serve-se de outros protocolos da camada de aplicação, como HTTP e SMTP para a troca das mensagens.

Uma mensagem em SOAP contém os seguintes elementos: um envelope (obrigatório), que identifica o documento XML como uma mensagem SOAP; um cabeçalho opcional, que contém informações específicas ao serviço oferecido, como autenticação e pagamento; um corpo, que contém a carga útil da mensagem (os dados de requisição e resposta das operações) e um elemento opcional sobre erros ocorridos durante o processamento da mensagem.

2.4 Considerações finais

As estratégias de reúso discutidas neste capítulo dividem-se em duas partes: os processos de desenvolvimento de linhas de produtos, baseados em engenharia de domínio e engenharia de aplicações, e técnicas e métodos para desenvolvimento de componentes. O processo de componentes UML define tanto o método de desenvolvimento (artefatos, atividades, fases, etc) quanto os detalhes técnicos de identificação de componentes e interfaces. A implementação desses componentes pode ser feita *ad-hoc* ou utilizando-se das tecnologias e especificações comentadas, como EJB, CORBA e web services.

Um padrão recorrente nos métodos apresentados é a separação da implementação em objetos ou componentes que armazenam os dados e outros que implementam toda a lógica de negócios. Esse padrão é conhecido como *script* de transação e não comunica bem o conhecimento sobre o do-

mínio, uma vez que as transações são tratadas como operações independentes, que eventualmente manipulam os mesmos conceitos de domínio.

Os métodos tradicionais, apresentados neste capítulo, apresentam outra característica comum: a serialização das atividades. A fim de tentar reduzir as incertezas dos projetos, o desenvolvimento é dividido em fases e pressupõe-se um fluxo seqüencial de artefatos entre as fases. A implementação, por exemplo, começa somente após a conclusão dos diagramas UML provenientes da análise. A lógica por trás dessa forma de divisão de trabalho é que quanto mais cedo as decisões forem tomadas, menor será o risco de erros no futuro.

Alguns métodos, como o FAST, vão além e partem da hipótese de que é possível prever as necessidades futuras dos usuários e que essas necessidades podem ser implementadas independentemente entre si, o que incentiva ainda mais a modelagem do domínio com *scripts* de transação.

CAPÍTULO

3

Domain-Driven Design e Métodos Ágeis

3.1 Considerações iniciais

Nenhum software robusto e confiável pode ser desenvolvido com base em informações ambíguas ou falhas. Por isso é importante formalizar e refinar o conhecimento, criando um *modelo* do domínio. Como todos os modelos, os de domínio são representações incompletas da realidade, com o objetivo de resolver um problema em particular. Neste caso, o problema é implementar as regras de negócio do domínio em um software.

No entanto, não basta produzir um modelo de domínio qualquer. O foco desse modelo deve estar nos conceitos ou regras de negócios do domínio. Outro ponto importante é que o código das aplicações deve refletir esse modelo, transmitindo informações claras sobre o domínio. Linguagens orientadas a objeto facilitam a observância a esse princípio, já que permitem a construção de estruturas mais abstratas, como objetos e classes.

A orientação a objetos, por outro lado, pode ser mal usada. Um dos sintomas desse emprego incorreto do paradigma é a implementação de todo o comportamento da aplicação em serviços. Ao mesmo tempo, outros objetos são utilizados meramente como estruturas de dados, sem comportamento algum. Estes últimos constituem a implementação do modelo de domínio da aplicação. Conseqüentemente, há uma separação artificial entre dados e comportamento, o que costuma levar a duplicações de código, falta de coesão e alto acoplamento entre as classes. Esse tipo de modelo é denominado modelo anêmico de domínio (Fowler, 2003a).

A metodologia de projeto proposta no ESPLEP, conforme citado na seção 2.2.1, assim como os modelos de componentes baseados em UML, discutidos na seção 2.3.1 são exemplos de modelos anêmicos. Os padrões J2EE também se enquadram nesta categoria.

A fim de tentar corrigir distorções como essas, foi proposto um conjunto de princípios e práticas de desenvolvimento de software, conhecido como *Domain-Driven Design* (DDD) (Evans, 2003). Esse conjunto de princípios e práticas baseia-se em dois pressupostos:

1. Para a maioria dos projetos de software, o foco deve estar na lógica do domínio.

2. Projetos para domínios complexos devem ser baseados em um modelo.

Em particular, uma das implicações desses pressupostos merece destaque: o foco do desenvolvimento não deve estar nos detalhes de infra-estrutura e sim nos detalhes do domínio. A infra-estrutura é importante, mas todo seu valor está em dar suporte à lógica do domínio. Problemas complexos, que surgem normalmente no desenvolvimento, devem ser resolvidos tentando-se compreender melhor os conceitos, que devem ser refletidos no projeto, na documentação, no código e nas discussões sobre o domínio.

Além dos detalhes específicos de projeto, os métodos tradicionais propostos para o desenvolvimento de linhas de produtos de software mostram-se deficitários também no que tange a aspectos técnicos mais gerais de desenvolvimento, quanto em relação ao gerenciamento do projeto. Enquadram-se nesses aspectos: pouca ênfase dada à implementação e aos testes, especificação prematura e exaustiva dos requisitos e entrega tardia dos produtos. A contrapartida sugerida por diversos autores para esses problemas são os métodos ágeis de desenvolvimento.

DDD aplica-se mais adequadamente a projetos orientados a objetos, embora não esteja restrito a este paradigma. No contexto de orientação a objetos, DDD é um resgate dos objetivos que motivaram a criação das primeiras linguagens OO, como Smalltalk (Goldberg e Robson, 1983) no início dos anos 1980. Dentre esses objetivos, encontram-se o de representação em código dos conceitos do problema em questão e a simplicidade e elegância de uma linguagem para representação desses conceitos.

O texto a seguir está organizado da seguinte forma: a seção 3.2 trata dos princípios gerais sobre os quais DDD se baseia. Tais princípios servem como fundamentação teórica para os padrões apresentados na seção 3.3. A revisão apresentada aqui é baseada principalmente nos trabalhos de Evans (2003) e Nilsson (2006). Em seguida, na seção 3.4, são apresentados brevemente alguns exemplos de projetos bem-sucedidos, baseados em DDD, bem como contribuições de ordem teórica ao assunto. A seção 3.5 visa a esclarecer a diferença entre DDD e outros conceitos similares, como MDD e MDA. Por fim, a seção 3.6 trata dos métodos ágeis de desenvolvimento de software.

3.2 Princípios gerais de DDD

Uma das grandes diferenças do DDD em relação a outras filosofias de projeto é o enfoque dado à modelagem. Em DDD, a modelagem do domínio e a implementação da aplicação não são vistas como atividades distintas, ao contrário do que se propõe nos outros métodos, citados anteriormente. De modo geral, o que DDD propõe é que o modelo seja expresso em código. Há duas implicações desta proposta:

1. Todos os outros artefatos são de importância secundária. O código deve bastar para transmitir aos desenvolvedores o conhecimento sobre o domínio.

2. O código deve refletir a linguagem, as regras e as relações entre conceitos, os quais estão presentes nas formas de comunicação entre especialistas do domínio. Deve-se evitar divergências entre o código e a visão do domínio por parte dos especialistas.

Para que essa forma de modelagem seja eficaz, é preciso que os desenvolvedores trabalhem segundo determinados princípios, que são comentados nas seções seguintes.

3.2.1 Processamento do conhecimento

As regras de negócio de um domínio, estejam elas registradas em livros ou no conhecimento tácito dos especialistas, possuem um grau de complexidade que dificilmente pode ser captado em uma única leitura ou conversa. Por isso, o maior desafio em um processo de desenvolvimento de software é a compreensão do domínio por parte dos desenvolvedores. Para se chegar a um bom projeto de software, é preciso conhecer bem os detalhes do problema a fim de elaborar um bom modelo do domínio. No entanto, para se chegar a um modelo maduro e bem representativo, é necessário passar por vários refinamentos, num processo iterativo e ágil.

Evidentemente, não se espera que a equipe de desenvolvimento também se torne especialista no domínio. Mas é preciso que a equipe tenha interesse em conhecer cada vez mais os detalhes e rever suas noções constantemente. Cada novo *insight* pode ser uma oportunidade de melhoria do modelo – e conseqüentemente do projeto e do código. Uma tarefa que os desenvolvedores devem perceber como essencial, portanto, é o processamento do conhecimento (*knowledge crunching*), isto é, a transformação do conhecimento "bruto" proveniente dos especialistas, em um modelo claro, sem ambigüidades, e útil o bastante para ser transformado em código.

Por outro lado, deve haver, também, interesse dos especialistas em compartilhar de forma didática o conhecimento que possuem. Questões culturais podem dificultar essa interação. O exemplo mais evidente é a idéia de que o conhecimento deve ser especializado e separado em áreas funcionais. Mas a dificuldade maior talvez seja a diferença entre a linguagem usada pelo especialista no domínio e aquela empregada pelos desenvolvedores. Os jargões, tão úteis para um grupo, tornam-se termos incompreensíveis para o outro. O resultado é uma comunicação cheia de

ruídos e que normalmente leva a códigos incompreensíveis. O mesmo desenvolvedor que produziu um determinado código pode não ser capaz de entendê-lo seis meses mais tarde, por exemplo.

A solução para esse problema é a prática de uma *linguagem ubíqua*: uma linguagem que esteja presente em todas as formas de comunicação sobre o domínio. Os mesmos termos e conceitos empregados pelos especialistas devem ser aprendidos e exercitados pelos desenvolvedores. Como os métodos ágeis estimulam a comunicação oral, é importante que todos estejam de acordo com a terminologia a ser utilizada. Essa linguagem ubíqua deve aparecer, também, em outros artefatos que a equipe venha a produzir, como diagramas e documentos escritos.

Esses artefatos, inclusive, devem acompanhar a evolução do modelo, estando sempre sincronizados com a linguagem oral dos desenvolvedores e especialistas. É preciso ter em mente, porém, que a produção e a manutenção desses artefatos têm um custo. Artefatos desatualizados, com informações incorretas, têm um custo maior ainda. Logo, é aconselhável que o uso de artefatos desse tipo seja feito com parcimônia. A sugestão de representação do modelo diretamente no código surge dessa constatação.

Por fim, todo o conhecimento e a semântica da linguagem ubíqua devem se refletir no código, a peça mais importante de todo o processo. Nomes de classes e métodos, por exemplo, devem comunicar suas intenções. O emprego da linguagem do domínio é fundamental para atingir um alto nível de expressividade. O refinamento dessa linguagem também é um processo iterativo. Ela acompanha a evolução natural do modelo à medida que os desenvolvedores ganham conhecimentos mais amplos e profundos sobre o domínio.

3.2.2 Modelos ricos em conhecimento

Um dos pontos-chave das linguagens orientadas a objetos é o encapsulamento, tanto de dados quanto de comportamento. Quando esse fundamento é bem usado, aplicando-se princípios de OO, como o de alta coesão/baixo acoplamento (Larman, 2002), é possível alcançar um projeto flexível, extensível e reusável. Esse paradigma favorece a criação de modelos ricos em conhecimento, ou seja, modelos em que os conceitos estão representados no código de maneira legível e, quando possível, abstraídos e generalizados a fim de torná-los fáceis de manter.

Nos modelos anêmicos, o desenvolvimento é orientado somente aos dados e não às operações. Geralmente, a estrutura de classes desses projetos é apenas uma cópia da estrutura relacional presente no banco de dados da aplicação. Os pilares do modelo de objetos, como herança, polimorfismo e encapsulamento, são deixados de lado.

3.2.3 Isolamento do domínio

Considerando que o desenvolvimento deve permanecer focado no modelo do domínio, o primeiro passo deve ser justamente o isolamento dos conceitos que representam o conhecimento

desse domínio. A melhor maneira para alcançar tal isolamento é empregar uma arquitetura multi-camadas.

No caso de desenvolvimento de aplicações corporativas, geralmente são criadas quatro camadas, conforme mostrado na Figura 3.1. A interface com o usuário (também conhecida como camada de apresentação), no topo da pilha, apresenta informações e recebe comandos, que são repassados para a camada de baixo, a de aplicação, que é responsável por coordenar os objetos da camada de domínio a fim de atender aos casos de uso do sistema. A camada de domínio contém as regras de negócio e todo o conhecimento do domínio, representado pelas classes e associações. Por fim, na base da pilha reside a camada de infra-estrutura, que oferece as funcionalidades genéricas que auxiliam as camadas superiores. São tarefas como persistência em bases de dados, serviços de email e comunicação com aplicativos de *backoffice*.

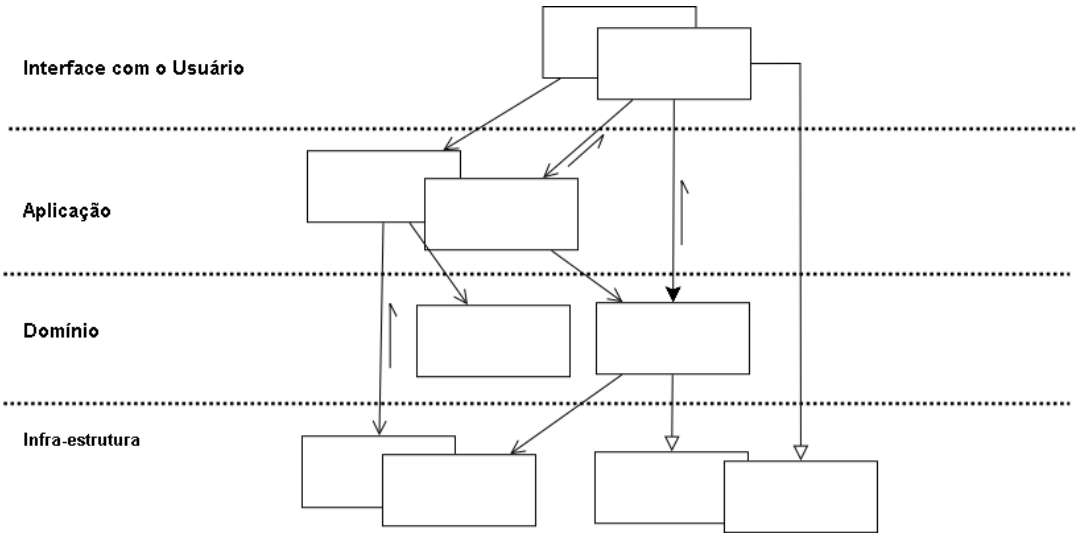


Figura 3.1: Arquitetura em quatro camadas, segundo proposta por Evans (2003).

A vantagem de se usar uma arquitetura multi-camadas é a possibilidade de isolar responsabilidades diferentes em camadas diferentes. O projeto se beneficia, sobretudo, do isolamento da camada de domínio. Isso permite ao desenvolvedor se concentrar nos conceitos do domínio e em como eles se relacionam pelas regras de negócio. O modelo não depende da tecnologia usada para exibir os dados ao usuário, ou do mecanismo usado para armazenar os dados, por exemplo.

Essa arquitetura é bastante semelhante àquela mostrada na seção 2.3.1. Uma das diferenças é a quantidade de camadas propostas para interação com o usuário. O modelo de componentes UML separa em duas camadas, enquanto o de DDD sugere apenas uma. Outra diferença, um pouco mais significativa, concerne à camada de negócios (correspondente à camada de domínio em DDD). Não há uma separação entre os interesses de negócios e os interesses de infra-estrutura, como bases de dados, por exemplo. Os componentes dessa camada apenas encapsulam estruturas de dados e a lógica de persistência dessas estruturas. As regras do domínio estão localizadas na camada superior, separadas dos dados.

Isolar a camada de domínio é essencial para manter a integridade do projeto e o compromisso com um modelo rico em conhecimento. Na implementação, esse modelo precisa ser expresso basicamente em termos de classes e associações. A dificuldade está em definir corretamente essas classes e associações de modo a manter a coerência com os conceitos do domínio. Mais uma vez, os princípios de orientação a objetos podem ajudar bastante na atribuição de responsabilidades.

3.3 Padrões de desenvolvimento em DDD

Dentre os conceitos de DDD, há um conjunto de padrões básicos, que ajudam a organizar o desenvolvimento de aplicações com foco no domínio. Esses padrões possuem significados bem definidos e que enfatizam boas práticas em orientação a objetos. São idéias que auxiliam na tomada de decisões durante a modelagem e o projeto. Cada um desses padrões é detalhado nas próximas seções.

3.3.1 Entidades

Uma das características fundamentais na modelagem de qualquer domínio é o controle e o rastreamento da identidade de alguns objetos. Nesses casos, a identidade deve ser mantida ao longo do tempo, mesmo que os atributos do objeto se modifiquem. Ou seja, a identidade não é definida pelo conjunto dos atributos ou pelo estado do objeto, e sim por uma continuidade temporal. Identidades mal definidas podem levar a inconsistências de dados.

Objetos desse tipo, que precisam ter sua identidade rastreada ao longo do tempo, independentemente dos estados e atributos que possam assumir, são chamados de *entidades*. Considere o conceito de Cliente, por exemplo, em um domínio que envolva transações comerciais. Algumas perguntas ajudam a decidir se Cliente é uma entidade ou não:

- O cliente que está fazendo esta compra é o mesmo cliente que fez uma compra de outro produto no ano passado?
- Uma mudança de endereço, por exemplo, deste cliente afetará o seu reconhecimento em situações futuras?

É fácil perceber que Cliente, neste caso, é uma entidade. Portanto, a implementação da classe Cliente deve focar na identidade e na continuidade do ciclo de vida. Normalmente, entidades são identificadas por números ou cadeias de caracteres, seguindo um determinado padrão na criação e atribuição aos objetos. Qualquer que seja o padrão de identificação utilizado, esse valor deve ser único no domínio. A unicidade deve ser mantida, inclusive em casos de sistemas distribuídos, em que os objetos são serializados por um componente, passam pela rede (codificados em um arquivo XML (*eXtensible Markup Language*), normalmente) e são reconstruídos em outro componente. Perceba-se que os espaços de memória são diferentes e outros aspectos técnicos, como o sistema

operacional, a linguagem e o hardware também podem ser diferentes. No entanto, a identidade do objeto não se altera.

A identidade pode ser atribuída com base em atributos que nunca mudam e que, em conjunto, distinguem um objeto de todos os outros da mesma classe. Quando essa identidade baseada em atributos não é possível, pode-se usar um identificador externo (como um número de CPF, por exemplo), o qual deve permanecer imutável, uma vez atribuído. Em outros casos, o identificador é atribuído automaticamente pelo sistema. Bancos de dados relacionais, por exemplo, já possuem mecanismos de geração automática de IDs para chaves primárias.

Outra questão importante relacionada à identificação refere-se ao que fazer quando uma entidade é excluída do sistema, ou seja, não será mais rastreada de forma alguma. O ID pode ser reusado ou não? Se for reusado, como distinguir a qual das entidades ele corresponde? Todas essas possibilidades devem ser consideradas levando em consideração as características do domínio. As decisões de projeto a respeito de qual esquema de identificação usar devem ser baseadas nessas características.

3.3.2 Objetos de valor

Ao contrário das entidades, alguns conceitos do domínio não possuem identidade conceitual. Eles são caracterizados apenas pelo conjunto dos seus atributos. A implementação desses conceitos em uma linguagem orientada a objetos é feita usando-se *objetos de valor*. Note-se que, do ponto de vista de infra-estrutura, eles podem até ter uma identidade. Por exemplo, o sistema operacional (ou a máquina virtual, o hardware, etc) podem manter um registro da posição de memória que o objeto ocupa, e usar esse valor para manipulações internas. Porém, do ponto de vista do domínio, dois objetos de valor com os mesmos atributos são indistinguíveis e intercambiáveis. Além disso, um objeto de valor também não possui continuidade temporal. Normalmente, eles são criados sob demanda para uma determinada operação e descartados logo em seguida.

Por não possuírem identidade conceitual, objetos de valor podem ser compartilhados livremente. Por exemplo, duas pessoas podem ter o mesmo nome. Isso não faz delas a mesma pessoa, conceitualmente. Porém, os nomes são exatamente os mesmos e, portanto, as duas entidades podem compartilhar o mesmo valor. Outra maneira de implementar esse conceito é copiar os valores, em vez de compartilhá-los. Essa decisão de projeto deve levar em conta questões de desempenho específicas à implementação. Entretanto, se um objeto de valor é compartilhado, ele deve necessariamente ser imutável. Se for necessária alguma alteração, esse objeto deve ser substituído por outro. Além disso, a liberdade de compartilhamento e a simplicidade de implementação dos objetos de valor possibilitam um melhor aproveitamento dos recursos do sistema.

A distinção entre objetos de valor e entidades depende do contexto. Considere-se o exemplo de Endereço. Em um software para uma companhia de vendas por correio, o Endereço é necessário para enviar o pedido. Mas outra pessoa que more na mesma residência pode fazer outro pedido

na mesma companhia. Neste caso, Endereço é um objeto de valor, uma vez que a loja não precisa tomar conhecimento de que as duas entregas referem-se ao mesmo local.

Em outro caso, um software para empresas de correios, cujo objetivo é organizar as rotas de entrega, Endereço é uma entidade, já que possui identidade e continuidade temporal. Cada endereço está associado a um CEP, formado por uma hierarquia de regiões, cidades, zonas postais e blocos, terminando no endereço individual. Uma mudança na organização dessa hierarquia pode até mudar a regra de criação de CEPs, os nomes de ruas e números de imóveis podem mudar, mas o Endereço é conceitualmente o mesmo.

3.3.3 Serviços

Tradicionalmente em orientação a objetos, as classes são usadas para representar estados, no caso de objetos de valor, ou entidades que possuem estado. As operações nesses objetos são codificadas em métodos das próprias classes. Embora essa forma de organização seja altamente desejável, seu mau uso pode provocar o inchamento das classes, tornando-as incompreensíveis e inflexíveis. A causa desse problema é a diminuição na coesão das classes, já que se encarregam de várias tarefas diferentes, muitas delas envolvendo interações com outras classes.

A fim de assegurar alta coesão nas classes, os métodos devem cuidar apenas de operações relacionadas estritamente às classes a que pertencem. Quando surgem operações que não pertencem conceitualmente a nenhuma classe, a solução é tornar o conceito mais explícito, representando a operação em uma classe própria. Tais classes são denominadas *serviços*, e foram inspiradas em um padrão proposto por Larman (2002), denominado Controlador ou Coordenador.

Os serviços devem possuir significado para o domínio, isto é, devem representar operações baseadas nas regras de negócio. Isto implica que os serviços pertencem à camada de domínio e sua função é coordenar as operações que envolvem outros objetos também da camada de domínio. É importante não confundir os serviços da camada de domínio com os serviços de infra-estrutura ou os serviços da camada de aplicação. Por exemplo, uma aplicação bancária pode precisar enviar um aviso por email ao cliente quando seu saldo ficar abaixo do limite. A tarefa de enviar um email deve estar encapsulada por um serviço, que esconda os detalhes da implementação. Porém, esse é um serviço de infra-estrutura, já que não possui nenhuma lógica de negócios associada.

Um bom serviço reúne três características:

1. A operação se relaciona a um conceito do domínio que não é parte natural de nenhuma entidade ou objeto de valor.

2. A interface é definida em termos de outros elementos do modelo do domínio.

3. A operação não guarda estado. Ela pode até consultar e alterar estados de outros objetos ou estados globais do sistema. No entanto, não deve possuir um estado próprio, de modo que uma determinada execução da sua operação não dependa do histórico de operações realizadas pelo serviço.

Vale lembrar que "serviço" é um termo muito sobrecarregado na área de desenvolvimento de software. Não se deve confundir, portanto, o padrão de serviços de DDD com outros conceitos, como o presente na Arquitetura Orientada a Serviços (*Service Oriented Architecture* – SOA, na sigla em inglês) (MacKenzie *et al.*, 2006). No padrão SOA, o foco é encapsular os processos de negócio na forma de serviços, que podem ser acessados por uma gama variada de clientes, de maneira independente da linguagem em que esses clientes estão escritos ou do sistema operacional em que estejam executando. Nesse sentido, os serviços possuem uma granularidade mais alta e o objetivo principal é a troca de dados entre aplicações distintas (possivelmente baseadas em modelos de domínio distintos).

3.3.4 Módulos

À medida que o modelo (sempre em sincronia com o projeto e a implementação) evolui, aumenta o número de conceitos com os quais os desenvolvedores têm que lidar. Conseqüentemente, o número de classes e associações também aumenta. Pensar em todos os conceitos ao mesmo tempo é simplesmente proibitivo, porque a sobrecarga cognitiva que isso implica é muito grande. O limite de conceitos em que uma pessoa consegue pensar simultaneamente é bastante limitado. Logo, surge a necessidade de agrupar as classes em *módulos*, que são blocos lógicos nos quais o conjunto de classes da aplicação é particionado. Linguagens como Java suportam nativamente esse conceito, sob o nome de pacotes.

Como em todos os níveis de abstração em um programa orientado a objetos, o critério de particionamento dos módulos deve também seguir o princípio de alta coesão/baixo acoplamento. Cada módulo deve conter somente classes que estejam semanticamente relacionadas entre si, enquanto módulos diferentes devem corresponder a conceitos diferentes. A interação entre os módulos deve ser pequena, de modo que não exija muito esforço de compreensão do código. Tome-se como exemplo um aplicativo Web de comparação de preços entre várias lojas. Os dois conceitos principais são o de Oferta e o de Produto. Uma Oferta está associada a um Produto e contém dados de venda desse Produto, como preço, promoções, etc. Não é difícil perceber que existe uma forte relação semântica e funcional entre os dois conceitos. Portanto, as classes que os representam ficam melhor posicionadas se estiverem no mesmo módulo.

Módulos fazem parte do modelo e evoluem junto com o conhecimento do domínio e a linguagem ubíqua. No início do desenvolvimento, portanto, o particionamento dos módulos reflete um conhecimento raso e provavelmente errôneo do domínio. Com a aquisição de conhecimentos mais profundos, a equipe de desenvolvimento percebe que existem maneiras melhores de efetuar o particionamento. Ao contrário das classes e métodos, porém, refatorar módulos é uma tarefa bem mais complicada. Essa complexidade provém, na maioria dos ambientes, das ferramentas utilizadas no desenvolvimento, como servidores de controle de versão, IDEs e scripts de instalação e execução de aplicativos, para nomear apenas alguns.

Desse modo, os desenvolvedores tendem a não fazer alterações na estrutura dos módulos. Com o tempo, a falta de refatoração só piora o problema, levando a uma estrutura cada vez mais incompreensível, criando um ciclo vicioso. Em algum ponto é preciso enfrentar o problema e fazer a refatoração necessária. Nesse ponto, o custo da refatoração é muito alto. Portanto, a abordagem mais racional é fazer refatorações constantes e em pequenos passos, mesmo que isso signifique um esforço adicional ao lidar com a tecnologia que apoia o desenvolvimento.

3.3.5 Agregados

Um código orientado a objetos pode ser visto como um grafo de objetos conectados por suas associações. Em geral esse grafo torna-se grande e complexo, resultando em problemas relacionados à propagação das mudanças. Por exemplo, uma Pessoa tem um Endereço associado. Se uma determinada Pessoa é removida do sistema, o que deve acontecer com o Endereço? Se ele também for removido, pode haver corrupção dos dados, uma vez que outra Pessoa poderia estar associada àquele Endereço. Por outro lado, se o Endereço for mantido, corre-se o risco de acumular dados que nunca mais serão usados.

O problema pode até ser tratado com soluções tecnológicas, como coleta de lixo e mecanismos de travamento de linhas e tabelas em bancos de dados, além de controle de transações. Mas essas soluções não atacam a causa do problema, que está em um modelo mal construído. Um investigação mais profunda do domínio pode revelar limites no gerenciamento de objetos e na propagação de mudanças. Esses limites podem ser modelados usando mais um dos padrões básicos de DDD, os *agregados*.

Agregados são grupos de objetos delimitados por uma fronteira e que possuem uma raiz, que deve ser necessariamente uma entidade. Ao definir um limite para a propagação de mudanças, os agregados proporcionam maior eficiência ao sistema (menos objetos precisam ser atualizados) além de assegurar as invariantes semânticas que devem ser mantidas para aquele grupo de objetos.

Invariantes semânticas (ou invariantes de classe) são definidas como "uma propriedade que se aplica a todas as instâncias da classe, transcendendo rotinas particulares" (Meyer, 1992). Invariantes semânticas são a chave para a *programação por contrato*, uma metáfora sobre como os elementos de um programa devem colaborar entre si. Cada operação é vista como um contrato entre duas partes, um cliente e um fornecedor. O fornecedor tem a obrigação de entregar um determinado produto desde que o cliente pague o preço combinado.

Em termos de métodos, essa metáfora pode ser traduzida em termos de pré-condições e pós-condições. Dado um conjunto de parâmetros que satisfaçam determinada condição e asseguradas as invariantes semânticas da classe, o método está obrigado a retornar um valor ou alterar algum estado do sistema, de acordo com uma regra pré-estabelecida. Esse estilo de programação simplifica bastante o código, por evitar validações desnecessárias, além de tornar as operações mais confiáveis.

Para que um agregado proteja os objetos da violação das invariantes semânticas, as seguintes regras devem ser obedecidas na implementação:

- A entidade raiz possui identidade global e é responsável, ao fim das atualizações, por verificar as invariantes.
- Entidades dentro da fronteira do agregado possuem identidade local, única apenas dentro do agregado.
- Nenhum objeto fora da fronteira do agregado deve manter referência a qualquer objeto do lado de dentro, exceto à raiz.
- A raiz pode passar uma cópia de um objeto de valor para outro objeto e não importa o que aconteça a ele, porque trata-se apenas de um valor, que não terá mais nenhuma associação com o agregado.
- Como corolário da regra anterior, apenas raízes de agregados podem ser obtidas diretamente por consultas à base de dados. Todos os outros objetos devem ser encontrados atravessando-se as associações.
- Objetos dentro do agregado podem manter referências a raízes de outros agregados.
- Uma operação de exclusão deve remover todos os objetos do agregado de uma só vez.
- Quando uma mudança é efetuada em qualquer objeto dentro da fronteira do agregado, todas as invariantes do agregado como um todo devem ser satisfeitas.

De volta ao exemplo apresentado no começo desta seção, Pessoa, que é uma entidade, seria a raiz de um agregado contendo (entre outros possíveis objetos) um ou mais Endereços, que são objetos de valor. O controle do ciclo de vida dos Endereços ficaria por conta da classe Pessoa, que também cuidaria de verificar as invariantes semânticas do agregado. Se uma determinada Pessoa é removida do sistema, essa mudança deve necessariamente se propagar para todo o agregado.

Dependendo da implementação do objeto de valor Endereço, essa propagação pode ser feita removendo-se a associação entre os objetos (no caso de um objeto compartilhado), ou removendo-se o próprio objeto, no caso de cópias. Seja qual for a implementação, o importante é que o conceito de agregado garante o encapsulamento das mudanças. Assim, nenhuma informação será perdida ou ficará registrada inutilmente no sistema.

3.3.6 Fábricas

Em um artigo sobre o pensamento científico, Dijkstra (1982) cunhou o termo “separação de interesses”. A expressão refere-se ao modo como se deve organizar o raciocínio em uma atividade científica: os vários aspectos de um determinado problema devem ser estudados isoladamente. Por

exemplo, em relação a um programa, pode-se estudar sua eficiência, sua correção ou sua utilidade. O fato é que ao estudar um desses aspectos, os demais devem ser considerados irrelevantes e merecem ser ignorados.

Embora a preocupação de Dijkstra fosse basicamente filosófica, o princípio de separação de interesses recebeu uma grande aceitação entre os projetistas de software. No contexto de desenvolvimento de software, esse princípio sugere que tarefas diferentes devem estar implementadas em pontos diferentes do código, com o mínimo possível de acoplamento entre eles (o que, na prática, é bem mais difícil do que em teoria).

Restringindo esse princípio a um contexto ainda mais específico, as responsabilidades de um objeto devem estar desacopladas das responsabilidades de manutenção do seu ciclo de vida. Isso compreende a criação, modificação, armazenamento e possivelmente, exclusão do objeto, como mostrado na Figura 3.2. A manutenção desse ciclo de vida, no entanto, é uma tarefa totalmente dissociada das responsabilidades dos próprios objetos. Em outras palavras, um objeto não deve conhecer os detalhes da sua própria criação, armazenamento em bases de dados, envio pela rede, etc. Essa acumulação de tarefas em um objeto tornaria sua implementação bastante complexa, comprometendo a facilidade de manutenção e o desempenho desse objeto.

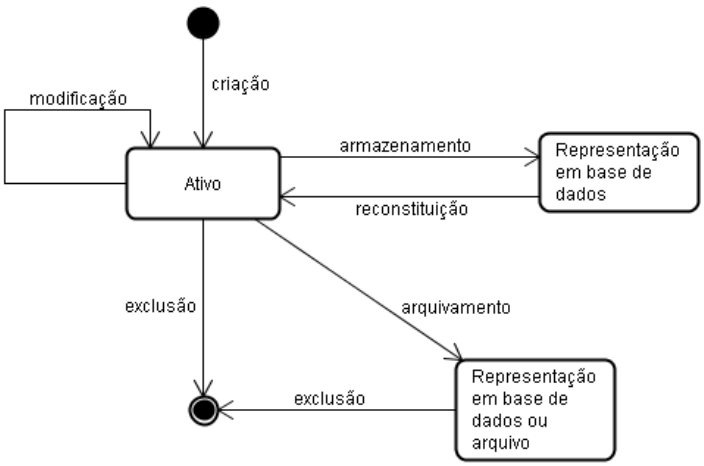


Figura 3.2: Ciclo de vida de um objeto (Evans, 2003).

Conforme discutido anteriormente nesta seção, os agregados garantem a manutenção das invariantes de um conjunto de objetos semanticamente relacionados. Embora durante uma operação de modificação essas invariantes possam ser temporariamente violadas, elas são asseguradas no final. Observe-se que essa propriedade dos agregados pressupõe que as invariantes já estejam asseguradas antes das atualizações. Remontando a seqüência de alterações no objeto, no caminho inverso, percebe-se que é necessário que as invariantes já estejam asseguradas no momento da criação. Como assegurar isto?

Uma solução bem direta é implementar essas regras nos construtores dos objetos (ou a construção sintática equivalente, na linguagem em que se estiver trabalhando). Construtores, no entanto, devem ser extremamente simples. Quando as invariantes começam a ficar muito complexas ou

quando elas envolvem vários objetos diferentes, os construtores não são mais adequados. A lógica de criação de agregados e manutenção de invariantes deve ser encapsulada em outro lugar, as *fábricas*.

Em DDD, a metáfora das fábricas está intimamente relacionada às suas contrapartes reais. Ambas são responsáveis pela tarefa relativamente complexa de construir um determinado produto. No âmbito da orientação a objetos, esses produtos são os agregados, com suas associações internas já definidas e as invariantes semânticas asseguradas.

A implementação das fábricas pode ser feita de várias maneiras. Os padrões de projeto criacionais, propostos inicialmente pela GoF (*Gang of Four*, nome dado ao grupo de criadores dos padrões de projeto de software) (Gamma *et al.*, 1995), ajudam bastante nessa tarefa. Entre eles, estão o método fábrica (*factory method*), a fábrica abstrata (*abstract factory*) e o construtor (*builder*). Assim, as fábricas podem ser implementadas como um método da raiz de um agregado, um método de outro objeto que esteja altamente relacionado ou como uma classe separada. Essa é uma decisão de projeto que deve levar em conta o nível de acoplamento entre a fábrica e o produto.

É importante ressaltar que as fábricas não possuem nenhum significado para o modelo. A criação de um objeto ou agregado geralmente não tem nenhum significado em termos de regras de negócio. São apenas uma necessidade de implementação. Ainda assim, as fábricas pertencem à camada de domínio, pelo fato de estarem fortemente acopladas com os seus produtos e por conhecerem as invariantes semânticas que devem ser mantidas.

Duas diretrizes são fundamentais na implementação de uma fábrica:

1. As operações de criação de objetos devem ser atômicas. Todas as informações devem ser passadas de uma única vez, para que a fábrica seja capaz de criar o agregado em apenas uma interação com o cliente. Isso evita possíveis efeitos colaterais e facilita a afirmação das invariantes.

2. A fábrica deve estar desacoplada dos argumentos. Preferencialmente, as interfaces de criação devem depender de tipos abstratos e não de classes concretas. A escolha dos argumentos também é importante. Sempre que possível, recomenda-se que se passem objetos que já estejam inerentemente acoplados aos produtos. Exemplos desses argumentos são dependências do produto, que serão injetadas pela fábrica.

3.3.7 Repositórios

Há três maneiras de se obter referência a um objeto:

1. Criar o objeto, usando seu construtor ou uma fábrica,

2. Navegar pela associação de outro objeto que tenha uma referência ao objeto desejado, ou

3. Fazer uma busca no mecanismo de armazenamento de dados pelo objeto desejado.

A criação de um objeto corresponde ao início do seu ciclo de vida, conforme discutido na seção 3.3.6. Muitas vezes, porém, o objeto já existe (foi criado anteriormente) e o que se pretende é acessá-lo. Para tanto, os modos de acesso citados nos ítens 2 e 3 acima são adequados. Embora ambas as operações sejam computacionalmente custosas, a do item 2 é especialmente problemática. Quando o grafo de objetos começa a crescer demasiadamente, o gasto de memória e o tempo de acesso tornam-se impraticáveis. Por esse motivo, inclusive, é que foram introduzidos os agregados.

Travessias em associações de objetos, entretanto, têm seus limites (além da demanda de recursos). Mesmo que se queira alcançar todos os objetos atravessando associações, como conseguir a referência ao primeiro objeto? A solução é o terceiro modo de acesso: busca em mecanismos de armazenamento para os quais as bases de dados relacionais ainda são o padrão *de facto* da indústria de software. Por esse motivo, este texto trata somente de bases de dados relacionais para discutir armazenamento de dados.

A adoção em larga escala desse tipo de base de dados deve-se, em parte, à eficiência desses sistemas. Nas últimas décadas, um grande volume de trabalho em Pesquisa e Desenvolvimento foi dedicado a essa tecnologia. Atualmente os Sistemas Gerenciadores de Bancos de Dados relacionais são a melhor alternativa para ambientes que requeiram a manipulação de enormes volumes de dados em um tempo razoável.

Por outro lado, o modelo relacional oferece alguns obstáculos ao DDD. Ao contrário do modelo orientado a objetos, o modelo relacional não abriga conceitos como polimorfismo, herança e encapsulamento de dados. Assim, não é possível representar diretamente esses conceitos de objetos em bases de dados. No sentido contrário, ter uma classe para cada tabela, embora possível, é uma prática desaconselhável, por tornar o modelo anêmico.

Como as aplicações normalmente lidam com armazenamento e recuperação de informações, esses dados acabam tendo que passar de um modelo OO para um relacional (e vice-versa) muito freqüentemente. Dado que esses modelos não são equivalentes, é preciso "traduzir" os dados de um modelo para outro, toda vez que se busca ou atualiza um objeto na base. Cada linguagem possui bibliotecas que oferecem utilitários básicos para essa conversão (como a JDBC para Java). No entanto, a manipulação dos detalhes de conversão por objetos do modelo é desencorajada, pois o desenvolvimento e a manutenção de códigos SQL é difícil e consome muito tempo. Além disso, embora a linguagem SQL seja totalmente portável, muitos gerenciadores de bancos de dados não seguem o padrão fielmente, o que levou ao surgimento de vários "dialetos" da linguagem, tornando a portabilidade, na prática, bastante difícil. Outra conseqüência dessa abordagem é o acoplamento desnecessário das regras de negócio com os detalhes de infra-estrutura, o que é uma clara violação do princípio básico de isolamento do domínio.

A fim de solucionar esse tipo de problema, o uso de *repositórios* foi proposto primeiramente por Fowler (2003b) e depois por Evans (2003). Repositórios são padrões cujo objetivo é emular uma coleção de dados (de um determinado tipo) em memória, com potencialidades de busca mais sofisticadas. Os detalhes de implementação são escondidos e as classes clientes que necessitem

fazer uso de persistência não tomam conhecimento dos detalhes de mais baixo nível envolvidos na manipulação dos dados. Para busca, os repositórios podem expor métodos específicos para determinadas consultas mais freqüentes. Podem também receber como parâmetro objetos que representem uma determinada especificação (Evans e Fowler, 1997).

Atualmente existem vários *frameworks* de persistência, como o Hibernate (Bauer e King, 2005) e o TopLink Essentials (Dub *et al.*, 2003), que auxiliam na implementação dos repositórios. Esses *frameworks* provêem várias funcionalidades além do mapeamento objeto-relacional, propriamente dito. Entre essas funcionalidades estão o suporte a transações, uma linguagem de consulta de objetos (e não de tabelas como SQL), carregamento ávido (*eager loading*) e preguiçoso (*lazy loading*)[1], *caching*, entre outros (Richardson, 2006).

3.4 Trabalhos relacionados

Desde que foram propostas, as técnicas de DDD têm sido aplicadas com sucesso em vários projetos. Hu e Peng (2007), da Custom House Global Foreign Exchange, demonstraram o uso de DDD no domínio do mercado cambial. Nesse domínio há quatro conceitos centrais: Dinheiro, Moeda, Taxa de Câmbio e *Markup* (um conceito semelhante ao lucro em transações comerciais). Ao explicitar esses conceitos no modelo do domínio, a facilidade de manutenção e compreensão do código aumentaram sensivelmente.

Grandes projetos também podem se beneficiar da aplicação de DDD. Um exemplo é o sistema de gerenciamento de venda e distribuição da Statoil ASA, empresa norueguesa de exploração de gás e petróleo (Landre *et al.*, 2006). Considerando que esse domínio é bastante extenso e com muitas regras de negócio, o modelo foi dividido em vários sub-modelos, cada um restrito a um contexto específico. Como todo o processo é englobado nesse modelo, é possível conduzir o projeto no nível estratégico da empresa, e não somente técnico. A complexidade, por outro lado, é bem maior. Em casos como esse, é imprescindível definir fronteiras claras entre os diversos contextos.

Embora o desenvolvimento de novos sistemas seja o foco das idéias e padrões do DDD, ele também pode ser usado para avaliar soluções comerciais já prontas (*off-the-shelf*). Essa experiência foi relatada por Wesenberg *et al.* (2006), também na Statoil ASA. O foco no domínio permitiu aos analistas identificarem os produtos que proporcionavam a maior cobertura funcional, ou seja, o quanto os candidatos cobriam as necessidades de negócio da empresa. O modelo de informação também pôde ser avaliado com maior precisão. Em outras palavras, havia uma base mais sólida para avaliar se os candidatos possuíam a informação necessária corretamente estruturada. Esse ponto é de extrema importância na aquisição de uma solução pronta, já que a empresa compradora

[1]Esses dois termos referem-se ao momento em que um objeto é carregado. No carregamento ávido, sempre que um objeto A é carregado em memória, todos os objetos associados a A são carregados junto. No carregamento preguiçoso, os objetos associados a A somente são carregados quando necessário, isto é, quando houver uma demanda por esses objetos, por meio de suas referências.

adquire apenas o produto e não tem a oportunidade de destilar o conhecimento do domínio durante o desenvolvimento.

Em geral, quando uma empresa decide desenvolver (ou contratar o desenvolvimento) de um sistema, essa solução não é construída do zero. A integração com sistemas legados costuma ser bastante comum. Se, por um lado, isso traz vantagens no reúso de idéias e partes de software, por outro sempre há problemas de incompatibilidade entre o novo sistema e o antigo. Por isso, é importante definir bem as fronteiras e criar camadas que evitem a corrupção do modelo. Peng e Hu (2007) implementaram uma camada desse tipo, demonstrando como ela é eficaz na manutenção da integridade do modelo do domínio.

No desenvolvimento de aplicações que manipulam grandes volumes de dados, como as aplicações corporativas, um dos principais desafios é a implementação da camada de persistência. Esse desafio é ainda maior quando se trata do problema do mapeamento objeto-relacional, quando há dois paradigmas diferentes de modelagem coexistindo no sistema. Nos últimos anos, várias soluções tecnológicas foram propostas para resolver esse problema, como os *frameworks* de persistência. Visando a dar uma solução mais geral, Witthawaskul e Johnson (2004) propuseram um modelo de persistência independente de plataforma, com forte embasamento teórico em DDD.

Na mesma linha de mapeamento objeto-relacional (porém com uma perspectiva prática), foi feito um estudo (Yemelyanov, 2008), cujo objetivo é formular um guia sobre a aplicação de DDD em conjunto com a tecnologia Entity Framework, da Microsoft (Blakeley *et al.*, 2006). Os resultados foram aplicados aos processos de desenvolvimento iterativo de software da Volvo IT.

Outro exemplo de aplicação de DDD é reportado por Landre *et al.* (2007), em um projeto de desenvolvimento de aplicações de comércio e fornecimento de petróleo e produtos refinados. O objetivo era substituir os sistemas legados da empresa por novas aplicações, as quais deveriam ser capazes de lidar com computações complexas sobre grandes volumes de dados. Nesse projeto, foi feita uma prova de conceito com o objetivo de avaliar os resultados de um desenvolvimento baseado em DDD e métodos ágeis. O subconjunto do domínio escolhido para essa prova de conceito foi o cálculo de preços de produtos.

Os resultados reportados foram bastante satisfatórios. O contato permanente com dois especialistas no domínio possibilitou a construção de um modelo a partir de conceitos complexos, que foram emergindo aos poucos. O exercício da linguagem ubíqua foi um fator decisivo na elaboração desse modelo. Os autores observaram que mesmo após 10 meses, o código ainda comunicava claramente o intuito dos desenvolvedores e as razões de negócio subjacentes, sem necessidade de documentação adicional. O isolamento da camada de domínio simplificou bastante a implementação do sistema, além de torná-lo mais resiliente, o que foi percebido quando houve a necessidade de troca do mecanismo de persistência.

3.5 Comparações com MDD e MDA

Quando se comenta sobre DDD, é preciso destacar a diferença entre este conceito e o de MDD (*Model Driven Design*). Como os nomes são parecidos, e pelo fato de os dois conceitos caminharem juntos, é comum a confusão entre eles. Contudo, são duas maneiras diferentes de abordar o desenvolvimento de software, de modo que uma reforça a outra.

DDD parte da premissa de que o desenvolvimento de software deve ser focado no conhecimento do domínio, e que toda a infra-estrutura necessária deve ser escolhida (ou desenvolvida) para suportar essa representação do domínio ou *modelo de domínio*. Segundo Evans (2003), "um modelo é uma forma de conhecimento seletivamente simplificada e conscientemente estruturada". Neste contexto, MDD é um projeto em que algum subconjunto dos elementos de software está intimamente relacionado aos elementos do modelo. É também um processo de alinhamento da implementação com o modelo, de modo que ambos evoluam em sincronia.

O uso de modelos, por outro lado, não está restrito a representações de domínio. Uma API para interface gráfica, por exemplo, pode ser construída com base em um modelo (usando metáforas como janela, caixa e botão). Esse é um exemplo de MDD aplicado à infra-estrutura e não ao domínio.

Contudo, a sigla MDD pode assumir um significado diferente: a geração automática de programas a partir de modelos mais abstratos que as linguagens de programação atuais. Essa elevação no nível de abstração seria correspondente ao passo que as chamadas linguagens de programação de terceira-geração deram em relação à linguagem de montagem. A fim de dar apoio ao desenvolvimento de sistemas utilizando o paradigma de MDD, o Object Management Group definiu um conjunto de especificações (Miller *et al.*, 2003) cujo objetivo é separar completamente a definição da arquitetura de um sistema da sua implementação. Esse conjunto de especificações recebeu o nome de MDA (*Model Driven Architecture*).

Um dos conceitos definidos em MDA é o de Modelo Independente de Plataforma (PIM, na sigla em inglês), por meio do qual o desenvolvedor pode definir um modelo de negócios e de sistema sem se preocupar com a plataforma que será usada para implementá-la. Esse modelo deve ser transformado, posteriormente, em um Modelo Específico para Plataforma (PSM, na sigla em inglês). Essas transformações podem ser feitas, teoricamente, por ferramentas de geração automática de código.

3.6 Métodos ágeis

Conforme discutido na seção 2.2.1, o fluxo de atividades e as fases de desenvolvimento no ESPLEP são combinadas de modo mais ou menos paralelo, e não ortogonal, como sugere o USDP. Em particular, vale notar que a implementação, propriamente dita, da linha de produtos é deixada para a penúltima fase. Mesmo nessa fase de implementação, há uma serialização das atividades:

primeiro implementa-se o núcleo, que é entregue todo de uma vez e em seguida, implementam-se iterativamente as variabilidades. A iteratividade, neste caso, consiste em alternar apenas as fases de Construção e Transição.

Há um pressuposto específico nessa alternância entre Construção e Transição: o de que todos os artefatos produzidos nas fases anteriores são suficientes para embasar o desenvolvimento nesta etapa. Revisões e alterações no que foi produzido nas fases de Requisitos, Análise e Projeto são feitas somente após os testes, na última fase. Um pressuposto mais geral, que se aplica a todo o processo, é o de que cada atividade deve ser exaustivamente trabalhada antes de dar início à próxima. Esse princípio é conhecido como "processo em cascata" e apresenta problemas sérios, conforme apontado por diversos autores.

Yourdon (2004), por exemplo, fez um estudo sobre projetos de software que desviam dos valores esperados em mais de 50%. Esses valores correspondem a estimativas de tempo, orçamento, pessoas e escopo. O autor denomina esses projetos de "marcha para a morte", já que há uma grande probabilidade de fracasso nesse tipo de projeto. O estudo mostra que por trás dessas estimativas ingenuamente otimistas há a idéia de processos em cascata.

O principal argumento a favor dos processos em cascata fundamenta-se sobre a questão da redução de riscos. Em tese, quanto mais cedo os riscos forem eliminados, maior será a chance de sucesso de um projeto. Riscos, neste caso, incluem riscos de não-aceitação do produto no mercado, inadequação às necessidades dos usuários, atraso na entrega, estouro de orçamento, defeitos no produtos final, etc.

Embora seja altamente desejável que se eliminem os riscos do projeto tão logo quanto possível, o que se observa no exame dos fatos é que existe uma dificuldade muito grande de tornar isso possível. No início de um projeto, fatores como desconhecimento da tecnologia a ser utilizada, níveis de habilidade dos desenvolvedores, obstáculos burocráticos, entre outros problemas, tornam a incerteza razoavelmente alta. Além disso, é natural que o escopo planejado no início de um projeto se altere ao longo do tempo. Isso ocorre em virtude do desconhecimento das reais necessidades dos usuários e clientes que se beneficiarão do software. Os próprios clientes não as conhecem por completo no início e, muitas vezes, têm *insights* somente após o contato com o produto, ainda que em uma versão inicial.

Cohn (2005) compara as abordagens de tentativa de redução de riscos do modo como são feitas (ou, pelo menos, tentadas) em projetos baseados em cascata com projetos ágeis. Os gráficos hipotéticos da Figura 3.3 resumem as duas abordagens. Os processos em cascata tentam reduzir a zero o risco no tempo 0 do projeto. Processos ágeis, por outro lado, procuram reduzi-lo a uma taxa aproximadamente logarítmica.

Cockburn (2002) apresenta os motivos culturais e psicológicos subjacentes à adoção de processos em cascata por gerentes de projeto, mesmo havendo um crescente número de evidências contrárias a esse tipo de processos e favoráveis ao desenvolvimento incremental e iterativo. No livro, o autor cita o trabalho de Piatelli-Palmerini (1996), que mostra estatisticamente que as pessoas,

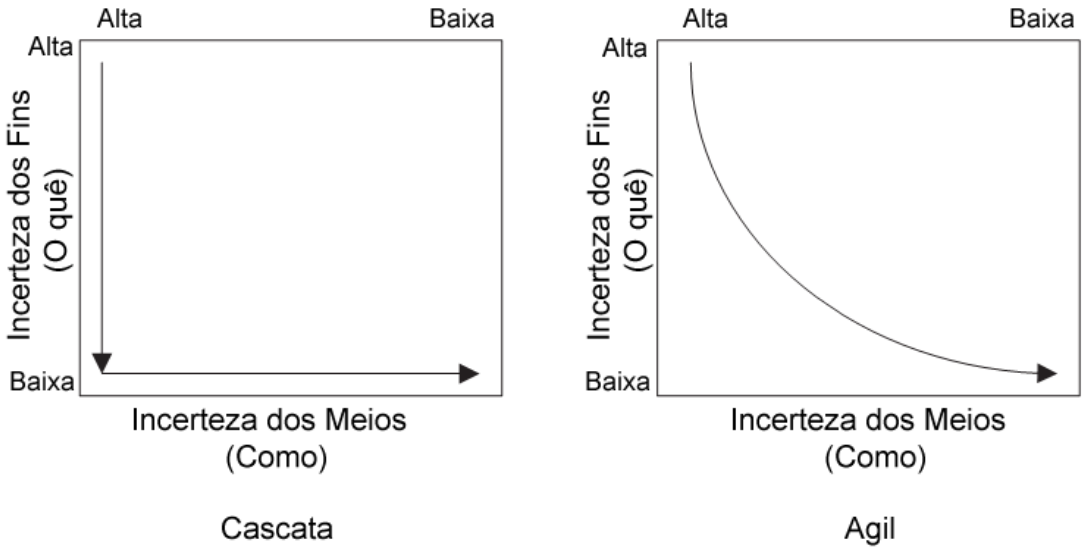


Figura 3.3: Diferentes abordagens para redução de riscos em projetos de software. (Cohn, 2005).

em geral (e os gerentes de projeto, em particular) são avessas a riscos quando existe a possibilidade de ganho, mas assumem os mesmos níveis de risco quando há a possibilidade de perda de algum bem.

Diante da possibilidade de mudar de um processo em cascata para algum método ágil, a tendência dos gerentes de projeto é adotar uma postura conservadora, mesmo sabendo que métodos ágeis possam trazer melhores resultados. Ou seja, prefere-se não correr o risco da mudança nas situações em que a recompensa, em caso de sucesso, fosse vantajosa.

Larman (2004) analisa mais detalhadamente os problemas presentes nos processos em cascata e lista os seguintes:

Especificações "completas" e precoces, com aprovação formal: O autor apresenta dados que mostram que especificações que permanecem inalteradas do início ao fim de um projeto são raras. Dados ainda mais alarmantes revelam que 45% das funcionalidades criadas a partir de especificações precoces nunca são usadas e 19% raramente são usadas.

Teste e integração tardios: Com essa abordagem, os problemas mais sérios somente são encontrados ao final do projeto. Testes executados ao longo do projeto e integração contínua ajudam a identificar os problemas mais cedo.

Estimativas e cronogramas "confiáveis": Segundo o autor, a forma de planejamento prognóstico usada nos modelos em cascata é mais adequada a processos de manufatura, em que o trabalho é repetitivo e bem conhecido. Desenvolvimento de software, por outro lado, é um trabalho inventivo, em que os requisitos são, por natureza, pouco conhecidos no início.

Relação entre planejamento e execução: Seguir um plano rígido, definido no início do projeto é uma prática que não funciona bem em desenvolvimento de software. Dada a incerteza do

trabalho (em virtude de sua natureza inventiva, conforme citado no item anterior), é melhor ter um plano flexível e reativo às mudanças que normalmente ocorrem ao longo dos projetos.

A fim de propor uma solução a esses problemas observados nos métodos tradicionais, foram criados vários outros métodos, baseados em um outro conjunto de valores. A maioria desses métodos de desenvolvimento de software foi criada e inicialmente implantada na década de 1990. Em 2001, os proponentes desses métodos se reuniram e redigiram o manifesto ágil, no qual propõem valores e princípios a serem adotados em projetos de desenvolvimento de software. Os quatro valores do manifesto são listados abaixo:

- Indivíduos e interações são mais importantes que processos e ferramentas.
- Software funcionando é mais importante que documentação abrangente.
- Colaboração com o cliente é mais importante que negociação de contrato.
- Responder a mudanças é mais importante que seguir um plano.

No estudo de caso apresentado no Capítulo 4, foram utilizadas duas técnicas comuns aos métodos ágeis: histórias de usuário e desenvolvimento iterativo. Ambas são exploradas na próxima seção. Em seguida, são analisados resumidamente os métodos Scrum e XP, por serem os mais difundidos. Segundo o último relatório anual sobre desenvolvimento ágil (VersionOne, 2008), das 2319 empresas consultadas (as quais já utilizam métodos ágeis) em 80 países, 71% adotam um desses dois métodos ou um híbrido dos dois.

3.6.1 Técnicas comuns

Histórias de usuário

Nos métodos tradicionais de desenvolvimento de software, a atividade de levantamento de requisitos geralmente é feita antes de todas as outras atividades e pretende ser exaustiva. Por isso, bastante tempo é gasto na formalização desses requisitos antes do início da implementação. Em geral, o principal tipo de documentação produzido nesta atividade são os casos de uso, embora alguns outros formatos tenham sido propostos na literatura, como o padrão IEEE 830 (IEEE, 1998).

Uma das conseqüências dessa especificação prematura dos requisitos é que os casos de uso produzidos tendem a ser muito grandes, com o intuito de prever todos os cenários possíveis de interação dos usuários com o sistema. A quantidade de casos de uso também tende a ser grande, dependendo da complexidade do sistema. Além disso, os diversos casos de uso possuem relacionamentos entre si, como extensão e inclusão. Por isso, torna-se necessária a elaboração de diagramas de casos de uso, de modo a "mapear" todos esses relacionamentos.

O grande volume de documentação dos requisitos torna a sua manutenção onerosa. A cada alteração no sistema, é preciso atualizar todos os cenários modificados. Um caso de uso desatualizado, inclusive, pode ser pior que a inexistência de um caso de uso para aquela funcionalidade, podendo confundir os desenvolvedores e criar uma espécie de dissonância cognitiva entre o comportamento observado do sistema e o comportamento especificado na documentação. Cohn (2004) aponta alguns outros problemas:

1. É difícil priorizar as funcionalidades de um sistema, em razão do tamanho dos casos de uso e da complexidade das interações entre eles.

2. Em projetos com um cronograma muito apertado, o tempo gasto na escrita de casos de uso pode ser relativamente muito grande.

3. Casos de uso desencorajam a participação dos *stakeholders* ao longo do projeto. Como os casos de uso geralmente são escritos logo no início e representam uma formalização dos requisitos, existe a tendência de usá-lo como referência, mais do que as respostas dos clientes ou usuários.

Por outro lado, é importante ressaltar que apesar dessas desvantagens observadas nos casos de uso, eles podem ser usados de maneira mais interativa e com menos sobrecarga, em termos de tempo e esforço. Essa é a abordagem sugerida por Cockburn (2001), por exemplo. Porém, a proposta deste trabalho é estudar o desenvolvimento de linhas de produtos de software com base em métodos ágeis e DDD. Por isso, adotou-se a técnica de gerenciamento de requisitos e funcionalidades mais defendida pelos proponentes de métodos ágeis: as histórias de usuário (*user stories*) (Cohn, 2004).

Histórias de usuário são sentenças sucintas, escritas em formato livre pelo próprio usuário ou cliente, sobre funcionalidades (*features*) desejadas no sistema. O significado do termo "funcionalidade" é o mesmo daquele usado na discussão sobre o ESPLEP, na seção 2.2.1: uma unidade básica dos requisitos da linha de produtos.

Apesar de não haver um formato definido, as histórias são normalmente compostas por três elementos: 1) um papel, isto é, algum usuário ou envolvido que se beneficie dessa funcionalidade de alguma forma. 2) um resumo da funcionalidade e 3) o valor de negócios que tal funcionalidade trará ao envolvido. Por exemplo, em um sistema de comparação *on-line* de produtos, uma das funcionalidades pode ser descrita pela seguinte história:

Como comprador, quero ver uma lista de preços anunciados para um determinado produto, para poder comprar da loja que oferecer o preço mais baixo.

Neste caso, o usuário é o potencial comprador, a funcionalidade é comparar preços de produtos e o valor de negócios do qual o usuário se beneficiará é a possibilidade de comprar o produto da

loja que oferecer o menor preço. É importante lembrar que o escopo de uma história de usuário é diferente do escopo de um caso de uso. As histórias são apenas uma visão geral sobre a funcionalidade e não pretendem ser uma descrição do comportamento do sistema. Seu principal objetivo é ser um lembrete sobre o requisito e sua importância.

Sendo apenas uma visão geral do requisito e não uma descrição, como um caso de uso, cada história precisa vir acompanhada de detalhes sobre o comportamento esperado do sistema para o requisito em questão. Esses detalhes são os chamados testes de aceitação, que também são sentenças curtas e informais, cada uma mencionando resumidamente o que se espera do sistema em um cenário diferente de interação.

Além de servirem como um conjunto de critérios que a história deve satisfazer para ser considerada completa, esses testes de aceitação informais podem ser usados como base para a criação de testes de aceitação automatizados. Tais testes podem, inclusive, ser escritos antes da própria história ser implementada. Esse processo é conhecido como desenvolvimento guiado por testes de aceitação (*acceptance test-driven development*, ou ATDD). Uma descrição aprofundada sobre ATDD e as técnicas envolvidas são dadas por Koskela (2007).

Na história recém-citada, os testes de aceitação são os seguintes:

- Alterar o tamanho da lista de anúncios, usando uma caixa de combinação, que deve estar presente na interface.
- Verificar se todos os anúncios possuem, além do preço, o nome da loja.
- Clicar em um anúncio. O sistema deve redirecionar para o *site* da loja.
- Verificar se os anúncios estão listados em ordem crescente de preço.
- Navegar pelo controle de paginação, que deve estar presente na interface, quando a quantidade de anúncios ultrapassar o tamanho da lista.

Para cada funcionalidade que se pretende implementar, deve ser escrita uma história de usuário. Todas essas histórias são incluídas em uma lista, denominada *backlog* do produto. Conforme já comentado, é fundamental que as histórias de usuário sejam escritas com uma razão de negócios em mente. Contudo, nem todas as histórias proporcionam o mesmo valor para o usuário. Conseqüentemente, é preciso considerar as relações de prioridade e posterioridade de cada história em vista às demais. Por esse motivo, a ordem das histórias em um *backlog* é importante: elas devem estar ordenadas por valor de negócio, do mais importante para o menos importante.

O chamado "valor de negócio" é um termo genérico usado para designar os benefícios proporcionados por um conjunto de funcionalidades de um software. Tais benefícios podem se manifestar de diversas maneiras. Apenas para citar alguns exemplos:

- Redução de tempo para execução de uma tarefa anteriormente manual, automatizada pelo software.

- Acesso a um determinado conjunto de informações úteis ao usuário.
- Ganho financeiro, em função de tarifação de uso, por exemplo.
- Gerenciamento de atividades muito complexas, que não poderiam ser feitas manualmente.
- Facilidade de comunicação.

Muitos podem ser os beneficiários do valor de negócio de um software: os usuários finais, empresas que o utilizem corporativamente, empresas que façam tarifação no uso desse software, órgãos governamentais, empresas terceiras, etc. Geralmente, cada um desses grupos de envolvidos se beneficia de modo diferente das funcionalidades oferecidas pelo software, o que torna o processo de priorização bastante complicado. Por isso, o ideal é que a priorização seja feita por algum desses envolvidos (de preferência a parte que contratou o desenvolvimento e entende suas necessidades) ou por um especialista no domínio representando os envolvidos.

Em um processo ágil de desenvolvimento, é escolhido um subconjunto de funcionalidades a se implementar a cada iteração. Com as histórias priorizadas por valor de negócio, a tarefa de selecionar esse subconjunto é simples: basta escolher as primeiras histórias da lista. A quantidade de histórias efetivamente escolhida dependerá de vários fatores: o tamanho e as habilidades da equipe de desenvolvimento, a complexidade de cada história, prazos, recursos à disposição etc.

Diferentemente dos métodos tradicionais de levantamento de requisitos, o *backlog* do produto é dinâmico. Não é necessário que se escrevam todas as histórias no começo do projeto. À medida que o sistema evolui e novos conhecimentos são adquiridos, histórias podem ser acrescentadas, removidas ou alteradas. Do mesmo modo, a ordenação das histórias pode sofrer alterações. Em contraste com os métodos tradicionais, esse modo de definição de requisitos permite maior adaptabilidade às mudanças de negócio e de entendimento dos envolvidos.

Desenvolvimento iterativo

Pelo que foi exposto na seção anterior, percebe-se que os métodos tradicionais enfatizam a especificação exaustiva dos requisitos antes que as outras atividades se iniciem. O resultado é um conjunto grande de documentos de casos de uso ou outros tipos de documentos, ainda mais formais, tais como os propostos pelo padrão IEEE 830, por exemplo.

O mesmo vale para as outras atividades, como análise e projeto. Os métodos de desenvolvimento de linhas de produtos discutidos no Capítulo 2 partem da idéia de que as atividades de análise, projeto, implementação e testes são independentes e podem ser feitas de modo seqüencial. Conseqüentemente, o mesmo volume de documentos que se observam em relação a requisitos podem ser constatados também em relação às outras atividades.

Desse modo, a implementação e o teste das aplicações são relegadas a segundo plano. Isso pode ser percebido pela diferença de tratamento dada a cada atividade pelos proponentes dos métodos

tradicionais. Uma grande ênfase é dada às atividades geradoras de artefatos não-executáveis (como documentos de especificação e diagramas UML) ao passo que muito pouco se discute sobre as atividades relacionadas a código executável, seja a escrita de código de produção, automação de testes em vários níveis ou execução manual de testes sobre uma aplicação entregável.

Essa diferença de ênfase em relação às diferentes atividades apóia-se na premissa de que os riscos, em projetos de desenvolvimento de software, podem ser mitigados utilizando-se especificações textuais ou diagramáticas. De posse dessas especificações, os desenvolvedores seriam capazes de entender — com grande acuidade — tanto o problema quanto a solução.

Um corolário dessa premissa é o de que a atividade de implementação é relativamente mecânica, isto é, uma vez que todos os requisitos e a arquitetura tenham sido especificados, o mapeamento para código executável é direto e exige pouca elaboração. Analogamente, o teste seria a atividade menos importante, uma vez que todas as outras atividades já teriam reduzido os riscos a um nível mínimo.

Conseqüentemente, há uma tendência a serializar o processo, de modo que cada atividade possa ser feita com base nos artefatos produzidos pela atividade anterior. Essa serialização implica que o produto somente será entregue para testes de aceitação pelo usuário final após um longo tempo de desenvolvimento. Paradoxalmente, essa tentativa de mitigar os riscos logo no início do projeto por meio de requisitos detalhados é o principal fator causador de falhas em projetos de software, segundo Thomas (2001).

Isso ocorre justamente porque na maioria das vezes é muito difícil reduzir os riscos de um projeto por meio de especificações não-executáveis detalhadas, produzidas em um estágio muito inicial do projeto. Para reduzir de fato os riscos, é preciso que haja um *feedback* constante dos clientes do projeto ou especialistas no domínio. Para tanto, é preciso que essas pessoas tenham em mãos uma aplicação executável. A melhor maneira de se reduzir os riscos é desenvolver o produto (ou linhas de produtos) em iterações curtas, ao fim de cada qual deve ser entregue um conjunto de funcionalidades potencialmente entregável.

A idéia é que o cliente receba esse conjunto de funcionalidades prontas, isto é, implementadas, testadas, documentadas, etc. Isso permitirá que ele avalie as diversas características do produto, como usabilidade, desempenho e segurança, além das funcionalidades, propriamente ditas. Como as iterações são curtas[2], o cliente tem a possibilidade de detectar rapidamente inconsistências, defeitos, problemas de interface, falta de adequação do produto ao perfil do usuário final, entre outros problemas que podem surgir durante o desenvolvimento de um produto.

O desenvolvimento iterativo foi proposto originalmente com foco em sistemas únicos. A cada iteração do desenvolvimento, faz-se um incremento ao produto. No caso de linhas de produtos, o desenvolvimento pode ser dividido em duas partes: a engenharia de domínio e a engenharia de aplicações, de sorte que esta última é realizada sobre os artefatos produzidos pela engenharia de domínio, levando necessariamente a uma serialização entre tais atividades. Isso implica que, sob

[2] A literatura sobre métodos ágeis sugere de 1 a 4 semanas.

um determinado ponto de vista, o engenheiro de aplicações é o cliente da linha de produtos. Mas o desenvolvimento iterativo proposto neste trabalho visa ao usuário final e não ao engenheiro de aplicações. Por essa razão, as histórias são escritas do ponto de vista do usuário e priorizadas por valor de negócios. Neste sentido, o desenvolvimento do produto final é tratado como um único processo, abrangente e coeso, mesmo que compreenda duas atividades distintas.

Respostas rápidas permitem, igualmente, resoluções rápidas dos problemas. Decorre que, na pior das hipóteses, se o que foi produzido em uma dada iteração não satisfizer em absoluto o que o cliente queria (ou necessitava), somente o trabalho daquela iteração será perdido. Ainda assim, nem tudo terá sido desperdiçado. A própria falha em entregar valor de negócios ao cliente é uma oportunidade de aprendizado para as iterações seguintes. Ao contrário, quando se decide entregar todo o produto ao final do projeto, corre-se um risco muito grande de que o produto como um todo não atenda às necessidades do cliente. Nesse momento também não há tempo para melhorar, já que o projeto terá sido dado por "concluído".

No entanto, em um desenvolvimento iterativo, o produto não precisa ser colocado em produção ao fim de cada iteração. Determinados conjuntos de funcionalidades somente fazem sentido ao usuário final quando entregues como um pacote. Por exemplo, não há razão para entregar um editor de texto que permita abrir e editar arquivos, mas não permita salvá-los. Por outro lado, do ponto de vista de desenvolvimento, essas funcionalidades são relativamente independentes e podem ser implementadas e testadas separadamente, podendo ser feitas, inclusive, em iterações diferentes. Esse fenômeno é especialmente comum no início dos projetos.

À medida que novas funcionalidades vão sendo acrescidas ao produto e entregues para avaliação do cliente, tal produto vai gradualmente amadurecendo até que chegue a um ponto em que possa ser disponibilizado aos usuários finais. Quando um produto atinge esse estágio, diz-se que ele está pronto para lançamento (*release*). Vários lançamentos podem ser feitos em um projeto, cada um englobando o resultado de uma ou mais iterações de desenvolvimento.

Na implementação do sistema BET feita neste estudo de caso, foram necessárias 4 iterações até que a linha de produtos estivesse pronta para lançamento. Ao fim de cada uma dessas iterações, contudo, a linha contava com um conjunto novo de funcionalidades, podendo ser considerada potencialmente entregável.

O desenvolvimento iterativo empregado no sistema BET está fortemente ligado ao *backlog* do produto (apêndice A). Já foi explicado aqui que a ordem das histórias em um *backlog* é de fundamental importância. Elas são ordenadas pelo valor de negócio que proporcionam. Como o desenvolvimento é iterativo, é preciso escolher quais histórias desenvolver a cada iteração. Se as histórias forem escolhidas na ordem em que estão dispostas no *backlog*, garante-se que as funcionalidades que agregam mais valor serão implementadas primeiro.

Uma história de usuário é considerada atômica; ou ela é inteiramente entregue ou não é entregue em absoluto (mesmo que já tenha sido desenvolvida parcialmente). O escopo de cada história, porém, é negociável e depende de um acordo entre desenvolvedores e clientes. No caso de linhas

de produtos, essa questão do escopo tem um peso ainda maior, motivado pela existência de variabilidades. Assim, uma história pode ter partes que variam de produto para produto, mas neste caso todas essas partes devem ser desenvolvidas e entregues na mesma iteração.

Caso se julgue que uma determinada história esteja com o escopo muito amplo ou que a dificuldade para se estimar a complexidade dessa história (e conseqüentemente o esforço de desenvolvimento) seja muito grande, essa história pode ser dividida em duas ou mais histórias. Estando separadas, essas histórias podem ser implementadas na mesma iteração ou em iterações diferentes, dependendo da priorização feita. O mesmo vale em relação às funcionalidades do núcleo. Nada impede que uma história contenha, por exemplo, uma variabilidade e uma funcionalidade do núcleo, desde que haja um acoplamento natural entre essas funcionalidades que justifique a inclusão de ambas na mesma história. Neste caso, a regra de entregar a história completa ao fim da iteração vale igualmente.

Apesar de não haver uma regra rígida sobre como elaborar e dividir histórias, é bastante razoável adotar a heurística de "quebrar" as histórias sempre que possível, reduzindo seu escopo sem que elas deixem de proporcionar algum benefício ao usuário final[3]. Quanto menor for o escopo de uma história, mais precisas serão as estimativas complexidade dessa história. Alguns métodos ágeis, como Scrum e XP, propõem o uso de uma escala não linear de números inteiros para essas estimativas. A seqüência de Fibonacci e as progressões geométricas são as escalas mais adotadas, por terem em comum a característica de que a distância entre os pontos aumenta à medida que se avança na seqüência.

A Figura 3.4 ilustra esse conceito. Nela está representada uma escala de Fibonacci, variando de 2 a 21. Há duas histórias, h_1 e h_2, cujas complexidades foram estimadas em 3 e 13 pontos, respectivamente. A história h_1 é relativamente simples, considerando que foi classificada no segundo marco da escala. Assim, caso sejam incluídas algumas validações a mais ou alguma regra de negócio seja criada, será fácil mensurar (ainda que subjetivamente) essa variação, fazendo com que a estimativa de complexidade passe para 8 pontos, configurando, portanto, uma variação de 3 pontos.

A história h_2, por outro lado, é relativamente complexa, dentro dessa escala de valores. Neste caso, pequenas alterações no escopo da história causam pouco impacto na estimativa de complexidade, já que a própria dificuldade prevista torna a estimativa mais incerta. Conseqüentemente, não faria sentido uma variação de 3 pontos. Por isso, o próximo ponto da escala é 21, para dar conta de uma grande mudança no escopo da história.

A heurística da máxima divisão possível tem um outro efeito: o aumento na flexibilidade para priorização. Duas funcionalidades relativamente independentes, se incluídas na mesma história, forçarão os desenvolvedores a entregá-las de uma só vez, segundo o que foi comentado acima. Ocorre que essas histórias podem ter relevâncias diferentes para o usuário final, o que implicaria

[3]Uma história puramente técnica, como "trocar o SGBD relacional para que seja possível utilizar um dialeto específico da linguagem SQL", por exemplo, não agrega nenhum valor observável pelo usuário final.

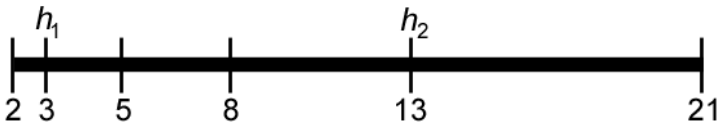


Figura 3.4: Escala de Fibonacci para estimativa de complexidade das histórias.

posições bem diferentes no *backlog*. Ao desassociá-las, torna-se possível entregar cada funcionalidade em uma iteração diferente, deixando livre uma parte do tempo dos desenvolvedores, que poderia ser usada na implementação de uma história mais relevante.

3.6.2 Scrum

O Scrum, proposto inicialmente por Schwaber (1995) é um método ágil de desenvolvimento de software baseado nas idéias de Takeuchi e Nonaka (1986) para desenvolvimento de novos produtos, em que se propõe que um time multifuncional desenvolva o produto em todas as suas fases. Além disso, deve haver uma grande sobreposição entre essas fases, de modo a aumentar a rapidez e flexibilidade no desenvolvimento. Essa abordagem holística mostrou-se bastante eficaz para empresas como Fuji-Xerox, Honda e Canon.

No contexto de desenvolvimento de software, o método Scrum prescreve um processo em que o produto é desenvolvido em iterações curtas, denominadas *sprints*. Um pré-requisito para que um *sprint* se inicie é que haja um *backlog* do produto, isto é, uma lista de funcionalidades desejadas e na ordem em que se espera que sejam implementadas. As funcionalidades são resumidas na forma de histórias de usuário, que são sentenças curtas para lembrar aos desenvolvedores o que precisa ser feito. No planejamento do *sprint*, os desenvolvedores escolhem um subconjunto dessas funcionalidades e produzem um *backlog* do *sprint*, que contém as tarefas detalhadas de implementação, testes, arquitetura, etc.

Durante o *sprint* são feitas reuniões diárias de 15 minutos, para que se faça uma troca de informações sobre o estado do projeto. Terminado o *sprint*, os desenvolvedores entregam um incremento ao produto. Para que o trabalho possa ser considerado "feito" (*done*), é preciso que ele atenda a uma série de critérios. As funcionalidades devem ter sido implementadas, testadas, refatoradas, além dos critérios específicos que a empresa ou o cliente porventura exijam. Atendendo aos critérios de acabamento, as funcionalidades são acrescidas ao produto, tornando-o potencialmente entregável. Uma visão geral do processo de desenvolvimento com Scrum é mostrada na Figura 3.5

Em Scrum, há apenas três papéis: os membros do time, o *Product Owner* e o *Scrum Master*. Considerando que o time é multifuncional, toda a responsabilidade pelo desenvolvimento do produto está a cargo de seus membros, que se comprometem a entregar o produto ao fim do *sprint*. A tarefa de gerenciamento do projeto, inclusive, é feita pelos próprios membros do time, que devem se auto-organizar.

O *Product Owner* (PO) é o representante do cliente (quando não for o próprio cliente, o que é ainda mais desejável). Suas funções incluem priorizar o *backlog* do produto, de modo a maximizar

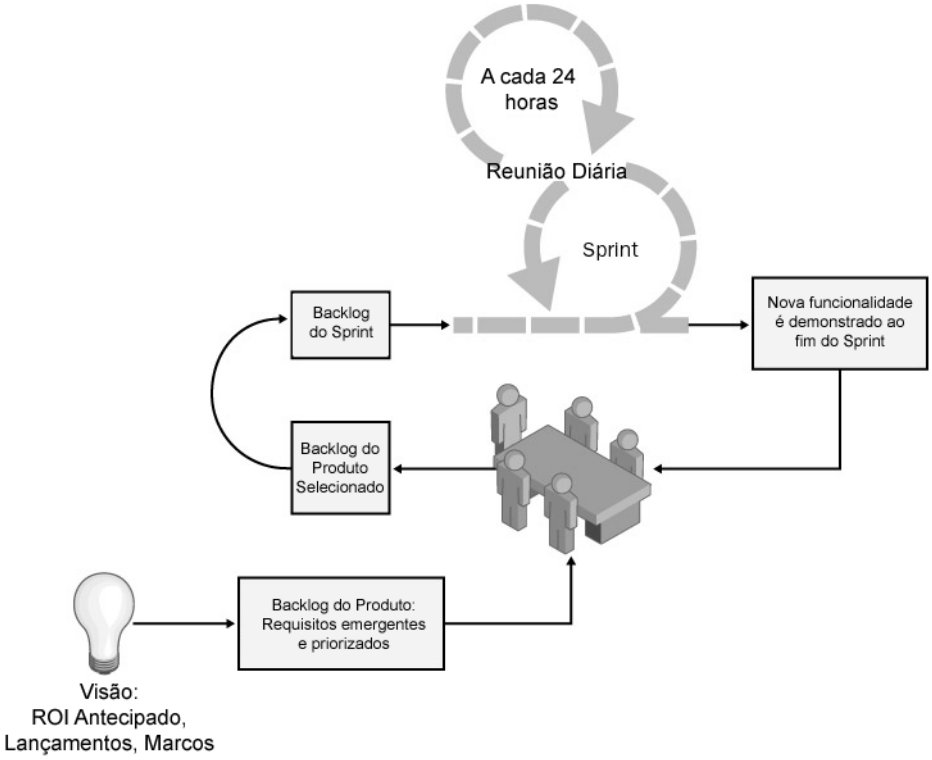


Figura 3.5: Visão geral do Scrum (Schwaber, 2004).

o ROI (*return on investiment*), acompanhar de perto o desenvolvimento do produto, responder às dúvidas do time e dar o aceite ou não para as histórias. É importante que o PO tenha autoridade e liberdade para determinar os rumos do projeto, compreendendo mudanças de escopo e priorização.

O *Scrum Master* é responsável por remover os impedimentos do time, marcar e controlar o tempo das reuniões[4], além de promover um bom ambiente de trabalho para o time. Seu papel é reconhecido como uma liderança servil (Stone *et al.*, 2004), já que em Scrum não existe a figura de um gerente de projetos que comanda e controla as atividades de profissionais hierarquicamente inferiores.

3.6.3 Programação Extrema

Ao contrário do Scrum, cujo foco principal são as práticas de gerenciamento de projeto, a Programação Extrema (Beck, 2000) — XP, na sigla em inglês — direciona a atenção dos desenvolvedores para as práticas de engenharia e para os modos de interação entre as pessoas. Os alicerces da Programação Extrema são conjuntos de valores, princípios e práticas de desenvolvimento de software, que representam uma mudança de paradigma em relação aos processos tradicionais. XP pretende ser tanto um método de desenvolvimento quanto um guia de hábitos sociais para os desenvolvedores.

[4]Em Scrum, todas as reuniões têm tempo fixo para começar e terminar.

Há cinco valores fundamentais em XP, que equilibram e apóiam uns aos outros. O primeiro deles é o da comunicação. Para que o desenvolvimento seja produtivo, é preciso que todos os problemas e dificuldades sejam comunicados honestamente ao resto da equipe, tão logo quanto possível. Do mesmo modo, o compartilhamento de soluções e idéias deve ser feito com freqüência. A comunicação face-a-face, particularmente, costuma produzir melhores resultados do que a comunicação feita por outros meios, como especificações textuais, por exemplo.

Relacionado à comunicação, há um outro valor: o da resposta (*feedback*), que sugere que deve-se estar atento às mudanças e responder a elas o mais cedo possível. Os métodos tradicionais sugerem que, por meio da execução de determinadas atividades, pode-se alcançar a solução "correta" para os problemas de projeto desde o início. Infelizmente, isso nem sempre é possível, porque problemas novos podem admitir soluções variadas, dentre as quais pode ser difícil determinar qual é a melhor. Além disso, as necessidades mudam constantemente e uma solução encontrada no início de um projeto pode não ser a mais adequada algum tempo depois. Por essas razões, a capacidade de resposta é tão importante.

Outro valor é a simplicidade. Um dos lemas de XP é "faça a coisa mais simples que funcione". Praticantes de XP valorizam, por exemplo, projetos simples, com classes pequenas, nomes intuitivos, que revelem a intenção do programador. Além disso, evitam associações desnecessárias entre classes e procuram reduzir dependências externas às aplicações.

Dentre os vários fatores comportamentais que podem causar problemas em uma equipe de desenvolvimento estão o medo e a confiança excessiva. O primeiro caracteriza-se pela fuga dos problemas críticos e o segundo por uma irresponsabilidade nas ações, o que pode ter conseqüências desastrosas para o time. O meio-termo entre essas duas tendências é a coragem, pregada como outro dos cinco valores, e que se manifesta de várias formas: coragem para falar a verdade, paciência diante da percepção de um problema cujos detalhes não são completamente conhecidos ou a ação enérgica diante de problemas evidentes.

Por fim, é preciso que haja respeito mútuo entre os membros do time, para que um projeto seja bem-sucedido, assim como é preciso que exista o respeito das pessoas pelo trabalho que estão realizando.

Esse conjunto de valores não teria utilidade a menos que pudesse ser aplicado a circunstâncias particulares. Por isso, há também em XP um conjunto de práticas de engenharia, baseadas nesses valores. Estas são divididas em dois grupos: as práticas primárias e as práticas corolárias. As práticas primárias são descritas como aquelas que devem ser implantadas inicialmente em um projeto XP. Entre elas, encontram-se:

Programação em pares: Modo de organização do trabalho em que dois desenvolvedores trabalham juntos no mesmo computador sobre o mesmo código ao mesmo tempo. Enquanto um dos programadores concentra-se nos detalhes, o outro raciocina em mais alto-nível, tentando prever o impacto de cada alteração no projeto como um todo. Com esta prática, o nível de comunicação e a capacidade de resposta do time aumentam consideravelmente.

Integração contínua: Quando o número de desenvolvedores trabalhando em paralelo na mesma aplicação é grande, a chance de problemas na integração entre o que foi produzido por cada um também tende a ser grande. Para evitar esses problemas, é aconselhável que seja feita uma integração do sistema a cada duas horas no máximo. Para que essa prática seja eficaz, é preciso que o sistema esteja bem coberto por testes automatizados. Assim, a cada integração, os testes são executados e qualquer falha pode ser comunicada aos desenvolvedores por email, por exemplo. Quanto mais cedo as falhas forem corrigidas, melhor.

Testes *a priori*: A prática de escrever testes automatizados antes do próprio código de produção ajuda os desenvolvedores a manter o foco nas funcionalidades requisitadas. Como os testes são escritos antes, o código de produção é que se adapta aos testes e não o contrário. O resultado é um código naturalmente testável, o que equivale a baixo acoplamento entre as classes e alta coesão interna. A confiança na qualidade do código também é maior quando os testes são feitos antes. A cada novo teste que é executado com sucesso, tem-se um bom indicativo de que houve um avanço no projeto, embora pequeno. Se esses pequenos avanços são feitos com bastante freqüência, o desenvolvimento segue a um ritmo sustentável.

Histórias: Assim como o Scrum, a Programação Extrema sugere o uso de histórias de usuário no lugar de longos documentos de requisitos ou casos de uso.

Projeto incremental: Tendo em vista que uma das sugestões é que se adote a solução mais simples possível, é preciso que o time esteja preparado para mudanças constantes. Decisões de projeto devem evoluir junto com as mudanças de escopo e de entendimento do problema. Se a mudança é constante, o projeto deve estar preparado para sofrê-las com o menor custo possível, o que as práticas anteriores ajudam a promover.

Uma vez adotadas as práticas primárias com sucesso, o time pode dar um passo à frente e implantar as práticas corolárias, consideradas mais avançadas e que requerem maior experiência dos desenvolvedores. Tais práticas incluem o envolvimento real do cliente (preferencialmente com o cliente fazendo parte do time), implantação incremental e diária, continuidade do time ao longo do tempo, redução no tamanho dos times, análise de causa-raiz dos problemas, código compartilhado (de modo que todos os membros do time trabalhem em todos os trechos de código), base de código única, escopo negociado e pagamento por uso.

3.7 Considerações finais

Domain-Driven Design é uma filosofia de projeto que incorpora a idéia de desenvolvimento iterativo e evolução incremental de um modelo de domínio, expresso diretamente no código. Paralelo ao modelo, deve existir uma linguagem ubíqua, isto é, uma linguagem que permeie todas as discussões sobre o domínio. De modo análogo, essa linguagem evolui de modo incremental ao longo do projeto.

A partir de princípios gerais, é proposto um conjunto de padrões básicos de desenvolvimento, que funcionam como "blocos de construção" para o modelo de domínio. Além desses padrões, DDD sugere diversas outras técnicas para desenvolvimento, algumas delas relacionadas a preservar a integridade do modelo face a outras aplicações e contextos. No entanto essas técnicas fogem ao escopo deste trabalho, que é analisar o desenvolvimento de linhas de produtos e os ganhos em reusabilidade que se podem obter pelo emprego dos princípios de DDD.

O conjunto de diretrizes proposto por DDD está intimamente relacionado aos valores propostos pelos métodos ágeis. Dentre outros pontos, podem-se observar as seguintes convergências entre ambas as propostas: comunicação freqüente entre desenvolvedores e clientes, valorização de código funcionando acima de outros tipos de artefatos, busca por soluções simples e desenvolvimento iterativo.

Esse conjunto de valores é exatamente o oposto daquilo que os métodos tradicionais de desenvolvimento sugerem. Por conta disto e por serem razoavelmente recentes — o manifesto ágil, que representa a formalização dessas idéias, foi publicado em 2001 — ainda causam muita polêmica e ceticismo. Apesar disso, alguns estudos sugerem que são superiores aos processos em cascata, em vários aspectos, como produtividade, qualidade do produto final, atendimento do escopo e cumprimento de prazos.

A revisão apresentada neste capítulo traz uma idéia geral de DDD e métodos ágeis, duas propostas que se complementam. O interesse deste trabalho é estudar como essas duas propostas podem ser aplicadas no contexto de linhas de produtos, que é o tema do próximo capítulo.

CAPÍTULO

4

Estudo de Caso e Diretrizes de Desenvolvimento

4.1 Considerações iniciais

Com o objetivo de embasar a discussão em problemas práticos, foi feito um estudo de caso para este trabalho. Uma linha de produtos de software foi implementada com base nos princípios de DDD e seu desenvolvimento foi feito de modo ágil. Este estudo de caso ajudou a levantar indícios de que essa combinação entre métodos ágeis e projeto guiado pelo domínio pode levar a resultados melhores, comparados aos métodos tradicionais de desenvolvimento de linhas de produtos.

A linha de produtos de software desenvolvida como estudo de caso neste trabalho é um protótipo conhecido como sistema BET (Bilhetes Eletrônicos de Transporte). Seu propósito é o gerenciamento de sistemas de transporte urbano. Considerando que as regras para transporte urbano variam de cidade para cidade, essas diferenças foram modeladas como variabilidades da linha, enquanto as regras e conceitos comuns foram desenvolvidos como parte do núcleo.

Na seção 4.2 são apresentados os detalhes sobre o domínio de transporte urbano e as regras de negócio relevantes para este estudo de caso, além de uma visão geral da linha de produtos desenvolvida. Em seguida, as decisões de projeto e implementação que foram tomadas neste projeto são discutidas na seção 4.3. Os detalhes de cada iteração do desenvolvimento da linha de produtos BET são expostos na seção 4.4. A engenharia de aplicações é discutida em duas partes; a seção 4.5 trata das decisões de projeto para engenharia de aplicações e a seção 4.6 aborda os resultados do uso da ferramenta Captor. Na seção 4.7, é feita uma comparação com uma linha de produtos para

o mesmo domínio, desenvolvida segundo as propostas tradicionais. Por fim, um conjunto de diretrizes para o desenvolvimento de linhas de produtos com base em métodos ágeis e *Domain-Driven Design* é sugerido na seção 4.8.

4.2 Sistema BET

No domínio de transportes urbanos, muitas regras de negócio variam de cidade para cidade, mesmo havendo um conjunto básico de regras e conceitos que permeiam todas elas. Para este trabalho, foram examinados os sistemas de transporte de três cidades: Campo Grande (MS), Fortaleza (CE) e São Carlos (SP). Em conjunto, os três sistemas compõem uma amostra significativa das variabilidades que se podem encontrar nesse domínio (Donegan, 2008).

Um sistema de transportes urbanos visa a prover serviços de locomoção, utilizando algum tipo de veículo como meio de transporte. Os veículos mais comumente utilizados são ônibus, micro-ônibus, metrô e trens metropolitanos. Os veículos trafegam em linhas determinadas, com horário de partida e chegada pré-agendados. Tanto o ponto de partida quanto o de chegada são terminais (podendo, inclusive, ser o mesmo para partida e chegada).

As linhas, seus itinerários e horários são definidos pelo administrador do sistema, de modo a atender de maneira eficiente as demandas dos passageiros. Os administradores também são responsáveis pelo cadastramento e atualização dos dados referentes aos terminais e à frota de veículos mantidos pela empresa.

Os passageiros podem efetuar o pagamento da viagem dentro do veículo ou ao entrar no terminal. Neste último caso, os terminais devem ser fechados e somente permitir a entrada de passageiros mediante pagamento da passagem. No caso básico, uma passagem dá direito a somente uma viagem. Variações dessa regra incluem:

Integração por tempo: os passageiros têm direito a utilizar o sistema de transporte quantas vezes desejarem pagando apenas uma passagem. Essa passagem é válida durante um determinado período de tempo (definido pela empresa viária) após o pagamento da passagem.

Integração por número de viagens: os passageiros têm direito a um determinado número de viagens (também definido pela empresa viária) por passagem. O tempo decorrido entre uma viagem e outra não é considerado.

Um híbrido das duas regras acima também é possível, ou seja, uma passagem é válida para um determinado número de viagens dentro de um determinado período de tempo, aquele que expirar primeiro.

Para que a empresa viária possa aplicar políticas diferentes para tipos diferentes de passageiros, existem as chamadas categorias de passageiros. Cada passageiro pode pertencer a uma ou mais categorias. Um caso particular desse tipo de política é a concessão de descontos. Por exemplo,

alguns municípios possuem leis que determinam que pessoas com mais de 60 anos paguem somente meia-passagem. Nesse caso, os passageiros com mais de 60 anos seriam incluídos em uma categoria chamada, digamos, "Idoso", que teria direito a 50% de desconto.

Por razões financeiras e de comodidade dos passageiros, as passagens são pré-pagas. Em um sistema de passagens pré-pagas, os usuários carregam seus cartões fazendo um pagamento. Esse pagamento dá direito a um determinado número de passagens, que podem ser usadas posteriormente. Para que isso seja possível, cada passageiro deve possuir um cartão para cada uma das categorias às quais pertence. Os cartões possuem um saldo, do qual se desconta o preço da passagem a cada viagem (ou integração entre viagens, caso haja) que o passageiro faça.

O valor da tarifa pode ser reajustado a qualquer momento pela empresa de viação. Por motivos de auditoria e possíveis análises financeiras que a empresa precise fazer, os valores antigos das tarifas permanecem armazenados. Isso torna possível rastrear qual era o valor vigente da tarifa em um dado momento do tempo.

O reajuste da tarifa não implica alterações no número de passagens que um cartão já possui. Por exemplo, considere-se um cartão com 10 passagens. Se um passageiro inserir R$ 50,00 em crédito naquele cartão quando o valor de tarifa for R$ 2,00, o número de passagens nesse cartão será acrescido de 25 passagens, passando para 35 o saldo total de passagens. Se o valor da tarifa for reajustado para R$ 2,50, o número de passagens desse cartão continuará sendo 35 e essas passagens poderão ser consumidas normalmente. Mas se o passageiro fizer novamente uma carga de R$ 50,00, o número de passagens, desta vez, será acrescido de 20. Em resumo, os créditos são feitos por valor financeiro e convertidos em passagens. Os débitos, isto é, o consumo durante o uso, são feitos por passagem, independentemente de quanto elas custaram ao passageiro.

Ainda em relação ao gerenciamento financeiro, a empresa precisa que os dados de receita, originados das passagens compradas pelos usuários do sistema, sejam armazenados. Esses dados são processados pelo departamento financeiro. Vale lembrar que esse processamento dos dados financeiros está fora do escopo do domínio deste estudo de caso. Os dados são apenas gerados e seu tratamento seria assunto para uma outra discussão, referente a um outro domínio, cujo foco seriam operações contábeis e financeiras.

Os passageiros podem recarregar seus cartões tão freqüentemente quanto quiserem, segundo a regra básica. Uma variação dessa regra estabelece um limite máximo de passagens que os passageiros de uma categoria podem comprar por mês. Isso é interessante do ponto de vista da empresa viária, uma vez que se não houvesse limite, as passagens com desconto compradas em grande quantidade em um curto período de tempo poderiam representar um risco financeiro para a companhia.

É de interesse dos passageiros consultar um extrato de uso de seus cartões. Por isso, a empresa viária deve oferecer a seus usuários, algum relatório de viagens realizadas, no qual deve constar data e horário de cada viagem, preço pago, linha utilizada (e integrações, quando for o caso).

A regra básica diz que cada passageiro pode pertencer a várias categorias. Conseqüentemente, os passageiros podem possuir um número qualquer de cartões em qualquer combinação. Como não há nenhuma restrição nessa regra, um passageiro pode, inclusive, possuir mais de um cartão para a mesma categoria. Há algumas variações para essa regra. Alguns exemplos:

Categorias não duplicadas: cada passageiro tem direito a, no máximo, um cartão para cada categoria.

Limite de cartões: cada passageiro tem direito a um número máximo de cartões definidos pela empresa viária.

Combinações permitidas: há combinações permitidas e proibidas para cartões. Os conjuntos de cartões de todos os passageiros devem respeitar as combinações permitidas.

As empresas podem adotar qualquer uma dessas regras de cartões, podendo inclusive, combinar duas ou mais regras, de acordo com as necessidades específicas de cada uma.

O oferecimento de vale-transporte como benefício a seus funcionários é uma prática comum entre as empresas. Por isso, muitas companhias firmam convênios com as empresas viárias. De um lado, a companhia ganha em facilidade e geralmente recebe descontos. Do outro, a empresa viária conquista um cliente fiel e assíduo.

Assim, todo mês essas companhias — denominadas usuários corporativos — fazem uma recarga em lote de créditos para todos os seus funcionários. Para que o processo seja simples, independente da quantidade de funcionários que irão receber o benefício, as empresas viárias criam uma categoria especial de cartões, destinada a vale-transporte. Sempre que um usuário corporativo conveniado deseja fazer uma recarga em lote, ele determina o valor a ser creditado e a empresa viária credita esse valor para todos os funcionários, de uma só vez.

Muitas outras regras e conceitos existem neste domínio. Entretanto, uma discussão limitada ao que foi apresentado nesta seção é o suficiente para analisar vários problemas e soluções relacionados a reúso e a aplicação de princípios de DDD e métodos ágeis. Vale lembrar que nem todas as regras citadas aqui foram implementadas. A própria decisão do que deveria ser implementado e em que ordem também seguiu princípios ágeis.

4.3 Decisões de projeto e implementação

O *backlog* apresentado no apêndice A contém 14 histórias de usuário, ordenadas segundo a prioridade de negócios. Por restrições de tempo, foram implementadas apenas as 7 primeiras histórias, ao longo de 4 iterações. As principais decisões de projeto e implementação para cada uma dessas histórias e a evolução da linha de produtos são discutidas a seguir.

4.3.1 Visão geral da aplicação

Para atender às necessidades listadas no *backlog*, o sistema BET foi dividido em dois subsistemas: um servidor, que centraliza o processamento das regras de negócio; e diversos clientes, que são instalados nos validadores, tanto dos terminais, quanto dos ônibus. Os validadores enviam os dados *on-line* para o servidor sempre que um cartão é passado. Os dados incluem um identificador do cartão e um identificador do validador. Ao receber as informações do validador, o servidor as processa e determina se o passageiro está autorizado a fazer a viagem ou não.

Os validadores estão associados a dispositivos de entrada e saída que são a interface com o usuário[1]. A entrada de dados é feita por dois dispositivos: um leitor de cartão e a catraca. O leitor decodifica os dados que estão armazenados no cartão e os transforma em um código numérico que é enviado ao validador. A função da catraca, como dispositivo de entrada, consiste apenas em enviar um sinal para o validador a fim de notificá-lo de que ela foi girada pelo passageiro.

A saída de dados também é feita por dois dispositivos: um visor e a catraca, novamente. O visor exibe informações sobre o estado da catraca (liberada ou travada), o código do cartão, seu saldo e quaisquer informações extras que se precise configurar. A catraca, na condição de dispositivo de saída, recebe um sinal do validador, que é interpretado como um comando para travar ou liberar a passagem.

O subsistema de validadores, do ponto de vista do usuário, é uma interface para um uso específico: a autorização de viagem dentro de um ônibus ou terminal. Além desta interface de usuário, há também uma interface *web*, para gerenciamento e consultas no sistema. Esta última inclui gerenciamento de cartões, passageiros, usuários corporativos, categorias, tarifas de transporte e viagens realizadas pelos passageiros. Todas essas funcionalidades são analisadas em detalhes mais adiante.

4.3.2 Tecnologia utilizada

A escolha da tecnologia a ser utilizada levou em conta o fato de que o sistema BET é essencialmente uma aplicação *web*. Mesmo a comunicação entre validadores e servidor pode ser feita via HTTP. Conseqüentemente, a decisão tecnológica mais importante diz respeito ao *framework* a ser empregado. Diversos arcabouços para desenvolvimento de aplicações *web* têm se destacado ultimamente, entre eles o Struts 2 (Brown *et al.*, 2007), JBoss Seam (Allen, 2008), Spring Web MVC (Walls e Breidenbach, 2007), Rails (Thomas e Hansson, 2006), Django (Holovaty e Kaplan-Moss, 2007) e Grails (Rocher, 2006). Destes, foi escolhido o Grails, pelas seguintes razões:

1. O *framework* é construído sobre diversos outros *frameworks* estáveis e populares, como Spring e Hibernate, entre outros.

[1]Neste caso, há dois usuários: o cobrador e o passageiro

2. A linguagem de programação usada no Grails é o Groovy (Koenig, 2007), uma linguagem de tipagem dinâmica, que se integra perfeitamente com a tecnologia Java, que também é estável, popular e bastante eficiente.

3. Adesão à idéia de "convenção mais que configuração": ao contrário dos *frameworks* tradicionais, a necessidade de configuração em Grails é bem menor. Grande parte das decisões tomadas pelo *framework* são baseadas em convenções de nomenclatura, o que ajuda a manter o foco mais no domínio que na infra-estrutura.

Conforme discutido na seção 3.2.3, o isolamento do domínio é uma das idéias centrais em DDD. Esse isolamento é mais bem alcançado quando se emprega uma arquitetura em camadas. O sistema foi dividido nas quatro camadas já citadas: apresentação (ou de interface), aplicação, domínio e infra-estrutura.

O mapeamento dessas camadas para o arcabouço tecnológico adotado é relativamente simples. No Grails, uma das convenções é a estrutura de diretórios usada para abrigar os diferentes tipos de arquivos, que é mostrada na Figura 4.1.

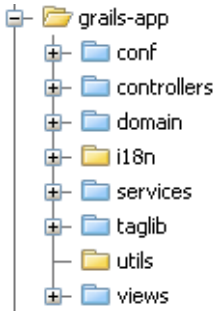


Figura 4.1: Estrutura de diretórios do Grails.

O principal diretório na estrutura do Grails é o `grails-app`. Dentro dele, os diretórios `controllers` e `views` são usados para armazenar, respectivamente, os controladores e as visões. O Grails é baseado no padrão MVC (*Model-View-Controller*), descrito inicialmente por Reenskaug (1979). O propósito desse padrão é promover uma separação entre elementos de interface, denominados visões (*views*) e modelos de informação subjacentes a esses elementos de interface, conhecidos no padrão como modelos (*models*). Para coordenar esses dois conceitos, existem os controladores (*controllers*).

Os controladores recebem as requisições do usuário, provindas dos elementos da interface, como botões e caixas de texto. Com base nos dados e comandos enviados pelo usuário, o controlador toma alguma decisão, como redirecionar o fluxo para outro controlador, fornecer uma mensagem ou delegar um comando a algum serviço da camada de aplicação.

Controladores e visões são gerados automaticamente pelo Grails, a partir de uma classe de domínio. Cada controlador gerado possui oito ações (comandos básicos que o usuário pode invocar): `create`, `delete`, `update`, `save`, `show`, `edit`, `index` e `list`. Cada uma dessas ações é, na

verdade, um método da classe. No conjunto, essas ações criadas automaticamente consistem de uma implementação básica — mas completa — das operações básicas de manipulação de dados. Esse código gerado pode ser facilmente alterado para atender as necessidades específicas para as quais o controlador foi criado.

Os elementos de interface são montados pelas visões. Estas são gabaritos (*templates*), contendo a estrutura básica dos arquivos HTML a serem gerados e algumas variáveis, cujo valor é ajustado pelos controladores em tempo de requisição. Nas visões é desejável que haja o mínimo de lógica de programação possível. Ainda assim, algumas estruturas de controle são necessárias, como estruturas de repetição para geração de tabelas, por exemplo.

A camada de aplicação é composta por um conjunto de classes de serviço (ver seção 3.3.3), que recebem os comandos dos controladores e coordenam outros serviços e/ou objetos do domínio. Essas classes estão localizadas no diretório `services`. Para as classes de domínio, há também um diretório específico, que foi apropriadamente denominado `domain`.

Quanto à infra-estrutura, a convenção é usar um diretório fora do `grails-app`. No entanto, como o Grails encapsula boa parte da infra-estrutura, geralmente não há necessidade de se criar classes com esse propósito. Dentre os ítens de infra-estrutura mais complexos e mais utilizados em aplicações corporativas, encontra-se a persistência de dados. Conforme já citado, o Grails é construído sobre *frameworks* estáveis para a plataforma Java, em especial o Hibernate, cuja função é prover um serviço de mapeamento objeto-relacional.

O Grails vai um passo além em relação à persistência e encapsula os serviços do Hibernate sob uma interface simples, utilizando um mecanismo da linguagem Groovy conhecido como metaprogramação. Resumidamente, a metaprogramação é um protocolo para troca de mensagens em Groovy. Toda vez que um objeto A envia uma mensagem a um objeto B, este último procura um método que responda a essa mensagem (por convenção, um método com a mesma assinatura da mensagem). Caso não exista tal método, a chamada é delegada para a sua *metaclasse* correspondente.

Em Groovy, todas as classes criadas pelos programadores implementam automaticamente a interface `GroovyObject` e já estendem uma implementação padrão dessa interface (do mesmo modo como todas as classes Java estendem automaticamente a classe `Object`). Pelo contrato da linguagem, todas as implementações de `GroovyObject` devem possuir uma associação com a classe `MetaClass`. Assim, ao serem criados, todos os objetos já possuem uma referência a uma instância de `MetaClass`. O padrão da linguagem Groovy, atualmente, é de uma `MetaClass` por classe, mas também é possível definir uma `MetaClass` por objeto.

Usando o protocolo de chamadas da linguagem, qualquer objeto pode enviar uma mensagem `save()` para um objeto do domínio. Como os objetos do domínio não devem conhecer nada sobre persistência, não deve haver nenhum método implementado com essa assinatura. O que esse objeto faz, então, é delegar a chamada para o método `invokeMethod(Object o, String name, Object[] args)` da sua metaclasse com os parâmetros correspondentes. O resultado é

um desacoplamento total da tarefa de persistência de todas as outras tarefas do sistema. Em outras palavras, o desenvolvedor trabalha apenas com os objetos do domínio, enviando a eles comandos de persistência, ignorando todo o mecanismo interno envolvido no mapeamento objeto-relacional que torna essa persistência possível.

Além de tornar a estrutura do projeto mais clara e previsível para os desenvolvedores, a organização convencionada de diretórios no Grails serve a outro propósito: facilitar o trabalho de *injeção de dependência*. Dados dois objetos A e B, diz-se que B é uma dependência de A se algum dos atributos (também chamados de variáveis de instância) de A é uma referência a B. Se A depende de B, uma referência a B pode ser obtida por um dos seguintes modos:

Internamente: A pode instanciar o B (usando o comando `new` em Java, por exemplo) ou requisitar a outro objeto uma referência a B. Os objetos que provêm instâncias a outros objetos geralmente são conhecidos globalmente pelos objetos da aplicação e recebem o nome de localizadores de serviço (*service locators*).

Externamente: A referência a B é passada como argumento a algum método de A, que se encarrega de armazenar essa referência internamente. Em Java, esses métodos são chamados de *setters*. Alternativamente essa referência pode ser passada por reflexão, nas linguagens que oferecem essa possibilidade.

Observe-se que há uma grande diferença entre as duas opções acima: no primeiro caso, o próprio objeto A está no controle sobre a obtenção da referência a B. No segundo, A não tem nenhum controle sobre a obtenção dessa referência. Algum outro objeto cuida da instanciação do objeto (um processo arbitrariamente complexo) e simplesmente repassa a referência para A. Portanto, obter uma referência a um objeto externamente é um caso particular de "inversão de controle" (Johnson e Foote, 1988). Esse caso particular recebe o nome de injeção de dependência (Fowler, 2004).

A injeção de dependência possui uma grande vantagem em relação à obtenção direta de referências, porque permite uma melhor separação de interesses. Idealmente, um objeto que encapsule lógica de negócios não deve também lidar com a tarefa de instanciação de outros objetos. Analogamente, deve haver classes cujo foco é exclusivamente a instanciação de objetos. Em orientação a objetos, essa separação é conhecida como Princípio da Responsabilidade Única (Martin, 2003), segundo o qual deve haver uma correspondência biunívoca entre classes e responsabilidades em uma aplicação. A injeção de dependência, portanto, ajuda os desenvolvedores a definir melhor as responsabilidades das classes, o que torna o código mais gerenciável.

Partindo-se do pressuposto de que cada classe apenas declara quais são seus colaboradores, conclui-se que a tarefa de "montagem" da aplicação, isto é, a instanciação dos objetos e o estabelecimento de interligações entre eles fica a cargo de outro conjunto de objetos, que não precisam fazer parte da aplicação. Por isso, existem atualmente diversos *frameworks* para injeção de dependência, como o PicoContainer (PicoContainer, 2008), o Spring e Guice (Google, 2008). Esses

frameworks possuem características diferentes entre si. Porém, uma característica comum a todos eles é a possibilidade de configuração declarativa do grafo de dependências da aplicação, utilizando, para isso, alguma linguagem que o *framework* reconheça.

No caso do Grails, o *framework* de injeção de dependência usado, conforme comentado anteriormente, é o Spring. Assim como no caso do mapeamento objeto-relacional, o Grails não apenas utiliza a tecnologia subjacente, mas a estende, incluindo novas funcionalidades e facilidades. Por meio de algumas convenções, o desenvolvedor consegue comunicar ao *framework* como ele deve injetar as dependências nos objetos. Uma dessas convenções prescreve que se houver variáveis de instância cujo nome contenha o sufixo `Service` (`employeeService`, por exemplo) e se houver uma classe com o mesmo nome dentro do diretório `services`, uma instância dessa classe será atribuída (injetada) a essa variável, sem necessidade de qualquer tipo de configuração.

Um último comentário em relação à estrutura de diretórios do Grails: os diretórios `conf`, `i18n`, `taglib` e `utils`, que não foram citados, contêm apenas classes e arquivos auxiliares e, portanto, foge ao escopo deste trabalho discuti-los. Uma abordagem mais completa sobre esse tópico é feita por Rocher (2006).

4.4 Iterações do Sistema BET

Em DDD, a atividade de modelagem e a de implementação não são vistas como processos separados. Ao contrário, prega-se que a modelagem deve ser feita no código, ao mesmo tempo em que o código deve refletir o conhecimento sobre o domínio. Isto torna as duas atividades quase indiscerníveis.

Além disso, a prática de modelagem adotada parte do pressuposto de que não é vantajoso tentar prever possíveis cenários futuros no modelo enquanto eles não forem necessários. Por isso, a cada iteração, o modelo reflete apenas o conhecimento necessário para o desenvolvimento das funcionalidades previstas em suas respectivas histórias de usuário. Com base nesses dois princípios (modelagem direta no código e foco nas necessidades imediatas), é apresentada nesta seção a evolução do projeto, iteração por iteração.

As funcionalidades foram desenvolvidas na ordem em que estão listadas no *backlog*, apresentado no apêndice A. A criação das histórias e a conseqüente priorização do *backlog* foi feita com a ajuda de uma pessoa com bastante conhecimento sobre o domínio em questão. No início do projeto, havia uma idéia razoavelmente clara a respeito do que precisava ser feito; a linha de produtos a ser desenvolvida deveria conter um subconjunto das funcionalidades implementadas para o mesmo domínio, em outra linha de produtos, referida neste trabalho como implementação de referência[2]. Apesar desse conceito inicial, a elaboração e priorização das histórias foi sendo feita iterativamente, alguns dias antes do início de cada iteração.

[2]Os detalhes dessa implementação são apontados na seção 4.7

Durante a implementação do sistema BET utilizando como base os princípios de DDD e métodos ágeis, surgiram vários problemas práticos de projeto. Sempre que possível, as soluções encontradas para cada um desses problemas foram generalizadas com o objetivo de serem reutilizadas em projetos posteriores. Tais generalizações, codificadas na forma de diretrizes de desenvolvimento, constituem uma das principais contribuições deste trabalho. Assim, ao fim das discussões sobre as soluções de projeto, as diretrizes correspondentes são enunciadas.

4.4.1 Iteração 1

Conforme citado anteriormente, as funcionalidades são sumarizadas na forma de histórias de usuário. Além dessa forma de representação, são também utilizados neste projeto os modelos de *features*, nos quais são mostradas as funcionalidades previstas para o sistema, seus tipos (opcional ou alternativa, por exemplo) e os diversos relacionamentos entre essas funcionalidades (grupos mutuamente exclusivos, pré-requisitos, funcionalidades mutuamente inclusivas).

Porém, considerando que as funcionalidades são desenvolvidas iterativamente, é necessário que o modelo de *features* também o seja. Conseqüentemente, o modelo de *features* neste projeto cresce a cada iteração, ao contrário dos métodos tradicionais, especialmente o ESPLEP, em que se propõe a criação de um modelo completo logo no início do projeto.

Na primeira iteração, foram desenvolvidas as duas primeiras histórias do *backlog*, relativas a aquisição de cartões e regras de combinação de cartões. As histórias são enunciadas abaixo, juntamente com seus respectivos testes de aceitação:

Como passageiro, quero adquirir um cartão de uma determinada categoria, para armazenar meus créditos.

- Criar cartão sem saldo
- Criar cartão com um saldo inicial
- Tentar criar um cartão com saldo negativo (deve falhar)
- Tentar criar um cartão para um passageiro inexistente (deve falhar)

A empresa de transporte municipal quer restringir as combinações de cartões permitidas por passageiro, para evitar perdas financeiras

- Criar cartão associado a uma categoria permitida. A categoria deve ser escolhida a partir de uma lista apresentada pelo sistema.
- Criar um cartão de modo a tornar uma das categorias não mais disponível (isto é, inválida para aquele passageiro). Como resultado, aquela categoria não deve ser apresentada na próxima tentativa de aquisição de cartões para o mesmo passageiro.

- Tentar forçar a criação de um cartão associado a uma categoria não-permitida (forjando URLs, por exemplo). A tentativa deve falhar.
- Esgotar as possibilidades de escolha de categorias para um passageiro. O sistema deve exibir uma mensagem informando que não há mais categorias disponíveis.

A escrita de histórias de usuários e testes de aceitação para cada história tornou o gerenciamento dos requisitos bastante simples, mesmo levando em conta o fato de que em linhas de produtos, o relacionamento entre as funcionalidades pode ser complexo (vide Seção 2.2.1). Tal constatação motivou a proposta da Diretriz 1.

Diretriz 1 *Ao elaborar as histórias de usuário, escreva os testes de aceitação, de modo que eles sirvam de base para a execução de testes manuais e para a escrita de testes automáticos.*

Antes do início do desenvolvimento, propriamente dito, dessas histórias, foi elaborado um modelo de *features* contendo apenas as duas funcionalidades a serem implementadas na primeira iteração. O diagrama de *features* da primeira iteração é apresentado na Figura 4.2. Uma das funcionalidades, na verdade, corresponde a um grupo de funcionalidades alternativas. Trata-se da restrição de cartões, do qual foram implementadas as funcionalidades de categorias não duplicadas e limite de cartões.

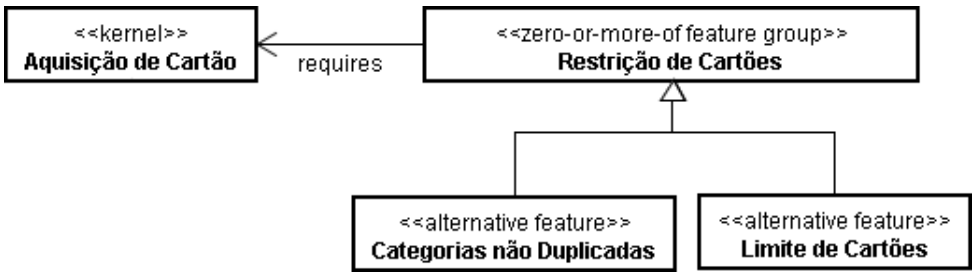


Figura 4.2: Diagrama de *features* correspondente às histórias da primeira iteração.

Relembrando as regras do domínio, cada passageiro pode ter vários cartões, mas cada cartão pertence a apenas uma categoria de passageiros. Há, portanto, três conceitos importantes no domínio: Passageiro, Cartão e Categoria. Como cada um deles possui identidade de continuidade temporal, todos são classificados como entidades, dentro da nomenclatura de DDD (ver seção 3.3.1). Esses conceitos foram mapeados diretamente para as classes `Passenger`, `Card` e `Category`, dentro do pacote `passenger`. Observe-se que alguns atributos, como `phoneNumber`, `address` e `cpf`, da classe `Passenger` pertencem aos tipos de dados `PhoneNumber`, `Address` e `CPF`, respectivamente. Esses tipos de dados são objetos de valor, no jargão DDD, já que representam apenas valores sem identidade.

A entidade `Passenger` e objetos de valor que compõem o seu conjunto de atributos formam um agregado de objetos. Deste modo, sempre que um `Passenger` (a raiz do agregado) for armazenado ou recuperado do mecanismo de persistência, todos os seus atributos também o serão.

Além disso, há invariantes semânticas que valem para todo o agregado. Por exemplo, ao criar um passageiro, é preciso que o CPF informado seja válido. Caso o código não passe na validação, que é feita pela própria classe `CPF`, o agregado, como um todo, não é criado.

Diretriz 2 *Dentre os conceitos do domínio envolvidos na história sendo implementada, identifique quais são entidades e quais são objetos de valor.*

Diretriz 3 *Identifique os conjuntos de entidades e objetos de valor que componham um conjunto coeso. Se houver uma entidade principal e invariantes semânticas que se apliquem ao conjunto como um todo, esse conjunto forma um agregado. As operações de persistência devem atuar sobre todo o agregado, respeitando essas invariantes.*

O diagrama da Figura 4.3 apresenta as classes desenvolvidas nesta iteração, bem como a divisão em pacotes e em camadas. Estas últimas estão diferenciadas por cores diferentes, explicadas na legenda.

Em um cenário típico de aquisição de cartão, o atendente da empresa viária fornece um identificador do passageiro (neste caso, o CPF). Considerando que o passageiro já está cadastrado no sistema, o próximo passo que o atendente deve efetuar é a escolha de uma categoria à qual esse cartão irá pertencer. Essa escolha é feita a partir de uma lista de categorias que o sistema sugere, dentre as categorias cadastradas.

Do ponto de vista da aplicação, o processo inicia-se quando o usuário requisita a página de aquisição de cartões. Essa página é gerada pelas visões da camada de apresentação, que não foram incluídas no diagrama por não serem relevantes a esta discussão. Uma vez carregada essa página, o usuário pode interagir com ela, preencher os dados necessários e submetê-los de volta ao servidor. A submissão desse formulário dispara uma chamada à ação `create`, da classe `CardController`.

O controlador apenas extrai as informações da requisição enviada pelo usuário e as repassa ao método `createCard`, da classe `CardAcquisitionService`, que é o serviço da camada de aplicação responsável pelas atividades relacionadas à aquisição de cartões. O método `createCard`, por sua vez, coordena os objetos da camada de domínio. De posse do identificador do passageiro, o serviço é capaz de recuperar (usando o mecanismo de persistência do Grails) o objeto da classe `Passenger` correspondente a esse passageiro. Uma operação análoga é feita para a categoria informada pelo usuário. O último passo, portanto, é criar um objeto da classe `Card`, associá-lo aos objetos `Passenger` e `Category` recuperados e persistir todo o grafo de objetos.

A funcionalidade de aquisição de cartões, por si só, não impõe qualquer restrição à escolha dos cartões. Contudo, essa liberdade nem sempre é desejável. Para algumas empresas viárias, a possibilidade de um passageiro adquirir mais de um cartão para uma mesma categoria, por exemplo, é uma situação problemática. Outras empresas precisam limitar o número de cartões que cada passageiro pode adquirir. Há outros casos, ainda, em que determinadas combinações de cartão são permitidas e outras, não. Por exemplo, pode-se estabelecer a regra de que um passageiro não pode ter um cartão da categoria “Idoso” e da categoria “Estudante” concomitantemente.

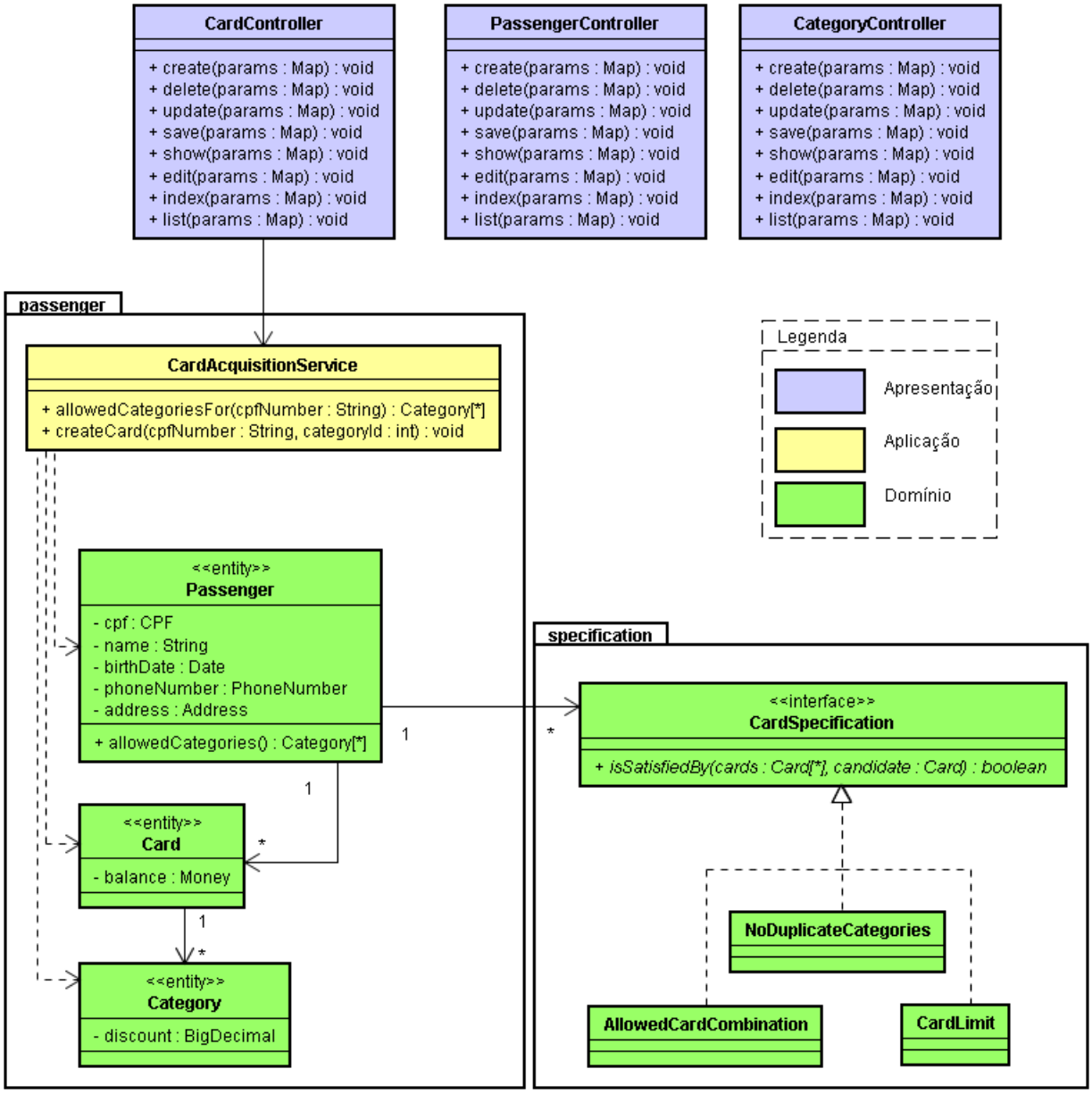


Figura 4.3: Diagrama de classes relacionado à aquisição de cartões.

Do ponto de vista de linhas de produtos, essas regras que restringem a escolha dos cartões são consideradas variabilidades alternativas não-mutuamente exclusivas. Qualquer combinação entre essas regras é possível, de modo que cada regra restringe um pouco mais o conjunto de cartões permitidos. O grupo de variabilidades de combinação de cartões, como um todo, é opcional: pode ou não haver restrições ao conjunto de cartões que o passageiro pode escolher.

O conjunto dessas regras e o relacionamento entre elas constituem uma parte importante do domínio. Além disso, um dos princípios de DDD é modelar o domínio no próprio código, de modo que ele reflita o mais próximo possível o conhecimento sobre esse domínio. As regras citadas, portanto, devem aparecer explicitamente no código, de alguma forma.

O padrão *Specification* (Evans e Fowler, 1997) é a solução ideal para esse problema de modelagem. Uma especificação é um predicado que determina se um objeto satisfaz ou não um certo critério; é uma afirmação em relação ao objeto e, como tal, pode ser verdadeira ou falsa. Seu uso é aconselhado nas situações em que as regras de negócio não se enquadram na responsabilidade de nenhuma entidade ou objeto de valor.

Cada regra de restrição de cartões pode ser compreendida como um predicado dos cartões, determinando se um cartão pode ou não ser adquirido, dado um conjunto de outros cartões. Deste modo, para cada regra foi criada uma classe diferente. Todas essas classes implementam a mesma interface, constituída de apenas um método: `boolean isSatisfiedBy(cards, candidate)`. A assinatura desse método possui dois parâmetros. O primeiro é uma lista de objetos da classe `Card` e o segundo é um objeto `Card`, candidato à especificação. O contrato desse método estabelece que suas implementações devem fornecer o valor *true* caso o cartão candidato satisfaça à especificação implementada (levando em conta a lista de cartões também passada como parâmetro) e *false*, caso contrário.

Diretriz 4 *Variabilidades alternativas expressas como predicados lógicos relativos a um determinado conceito podem ser modeladas como especificações.*

O código da implementação da regra de cartões não duplicados, por exemplo, é apresentada na Listagem 4.1. Da linha 3 à linha 6, todos os cartões da lista `cards` são analisados. Se algum desses cartões pertencer à mesma categoria do cartão candidato (linha 4), o método imediatamente fornece o valor `false`. Caso contrário, fornece `true`.

Observe-se que a iteração sobre os cartões é feita utilizando-se uma estrutura presente na linguagem Groovy, herdada das linguagens funcionais. Essas estruturas são chamadas de fechamentos (*closures*). Basicamente, um fechamento em uma linguagem de programação é um bloco de código reusável, que pode ser tratado como um valor literal, como um inteiro ou uma `String`, por exemplo. Isso implica que um fechamento pode ser atribuído a uma variável e passado como parâmetro para funções. Em linguagens que não contam com esse recurso, como Java, essa iteração seria feita usando um laço de repetição normal, iterando-se sobre a lista de cartões.

No caso da Listagem 4.1, um fechamento é passado como parâmetro para o método `each`, do objeto `cards`. Dado que esse método realiza uma iteração sobre os elementos da lista, cada elemento precisa ser referenciado por algum nome. O nome escolhido, neste caso, é `card` (linha 3). O sinal `->` é um separador entre os parâmetros do fechamento (antes do sinal) e o seu corpo (depois do sinal).

Listing 4.1: Implementação da regra de categorias não-duplicadas para cartões.

```
1 class NoDuplicateCategories {
2   boolean isSatisfiedBy(cards, candidate) {
3     cards.each { card ->
4       if (card.sameCategoryAs(candidate))
5         return false
6     }
```

```
7       return true
8     }
9   }
```

A implementação do limite de cartões é exposta na Listagem 4.2. A implementação dessa regra é ainda mais simples que a de cartões não duplicados. A especificação é baseada em um limite máximo de cartões definido igualmente para todos os passageiros. Conseqüentemente, dada uma lista de cartões e um cartão candidato, a especificação é satisfeita se, e somente se, o acréscimo desse cartão a essa lista não ultrapassar o limite. O contrato desse método permite a passagem de listas nulas (argumento `cards` sem referência a qualquer objeto). Uma lista nula deve ser interpretada exatamente como uma lista vazia. Logo, se a lista for nula ou se o tamanho dela for menor que o limite, existe a possibilidade de inserção de mais um elemento. O método fornece `true` nesses casos e `false` em todos os outros.

Em Groovy, nem sempre é preciso usar a palavra-chave `return` para indicar retorno de valores do método. Se essa palavra-chave não for usada, o compilador tentará interpretar o valor da última linha executada como um valor de retorno. Neste caso, o método possui somente uma linha cujo valor, portanto, é fornecido.

Listing 4.2: Implementação da regra de limite de cartões.

```
1  class CardLimit {
2    int limit
3
4    boolean isSatisfiedBy(cards, candidate) {
5      cards == null || cards.size() < limit
6    }
7  }
```

Embora todas as classes de especificação de cartões implementem a mesma interface (que é apresentada na Figura 4.3 sob o nome `CardSpecification`), isso não precisa ser declarado explicitamente em Groovy. Dado que a linguagem é de tipagem dinâmica, os tipos dos objetos somente são checados em tempo de execução. Isso implica que os métodos não precisam declarar os tipos de seus argumentos. É o que ocorre no caso das duas classes apresentadas aqui. Caso esses métodos recebam objetos que não implementam a interface esperada, o fluxo de execução segue de acordo com o protocolo de metaobjetos, explicado na seção 4.3.2.

Em linguagens com tipagem estática, há a necessidade de declaração das interfaces e, conseqüentemente, sua inclusão na declaração de variáveis. Contudo, o princípio básico não se altera; há uma separação entre interface e implementação, o que diminui o acoplamento entre as classes.

Uma das práticas de desenvolvimento adotadas neste projeto foi a escrita de testes de unidade, de modo a cobrir o máximo possível da lógica de negócios e de interface com o usuário. A maioria dos testes foi escrita antes do código de produção (*test-first programming*). A Listagem 4.3 mostra um exemplo de teste de unidade deste projeto, criado para verificação do comportamento da classe `CardLimit`.

No primeiro teste (linha 6), verifica-se a extrapolação do limite de cartões. Dada uma lista com dois cartões e um limite definido em 3, o teste assegura que antes da inclusão de um novo cartão a essa lista, esse cartão candidato satisfaz a especificação, ou seja, não excedeu o limite. O mesmo teste assegura que a especificação não é satisfeita após a inclusão do cartão. O segundo teste (linha 13) assegura que a especificação é verdadeira para qualquer cartão, dada uma lista nula.

É importante ressaltar que não é responsabilidade do método `isSatisfiedBy` informar se a especificação é válida para lista de cartões passada como parâmetro. Sua responsabilidade é responder se o cartão candidato pode ou não ser acrescido à lista. No exemplo da Listagem 4.2, uma lista com três cartões satisfaz a especificação, mas a inclusão de um quarto cartão a essa lista a tornaria inválida. Por esse motivo, o teste da Listagem 4.3, na linha 10 assegura que um cartão candidato a ser o quarto elemento da lista não satisfaz a especificação. Conseqüentemente a comparação entre o tamanho da lista e o limite deve ser do tipo "menor que".

Listing 4.3: Testes para a classe `CardLimit`.

```
1  class CardLimitTests extends GroovyTestCase {
2
3      def cardLimitUnderTest = new CardLimit(limit: 3)
4      def cards = [new Card(), new Card()]
5
6      void testLimitExceeded() {
7          def candidate = new Card()
8          assertTrue(cardLimitUnderTest.isSatisfiedBy(cards, candidate))
9          cards << candidate
10         assertFalse(cardLimitUnderTest.isSatisfiedBy(cards, candidate))
11     }
12
13     void testNullCards() {
14         def candidate = new Card()
15         assertTrue(cardLimitUnderTest.isSatisfiedBy(null, candidate))
16     }
17 }
```

A responsabilidade das classes que implementam a interface de especificação de cartões é responder a uma única pergunta: se um dado cartão satisfaz a uma especificação, levando em conta uma lista de outros cartões. Percebe-se, portanto, que a granularidade dessas classes é bastante fina. Elas respondem à pergunta ignorando o contexto no qual essa pergunta é feita. Essa ignorância do contexto garante um baixo acoplamento, ao mesmo tempo em que maximiza o seu reúso.

Servindo-se das especificações de cartão, qualquer objeto da classe `Passenger` é capaz de determinar quais categorias lhe são permitidas, isto é, quais categorias o passageiro que ele representa pode adquirir. Para isso, a interface da classe `Passenger` contém um método `allowedCategories`, que fornece uma lista das categorias permitidas. O código desse método é apresentado na Listagem 4.4.

Listing 4.4: Método que fornece as categorias que um passageiro pode adquirir.

```
1  class Passenger
2    def specifications
3
```

```
4     def allowedCategories() {
5       def allowedList = []
6
7       Category.list().each {category ->
8         def canAttach = true
9         specifications.each {
10          canAttach &= it.isSatisfiedBy(cards, new Card(category))
11        }
12        if (canAttach)
13          allowedList << category
14      }
15      allowedList
16    }
17
18    // outros métodos e atributos omitidos
19  }
```

A fim de encontrar as categorias permitidas, o método `allowedCategories` itera sobre todas as categorias cadastradas. Verifica-se, então, se cada categoria satisfaz todas as especificações atribuídas àquele passageiro. Tais especificações estão disponíveis ao objeto `Passenger` por uma lista denominada `specifications`, declarada na linha 2. Essa lista é preenchida por injeção de dependência, de modo que o objeto pode simplesmente confiar que a lista conterá os valores corretos quando o seu uso for necessário.

As categorias cadastradas são recuperadas na linha 7, por meio de uma chamada ao método estático `Category.list`. Ocorre que esse método não existe de fato. Ele é disponibilizado em tempo de execução pelo Grails utilizando as características dinâmicas da linguagem Groovy e o protocolo de metaobjetos. Uma chamada a esse método dispara a execução de consultas ao banco de dados e conversão dos registros em objetos.

De modo geral, as tarefas de persistência de qualquer objeto do domínio podem ser feitas usando-se essa estratégia, ou seja, a chamada de métodos estáticos incluídos pelo *framework* em tempo de execução. Por garantirem uma separação dos interesses de persistência de objetos e lógica de negócios, esse método de mapeamento objeto-relacional do Grails pode ser considerado como uma implementação dos repositórios (seção 3.3.7).

Não havendo esse recurso de inclusão de métodos em tempo de execução (o que é feito automaticamente pelo *framework*), torna-se necessária a implementação explícita dos repositórios, com chamadas aos métodos definidos em tempo de compilação. De qualquer forma, a responsabilidade dos repositórios é a mesma: isolar a camada de domínio dos interesses de persistência.

Sempre que um cartão é classificado como permitido, isto é, satisfaz todas as especificações, ele é acrescentado a uma lista de cartões chamada `allowedList`, que é fornecida pelo método. Isso é feito na linha 13, usando-se o operador «, sobrecarregado nas listas para anexar objetos. Essa é outra característica da linguagem Groovy que ajuda a simplificar o código e focar no domínio: a possibilidade de sobrecarga de operadores, um recurso que outras linguagens, como C++, também possuem. A sobrecarga de operadores, entretanto, é apenas um facilitador sintático, que simplifica um pouco o envio de mensagens para os objetos.

O nível de especialização observado nas classes de especificação de cartões também está presente na classe `Passenger`, cuja reponsabilidade consiste somente em representar o conceito de passageiro e suas regras de negócio. De modo geral, essa granularidade fina é desejável em todas as classes do domínio e foram implementadas neste estudo de caso visando a esse objetivo.

Por outro lado, as histórias de usuário, que são a referência para o desenvolvimento da linha de produtos, englobam um contexto mais amplo. Por isso, é preciso que haja classes cujo escopo seja maior, envolvendo funcionalidades completas. Essas classes são os serviços da camada de aplicação. O serviço dedicado à aquisição de cartões é chamado de `CardAcquisitionService` e seu código é apresentado na Listagem 4.5.

Diretriz 5 *Funcionalidades completas, com comportamentos de granularidade grossa, podem ser encapsuladas em serviços da camada de aplicação, que, por sua vez, coordenam os comportamentos de granularidade mais alta.*

Listing 4.5: Serviço da camada de aplicação para aquisição de cartões.

```
1  class CardAcquisitionService {
2      boolean transactional = true
3
4      List allowedCategoriesFor(String cpfNumber)
5       throws InvalidCpfException {
6          def passenger = Passenger.findByCpfNumber(cpfNumber)
7          return passenger?.allowedCategories()
8      }
9
10     def createCard(categoryID, cpfNumber)
11       throws InvalidCpfException, CategoryNotAllowedException {
12         def category = Category.get(categoryID)
13         def passenger = Passenger.findByCpfNumber(cpfNumber)
14
15         if (!passenger.allowedCategories().contains(category))
16             throw new CategoryNotAllowedException()
17
18         passenger.addToCards(new Card(category))
19         if (!passenger.save()){
20              log.error passenger.errors
21         }
22     }
23 }
```

O método `allowedCategoriesFor` recebe uma `String` que identifica o passageiro (neste caso, o CPF) e fornece a lista de categorias permitidas para aquele passageiro. Ao ser invocado, esse método recupera o passageiro cadastrado com o identificador `cpfNumber` e pede a esse objeto a lista das categorias permitidas (código da Listagem 4.4).

O segundo método da classe `CardAcquisitionService` é o `createCard`, que recebe dois argumentos: o identificador da categoria e o CPF do passageiro. Com essas informações, ele é capaz de recuperar tanto a categoria quanto o passageiro cadastrados, criar um novo cartão para a categoria encontrada e associá-lo ao passageiro. Um pré-requisito para a criação do cartão é que a

categoria informada esteja entre as categorias permitidas para aquele passageiro. Caso contrário, uma exceção é lançada (linhas 15 e 16).

A classe `CardAcquisitionService`, desse modo, expõe uma API simplificada para a camada superior (de apresentação), focada na funcionalidade de aquisição de cartões. Essa classe orquestra várias operações e conceitos a fim de que a funcionalidade, como um todo, se complete. Considera-se, em vista disso, que a granularidade das classes de serviço é grossa. No entanto, o conhecimento que essas classes possuem do domínio é relativamente baixo, já que os detalhes estão encapsulados pelas classes de domínio.

A configuração das variabilidades, portanto, é basicamente uma questão de interligação de objetos, por meio de injeção de dependência. A construção de um produto da linha que possua apenas a regra de limite de cartões, por exemplo, é feita incluindo-se na lista `specifications` (linha 2, da Listagem 4.4) apenas um objeto: uma instância da classe `CardLimit`. De modo geral, há uma classe para cada regra de combinação de cartões. Incluindo as instâncias correspondentes dessas classes, é possível configurar quais regras o produto construído irá usar. A configuração das dependências é a base do método de engenharia de aplicações usado neste estudo de caso, cujos detalhes são expostos na seção 4.5.

4.4.2 Iteração 2

Na segunda iteração, foram desenvolvidas duas histórias relacionadas a crédito de passagens. A primeira trata do crédito para cartões individuais e a segunda é referente ao crédito em lote para cartões associados a usuários corporativos. Essas histórias são enunciadas, respectivamente, a seguir:

Como atendente, quero efetuar operações de crédito no cartão de um passageiro, para que a empresa de viação possa fazer arrecadação pré-paga de passagens.

Como usuário corporativo, quero conceder crédito de passagens a todos os meus funcionários, como forma de benefício trabalhista (vale-transporte).

Com essas duas novas histórias, o modelo de *features* da linha de produtos sofreu uma evolução e o resultado é mostrado no diagrama da Figura 4.4.

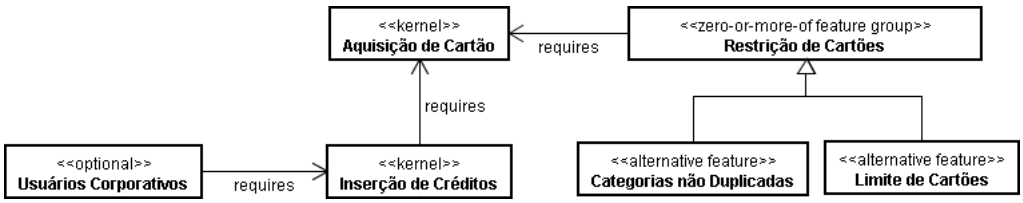


Figura 4.4: Diagrama de *features* da iteração 2.

Para a história denominada "Inserção de créditos", foram definidos os seguintes testes (ou critérios) de aceitação:

- O valor do crédito não pode ser negativo.
- O cartão que receberá o crédito deve estar cadastrado no sistema.
- Após a operação de crédito, o saldo resultante no cartão deverá ser a soma do saldo anterior com o valor creditado.
- Caso haja alguma falha, a operação deve ser abortada e deve voltar ao estado inicial.
- A operação de crédito poderá ser efetuada durante a criação do cartão.

Quanto à segunda história ("Usuários Corporativos"), os critérios de aceitação são definidos da seguinte forma:

- Deve haver uma categoria de passageiros cadastrada para funcionários associados a usuários corporativos.
- A categoria citada acima pode ser configurada pelo usuário do sistema (um administrador, por exemplo).
- O crédito informado por um usuário corporativo deve ser concedido a todos os funcionários associados.
- O crédito concedido aos funcionários aplica-se somente aos cartões da categoria criada para esse fim.

De acordo com o que foi exposto sobre a primeira iteração, a aplicação está sendo construída com base em uma arquitetura em camadas. O diagrama da Figura 4.5 mostra a organização em classes para a implementação desta funcionalidade. As classes que não são relevantes para essa discussão (como a classe `CardAcquisitionService`, por exemplo, implementada na iteração anterior) foram omitidas do diagrama.

Há três pacotes envolvidos nas funcionalidades citadas nas histórias apresentadas na seção anterior:

`passenger`: módulo relacionado a passageiros e cartões, já implementado anteriormente.

`fare`: módulo relacionado a taxas, cobranças e passagens. Por enquanto, possui apenas uma classe: `CardCreditService`, que será discutida mais adiante. Ele está planejado para abrigar outras classes, relacionadas a carga de cartão, que foram implementadas na iteração seguinte.

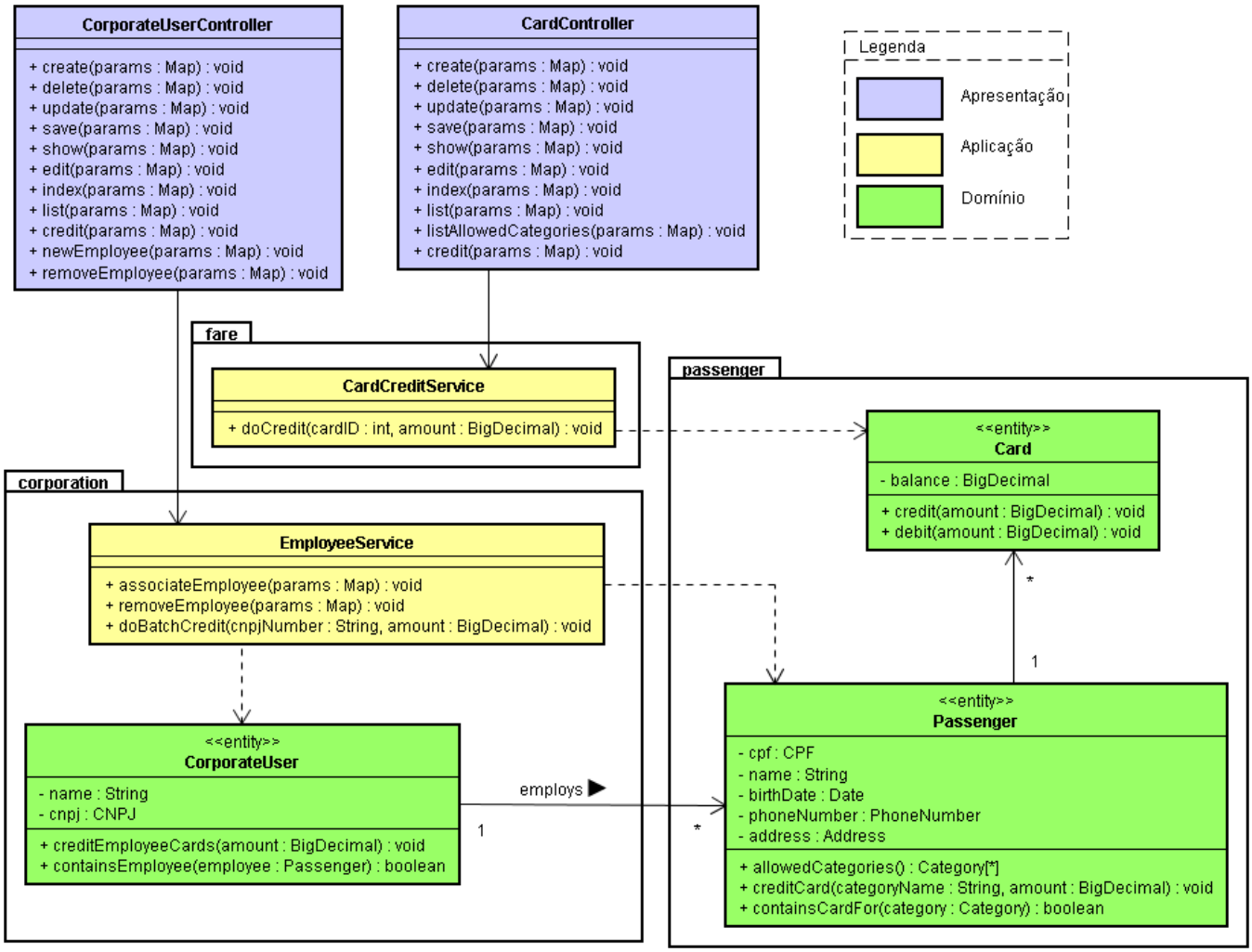


Figura 4.5: Diagrama de classes para a funcionalidade de crédito de cartões.

`corporation`: módulo criado para a variabilidade de usuários corporativos. Como essa é uma variabilidade opcional, é importante deixá-la o mais desacoplado possível da base de código existente. Por isso, ela foi encapsulada em um pacote, que possui dependência unidirecional do pacote passenger, ou seja, corporation depende de passenger, mas não o contrário.

Para acomodar as novas funcionalidades, foi preciso fazer algumas refatorações ao longo desta iteração, seguindo o princípio de refatorar constantemente e em pequenos passos, lembrando que o código está protegido por testes de unidade e de integração automatizados.

Dentre essas refatorações, a principal foi a generalização da identificação de passageiros e corporações. Anteriormente, um Passageiro era identificado por um CPF, representado por uma classe homônima, com toda a lógica de validação e comparação encapsuladas. Com a criação da classe `CorporateUser`, foi preciso criar uma classe semelhante para representar a identificação de uma empresa: a classe `CNPJ`. As duas classes, porém, compartilham a mesma interface além de uma pequena quantidade de código. Por isso, foi criada a classe `LegalIdentification`, conforme mostrado na Figura 4.6

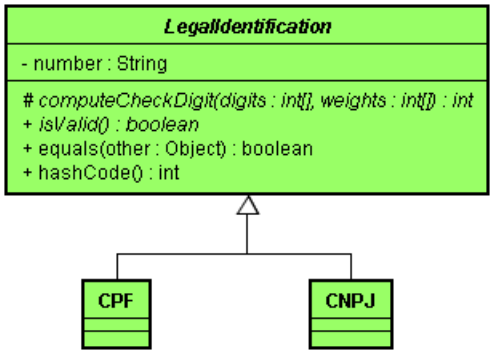


Figura 4.6: Hierarquia para identificação de passageiros e empresas.

Os métodos `isValid` e `computeCheckDigits` são abstratos, deixando a implementação da regra de validação específica para as subclasses. Outra refatoração importante foi a remoção da relação de dependência entre as classes `CardAcquisitionService` e `CardSpecification`. Essa relação existia apenas para que os objetos da classe Passenger pudessem receber uma referência à especificação em tempo de execução. Essa complexidade mostrou-se desnecessária, já que é possível injetar essa dependência pelo próprio Grails, apenas declarando-a no arquivo de configuração do Spring.

Diretriz 6 *Antes de iniciar o desenvolvimento das histórias (a partir da segunda iteração), procure pontos do código que possam ser refatorados, de modo a simplificar a implementação de novas funcionalidades.*

É importante lembrar que a classe `LegalIdentification` representa apenas o identificador dos usuários do sistema de transporte, sejam eles usuários corporativos ou passageiros (pessoas físicas). Como esses usuários são entidades, eles possuem identidade e continuidade temporal, garantidos pela unicidade desses identificadores. Os demais atributos, como nome e endereço, por exemplo, não compõem a identidade desses usuários. Em tese, todos esses atributos podem ser modificados ao longo do tempo, enquanto a identidade dos usuários se mantém inalterada.

Dois novos serviços da camada de aplicação foram adicionados: `CardCreditService` e `EmployeeService`. A primeira expõe uma interface para crédito de uma determinada quantidade em um determinado cartão. A segunda classe é responsável principalmente pela associação entre Passageiros e Usuários Corporativos. No entanto, ela também oferece um método para crédito em lote para todos os funcionários de uma determinada empresa.

Segundo o Princípio da Responsabilidade Única, essa abordagem é ruim: a classe é responsável por mais de uma tarefa e, conseqüentemente, possui mais de uma razão para mudar. Nesse sentido, seria melhor transferir esse método para a classe `CardCreditService`. Por outro lado, pensando em termos de extensibilidade dentro da linha de produtos, faz mais sentido deixar essa responsabilidade dentro do pacote `corporation`. A variabilidade de Usuários Corporativos é atômica, no sentido de que as funcionalidades relacionadas estão todas presentes ou todas ausentes. Assim, é razoável deixá-las todas dentro do mesmo pacote.

Uma terceira alternativa seria criar um outro serviço dentro do pacote `corporation`, cuja única responsabilidade seria a coordenação de crédito em lote. Essa solução, porém, iria aumentar a complexidade do projeto desnecessariamente. Caso haja algum requisito futuro que justifique essa divisão, as classes podem ser refatoradas nessa direção. Este é um clássico exemplo de *trade-off*, em que vale o bom-senso para escolher a melhor solução dentro do contexto.

Diretriz 7 *Uma variabilidade opcional que compreenda várias camadas deve ser encapsulada em um único pacote. As classes dentro desse pacote podem possuir dependências em relação a classes do núcleo (e somente do núcleo), mas o contrário não deve ocorrer.*

Na camada de apresentação, foi criado um controlador, `CorporateUserController` (e suas respectivas visões), para a variabilidade de Usuários Corporativos, no que diz respeito a interface com o usuário. O controlador `CardController`, já existente, recebeu uma nova ação, `credit`.

4.4.3 Iteração 3

Em um sistema de transporte público municipal, a principal utilidade das categorias de passageiros é a possibilidade de aplicar regras de negócio específicas. Um exemplo disso é a concessão de descontos diferentes por categoria, que não foi implementada nas quatro iterações feitas neste estudo de caso. No entanto, há uma história no *backlog* prevendo a implementação dessa funcionalidade.

Estudantes ou idosos, por exemplo, podem ter direito a 50% de desconto por determinação legal. A aplicação de descontos pode ser motivada também por razões comerciais. Porém, nos casos em que a taxa de desconto é muito alta (digamos, mais de 40%), a empresa de viação corre um risco de perda financeira razoavelmente alto. Para contornar esse problema, uma das soluções de negócio encontradas é limitar o número de passagens que podem ser inseridas em um cartão no período de um mês.

O limite de passagens por mês para os cartões é uma funcionalidade opcional. Como tal, ela pode ser acrescentada ao núcleo do sistema, dependendo da necessidade do cliente. A história de usuário escrita para essa funcionalidade é enunciada abaixo:

A empresa de transporte municipal deseja limitar a quantidade de passagens que um usuário pode comprar por mês, para evitar perdas financeiras.

O modelo de *features* da iteração 3 é mostrado na Figura 4.7, incluindo as duas novas histórias planejadas no início da iteração e enunciadas pelas histórias acima.

Os critérios de aceitação para essa história são detalhados a seguir:

- Cada categoria pode possuir um limite diferente de passagens. Caso não seja definido um limite para uma determinada categoria, considera-se que ela é ilimitada.

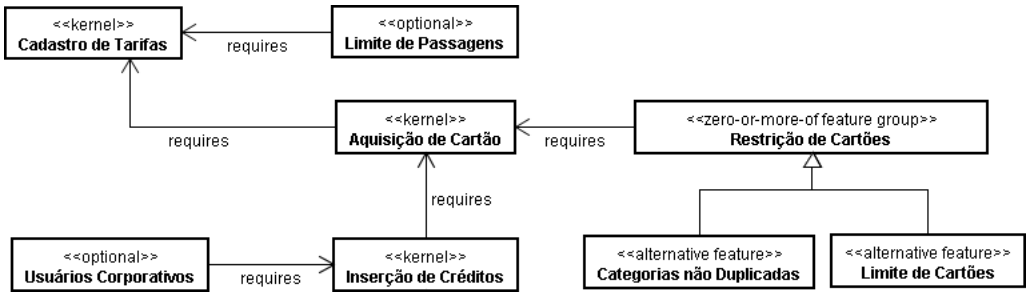


Figura 4.7: Diagrama de *features* incluindo as histórias implementadas na iteração 3.

- Esses limites equivalem à quantidade máxima de passagens que podem ser inseridas em um cartão no período de um mês.
- Enquanto um cartão não atingir o limite definido pela sua categoria, não há restrições quanto ao número de cargas que o passageiro pode fazer.
- Mesmo havendo restrição por número de passagens, a inserção de créditos é feita por valor monetário.
- O período de um mês começa a ser contado a partir da data da primeira carga. Essa data de início da contagem é usada como data de referência.
- Passado um mês da data de referência, o limite do cartão é "reiniciado", ou seja, pode-se inserir a quantidade limite determinada pela sua categoria.

Um dos critérios define que a inserção de créditos continuará sendo feita por valor monetário, mesmo nos casos em que haja limite de passagens. Isso quer dizer que, se a tarifa vigente que é cobrada pela passagem é R$ 2,30, por exemplo, e um determinado cartão está associado a uma categoria cujo limite é de 40 passagens por mês, esse cartão poderá receber, no máximo, R$ 92,00 de crédito, em intervalos mensais.

Conseqüentemente, o valor vigente da tarifa deve ser levado em conta, tanto na inserção quanto na exibição da quantidade de créditos permitidos para os cartões. Por isso, foi preciso implementar, antes, uma história relativa ao cadastramento de tarifas de transporte:

A empresa de transporte municipal precisa cadastrar tarifas para que possa cobrar pelas passagens.

Ao contrário da história anterior, esta representa uma funcionalidade do núcleo do sistema. Os critérios de aceitação definidos para essa história são:

- Pode-se cadastrar uma tarifa a qualquer momento.
- Uma tarifa, ao ser cadastrada, torna-se automaticamente vigente, revogando as tarifas anteriormente cadastradas.

- As tarifas cadastradas anteriormente não são removidas. Ficam armazenadas para possíveis consultas no futuro.

A história relativa ao cadastro e consulta de tarifas foi implementada usando um serviço de domínio, denominado `FareService`. Embora a tarefa de encontrar o valor vigente esteja relacionada a questões de persistência, a decisão de colocá-la em um serviço foi motivada pela particularidade do processamento. As tarefas mais comuns de persistência, como salvar ou atualizar um objeto, remover ou fazer buscas por critérios simples, já são fornecidas pelo Grails por seu mecanismo de metaprogramação. Justamente por serem bastante comuns, essas tarefas são expostas ao domínio na forma de uma API simples, em geral uma chamada de método para cada um desses comandos ou consultas. No caso de descobrir o valor vigente da tarifa, o processo é ligeiramente mais complexo. A Listagem 4.6 mostra o código da classe `FareService`.

Listing 4.6: Código da classe `FareService`.

```
1  package fare
2
3  import org.joda.time.LocalDate
4
5  class FareService {
6
7      Fare effectiveFare() {
8          def c = Fare.createCriteria()
9          def lastDate = c.get {
10             projections {
11                 max("date")
12             }
13         }
14
15         def d = Fare.createCriteria()
16         def effectiveFare = d.get {
17             eq("date", lastDate)
18         }
19
20         return effectiveFare
21     }
22 }
```

A classe `FareService` possui apenas o método `effectiveFare`, que fornece a tarifa vigente. Segundo o requisito, a tarifa considerada vigente é aquela que foi cadastrada mais recentemente. Para encontrar a tarifa que atenda a essa especificação, o método encontra primeiro a data mais recente dentre as tarifas cadastradas (linhas 8–13). Em seguida, a tarifa que corresponde a essa data mais recente é recuperada (linhas 15–18). Por fim, a tarifa é fornecida na linha 20.

Para essas consultas são usados os construtores de critérios de consulta, que são obtidos por uma chamada ao método `createCriteria` (linhas 8 e 15), injetado nas classes pelo mecanismo de metaprogramação da linguagem Groovy. No exemplo da Listagem 4.6, foram usadas uma projeção para encontrar a data mais recente dentre as tarifas (linha 10), utilizando-se o operador `max`. O resultado de uma projeção é um subconjunto dos atributos da classe sobre a qual a projeção é aplicada. Neste caso, o subconjunto é formado apenas pela data da tarifa. Na linha 17, é usada

uma condição que restringe o conjunto de objetos a um outro conjunto contendo apenas a tarifa mais recente, por meio do operador `eq`.

Diretriz 8 *Regras de negócio reusáveis, independentes de estado e que não se encaixem em nenhuma entidade ou objeto de valor podem ser modeladas como serviços de domínio.*

Conforme já citado algumas vezes, o conceito de Cartão é central ao domínio de transportes municipais. A maioria das regras de negócio envolve, de alguma forma, essa entidade. Com a funcionalidade de limite de passagens ocorre o mesmo. O limite de passagens de uma categoria influencia a quantidade de créditos que um cartão pode receber.

A regra de limite de passagens é uma variabilidade opcional, o que implica que no núcleo da linha de produtos, a quantidade de passagens é ilimitada para todas as categorias. Conseqüentemente pode-se inserir qualquer valor de crédito a qualquer momento, para qualquer cartão. Sendo uma variabilidade opcional, é extremamente importante manter a linha flexível, de tal modo que se possa incluir ou não essa funcionalidade, sem afetar o restante do sistema.

Neste caso, a estratégia empregada foi criar uma subclasse de `Card`, chamada `LimitedCard`, que acrescenta três atributos à classe base: `accumulatedTickets`, `referenceCreditDate` e `unusedBalance`. O primeiro denota a quantidade de passagens acumuladas naquele período de um mês. O segundo corresponde à data de referência, ou seja, a data a partir da qual é contado o período de um mês em que vale o limite de passagens. O terceiro corresponde ao saldo não usado, ou seja, o resto da divisão do saldo pelo valor da passagem. Para compreender melhor a utilidade desse atributo, considere o seguinte cenário: o preço da passagem é de R$ 2,00 e o passageiro inseriu R$ 81,00 no cartão. Como resultado, terão sido inseridas 40 passagens e ficará um saldo não usado de R$ 1,00. Esse saldo fica armazenado e pode ser usado em uma inserção futura.

Quanto aos métodos, são acrescentados dois novos: `canInsert`, que indica se um determinado valor de crédito pode ser inserido (esse método é utilizado mais com o propósito de simplificar a API) e `updateAccumulatedInfo`, um método privado que atualiza a data de referência e a quantidade de passagens acumuladas. Esse método é chamado a partir do método `insert`, presente na classe base e sobrescrito por esta subclasse.

A classe `LimitedCard` observa o Princípio da substituição de Liskov (Liskov e Wing, 2001), ou seja, as classes que fazem referência à classe Card não serão afetadas caso se use a classe `LimitedCard` no lugar. Isso permite que se inclua ou não essa variabilidade em um produto. Para todo o restante do sistema, é indiferente qual classe está sendo usada para representar o conceito de Cartão. O diagrama de classes da Figura 4.8 mostra uma representação do modelo de domínio do sistema BET na terceira iteração.

Até a segunda iteração, existia somente um tipo de cartão, representado pela classe `Card`. Conforme apresentado anteriormente, os objetos da classe `Card` eram instanciados diretamente pelo serviço `CardAcquisitionService`. Com a introdução da classe `LimitedCard` no modelo de

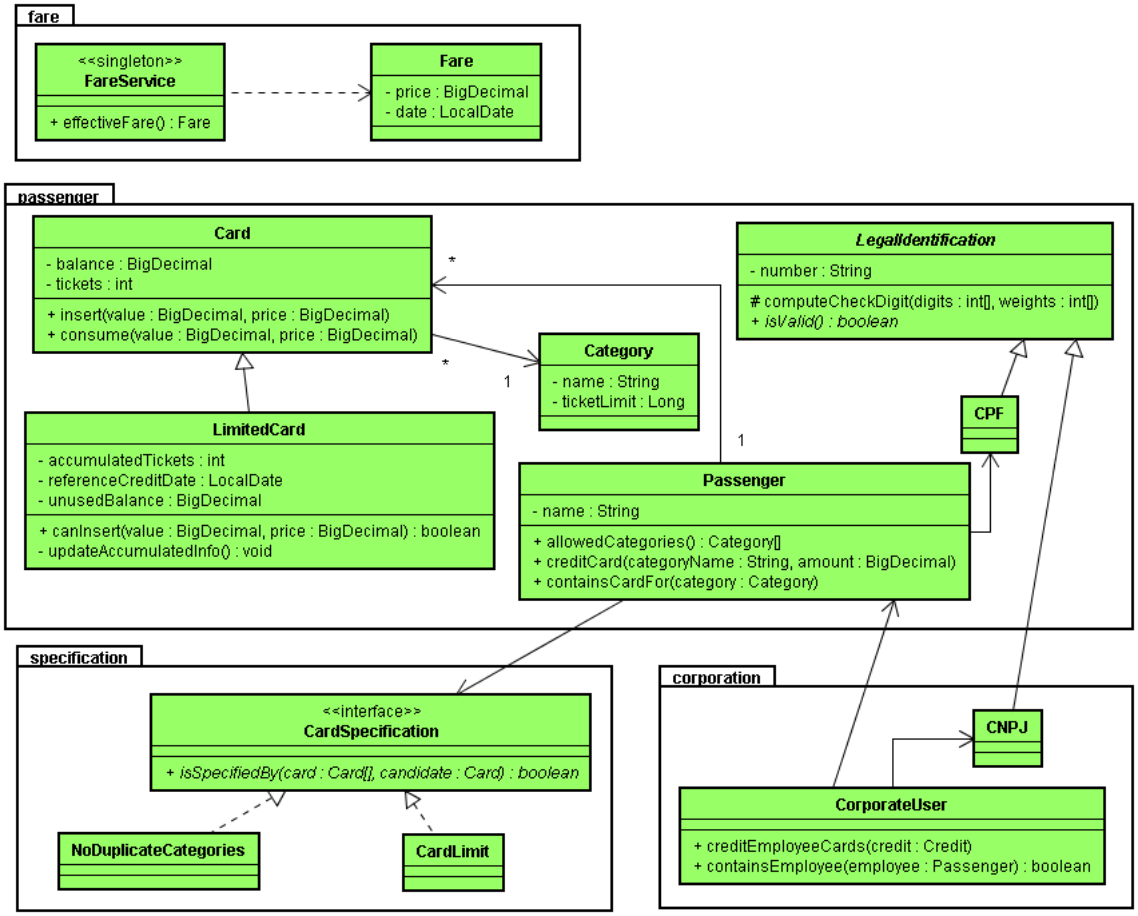


Figura 4.8: Modelo de domínio do sistema BET na terceira iteração.

domínio, foi preciso desacoplar a criação do objeto que representa o cartão do serviço de aquisição de cartões.

Para alcançar tal desacoplamento, foi criada uma fábrica (*factory*) destinada a produzir cartões sob demanda, cujo código é apresentado na Listagem 4.7. Essa fábrica esconde um importante detalhe de implementação: qual é a classe à qual pertence efetivamente o cartão produzido, isto é, se ele é uma instância de `Card` ou sua especialização, `LimitedCard`. Dado que a subclasse obedece o Princípio da Substituição de Liskov, o serviço de aquisição de cartões (ou qualquer outra classe) não precisa conhecer esse detalhe. Conseqüentemente, a instanciação direta (via operador `new`) foi substituída por uma chamada ao método `create`, da `CardFactory`.

Diretriz 9 *Variabilidades opcionais que alteram as regras de negócio relativas unicamente a um conceito do domínio podem ser implementadas com polimorfismo, desde que respeitem o Princípio da Substituição de Liskov.*

Diretriz 10 *Sempre que houver uma hierarquia de classes responsável pelas variabilidades relacionadas a um conceito, é aconselhável que se encapsulem a criação desses objetos em uma fábrica.*

Listing 4.7: Código da classe `CardFactory`.

```
1  package passenger
2
3  import passenger.Card
4  import passenger.LimitedCard
5
6  class CardFactory {
7      def cardType
8
9      Card create(category, balance){
10         if (!category) {
11             def msg = "Cartões somente podem ser criados se houver uma categoria para associar."
12             throw new RuntimeException(msg)
13         }
14         def card
15         if (cardType == "limited") {
16             card = new LimitedCard(category)
17         }
18         else {
19             card = new Card(category)
20         }
21         if (params.balance)
22             card.balance = new BigDecimal(balance)
23         return card
24     }
25 }
```

A classe `CardFactory` possui um atributo chamado `cardType`, que é atribuído por injeção de dependência. De acordo com o seu valor, que pode ser `"limited"` ou `"regular"`, ele produz, respectivamente, um cartão com limite de passagens ou um cartão comum.

4.4.4 Iteração 4

Todas as histórias implementadas nas iterações anteriores estão relacionadas à criação de cartões ou realização de operações de inserção de crédito. Na quarta iteração foi implementada a história que faltava para que o produto pudesse ser considerado pronto para lançamento de uma primeira versão operacional do sistema. Essa história é enunciada a seguir:

A empresa de transporte municipal deseja que os créditos armazenados nos cartões sejam consumidos a cada viagem, para manter atualizados os saldos dos passageiros.

O modelo de *features* final (considerando que esta é a última iteração) é apresentado no diagrama da Figura 4.9. Nele estão representadas todas as funcionalidades planejadas para a linha de produtos no momento do seu lançamento.

Os critérios de validação para essa história estão listados abaixo:

- Ao aproximar o cartão do validador, o número de passagens do cartão é decrescido de 1.
- Cartões sem passagens (número de passagens igual a zero) são recusados pelo validador.

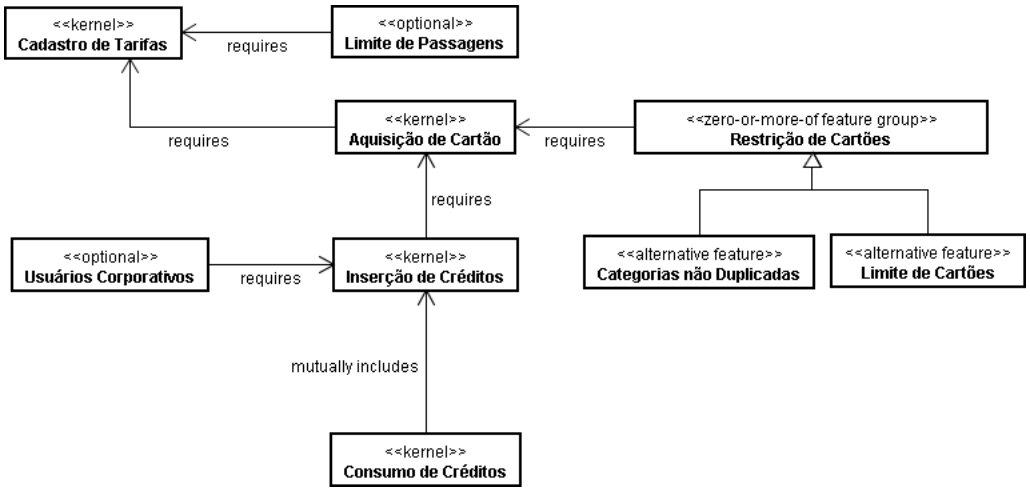


Figura 4.9: Diagrama de *features* da linha de produtos, pronta para lançamento.

- O débito de uma passagem libera a catraca. Ao girar, a catraca é travada.
- O saldo do cartão é exibido pelo visor do validador após o débito da passagem.

Em termos de arquitetura, a implementação desta história é um pouco diferente das demais. Isto porque trata-se de um controle distribuído. De um lado, vários validadores (tanto nos ônibus quanto nos terminais) processam os cartões dos passageiros que desejam fazer viagens. Do outro lado, há um ou mais servidores, que recebem as requisições e as processam de acordo com as regras do domínio vigentes para o produto em questão (incluindo as variabilidades). Em função dessa comunicação, as aplicações-cliente que executam nos controladores são capazes de efetuar o controle local necessário, como exibir informações no visor e controlar a liberação da catraca.

Em função do baixo acoplamento entre o cliente e o servidor, e levando em conta o fato de que não há variabilidades previstas para o cliente, optou-se por reusar uma aplicação pronta ao invés de implementar outro cliente[3]. A comunicação entre cliente e servidor é feita por um método de invocação remota de métodos baseada no protocolo HTTP, denominada HTTP invoker (Walls e Breidenbach, 2007).

Sempre que um passageiro aproxima um cartão do validador, este envia comandos ao servidor informando-o de que um passageiro requisita a liberação da catraca. Junto com esse comando é enviado um identificador do cartão. Ao receber essa chamada, o servidor deve decidir se o passageiro tem direito ou não à passagem e, em caso afirmativo, debitar do cartão uma passagem e registrar uma viagem. Em qualquer das situações, deve devolver uma resposta ao validador com um código de status da operação, que pode indicar que o pagamento foi efetuado ou não com sucesso. Caso o código do cartão seja inválido, um outro código deve ser fornecido.

A cada cartão está associado o número de passagens que ele possui, o qual é calculado no momento do crédito, levando em conta a tarifa vigente. Desse modo, o servidor é capaz de responder se aquele cartão dá direito ou não à passagem.

[3]Essa aplicação é, de fato, a implementação de referência, discutida na seção 4.7.

Em termos de implementação, essa funcionalidade consistiu apenas da implementação de um método, que é invocado remotamente pelo validador. Relembrando a arquitetura em camadas empregada neste caso de uso, o usuário interage somente com a camada de apresentação. Ora, pode-se considerar que o validador (juntamente com seus dispositivos de entrada e saída) atua como uma interface de usuário. Por isso, não é preciso implementar nada na camada de apresentação, do lado do servidor.

A classe mais adequada, portanto, para se implementar esse método remoto deve ser algum serviço da camada de aplicação. Dado que já existe um serviço para operações de crédito em cartões, denominado `CardCreditService`, essa classe foi a escolha mais natural. A implementação desse método, denominado `processarTransacao` é mostrada na Listagem 4.8. O nome e a assinatura desse método estão definidos em uma interface, denominada `IProcessarTransacao`, da implementação de referência. O cliente depende apenas da interface, o que permite a troca de implementações, sem que qualquer outra parte do sistema precise ser alterada. O método privado `price()`, que não foi mostrado na Listagem 4.8 e é invocado nas linhas 6 e 9, fornece o preço vigente da tarifa, utilizando-se do serviço de domínio codeFareService.

Diretriz 11 *Os detalhes do domínio devem ser abstraídos pela camada de aplicação sempre que houver a necessidade de servir a clientes diferentes, como controladores MVC e aplicações hospedadas em dispositivos externos.*

Listing 4.8: Método que processa as transações de débito de passagens.

```
1   public String processarTransacao(int code, int cardID, int validatorID) {
2       Card card = card.get(cardID)
3       if (!card) {
4           return CARD_NOT_OK;
5       }
6       def ticketPrice = new BigDecimal(price())
7       try {
8           def trip = new Trip()
9           card.consume(ticketPrice, price())
10          if (!card.save())
11              log.info "could not save: ${card.errors}"
12          trip.card = card
13          trip.save()
14          return "$PAYMENT_OK ${card.credit}"
15      } catch (InvalidCreditException e) {
16          return PAYMENT_NOT_OK
17      }
18  }
```

4.4.5 Resumo das diretrizes

As diretrizes de desenvolvimento apresentadas ao longo das últimas seções, relacionadas às iterações feitas na implementação do sistema BET, estão relacionadas na Tabela 4.1.

Tabela 4.1: Resumo das diretrizes de desenvolvimento baseado em DDD e métodos ágeis.

1	Ao elaborar as histórias de usuário, escreva os testes de aceitação, de modo que eles sirvam de base para a execução de testes manuais e para a escrita de testes automáticos
2	Dentre os conceitos do domínio envolvidos na história sendo implementada, identifique quais são entidades e quais são objetos de valor
3	Identifique os conjuntos de entidades e objetos de valor que componham um conjunto coeso. Se houver uma entidade principal e invariantes semânticas que se apliquem ao conjunto como um todo, esse conjunto forma um agregado. As operações de persistência devem atuar sobre todo o agregado, respeitando essas invariantes
4	Variabilidades alternativas expressas como predicados lógicos relativos a um determinado conceito podem ser modeladas como especificações
5	Funcionalidades completas, com comportamentos de baixa-granularidade, podem ser encapsuladas em serviços da camada de aplicação, que, por sua vez, coordenam os comportamentos de granularidade mais alta
6	Antes de iniciar o desenvolvimento das histórias (a partir da segunda iteração), procure pontos do código que possam ser refatorados, de modo a simplificar a implementação de novas funcionalidades
7	Uma variabilidade opcional que compreenda várias camadas deve ser encapsulada em um único pacote. As classes dentro desse pacote podem possuir dependências em relação a classes do núcleo (e somente do núcleo), mas o contrário não deve ocorrer
8	Regras de negócio reusáveis, independentes de estado e que não se encaixem em nenhuma entidade ou objeto de valor podem ser modeladas como serviços de domínio
9	Variabilidades opcionais que alteram as regras de negócio relativas unicamente a um conceito do domínio podem ser implementadas com polimorfismo, desde que respeitem o Princípio da Substituição de Liskov
10	Sempre que houver uma hierarquia de classes responsável pelas variabilidades relacionadas a um conceito, é aconselhável que se encapsulem a criação desses objetos em uma fábrica
11	Os detalhes do domínio devem ser abstraídos pela camada de aplicação sempre que houver a necessidade de servir a clientes diferentes, como controladores MVC e aplicações hospedadas em dispositivos externos

4.5 Engenharia de aplicações

No domínio de sistemas de transporte urbano, as regras de negócio variam de cidade para cidade. Das diversas regras implementadas neste trabalho e discutidas iteração por iteração, algumas delas são variabilidades. No contexto do sistema BET, isso equivale a dizer que algumas dessas regras são úteis para determinadas cidades e para outras não, no caso de variabilidades opcionais. Variabilidades alternativas podem ser entendidas como regras de negócio cujos detalhes variam de cidade para cidade, bem como as combinações entre essas regras.

Por outro lado, muitas funcionalidades são comuns a todos os sistemas, como também foi exemplificado pelas implementações da linha de produtos BET. Isso vale tanto para a implementação de referência quando a implementação baseada em DDD. A coexistência entre funcionalidades comuns e variáveis é um grande desafio para a engenharia de software, pois é bastante desejável que as funcionalidades comuns possam ser reusadas por todos os produtos. Analogamente, as fun-

cionalidades variáveis devem estar facilmente acessíveis e desacopladas, de modo que a criação de um novo produto consista basicamente em selecionar componentes de um repositório e montá-los, sem que qualquer trabalho extra seja necessário. Em outras palavras, procura-se maximizar o reúso.

O repositório de componentes também é conhecido como o conjunto de "ativos centrais" (*core assets*) da linha de produtos, segundo a nomenclatura proposta por Clements e Northrop (2002). A criação de novos produtos a partir dos ativos centrais é conhecida como instanciação de um produto.

Como a maximização do reúso é um dos objetivos fundamentais para a engenharia de aplicações, todas as decisões de projeto foram tomadas com base no reúso. Nesse sentido, vários princípios de orientação a objetos foram levados em conta, como polimorfismo, o princípio aberto-fechado, o princípio da substituição de Liskov, entre outros. Padrões de projeto e arquiteturais sugeridos pelos proponentes de DDD também foram empregados. Além disso, foi feito um uso intenso da técnica conhecida como injeção de dependências, em que a instanciação e interligação dos objetos é separada da lógica de negócios.

Em conjunto, todos esses princípios e técnicas tornaram possível a criação de um conjunto de ativos centrais implementados de tal modo que a instanciação de um novo produto corresponde exatamente à reconfiguração do grafo de dependências dos objetos, ou seja, escolhendo-se implementações diferentes das interfaces e interligando-se os objetos de modo diferente, obtém-se comportamentos diferentes.

Um corolário dessas proposições é que não é necessário trabalhar com geração de código-fonte da aplicação. Todo o processo de instanciação passa pela injeção de dependências, apenas. Esse processo consiste basicamente na alteração do arquivo que descreve o contexto da aplicação. Esse arquivo (em se tratando do *framework* Spring) é geralmente um arquivo XML contendo descrições de todos os objetos que se deseja configurar, conectados de modo a se descrever suas inter-dependências.

Para simplificar um pouco mais essa descrição de objetos e suas dependências, o Grails oferece dois facilitadores: uma linguagem específica para configuração das dependências no Spring e o conceito de "convenção mais que configuração" (*convention over configuration*). O primeiro permite que se escreva um arquivo de configuração bem mais simples, por evitar todos os ruídos e duplicações presentes naturalmente em um arquivo XML. O segundo reduz ainda mais a quantidade de código necessário, já que o *framework* é capaz de interpretar um conjunto de padrões (convenções) e decidir qual é a ação a ser tomada, sem que, para isso, necessite de intervenção explícita do desenvolvedor.

O resultado da aplicação dessas técnicas e princípios na implementação do sistema BET é que toda a configuração do sistema pode ser feita usando um arquivo de configuração pequeno e fácil de entender. A Listagem 4.9 contém um código de exemplo, que poderia ser usado para configurar um produto da linha.

Listing 4.9: Exemplo de arquivo de configuração do Spring para o sistema BET.

```
1  import specification.*
2  import variability.*
3  import passenger.*
4  import fare.*
5
6  beans = {
7    cardLimit(CardLimit) {limit = 2}
8    noDuplicateCategories(NoDuplicateCategories) {}
9    cardCombination(CardCombinationRules){
10         selected = [cardLimit, noDuplicateCategories]
11   }
12
13   cardFactory(CardFactory){
14         cardType = 'limited'
15   }
16
17   features(OptionalFeatures){
18         selected = ['corporate user']
19   }
20 }
```

Nesse exemplo, as quatro primeiras linhas são usadas para importar as classes que estão envolvidas na configuração da aplicação. O código entre as linhas 6 e 20 é o conteúdo principal para essa configuração. Nesse trecho estão presentes as definições de objetos que o Spring deve instanciar. São os chamados *beans*, no jargão do Spring. Cada *bean* configura um objeto da aplicação e contém três elementos principais: um nome, a classe à qual o objeto pertence e uma lista de propriedades com os seus respectivos valores definidos. O nome é o primeiro identificador, seguido da classe entre parênteses. As propriedades são listadas entre chaves, logo após a definição da classe. O valor de uma propriedade pode ser um tipo primitivo, como um inteiro, ou uma referência a outro objeto (representada no arquivo de configuração por um outro *bean*).

Na linha 7 é declarado um *bean* cujo nome é `cardLimit` e a classe é `CardLimit`. Esse *bean* possui apenas uma propriedade, chamada `limit`, um atributo do tipo inteiro, que neste caso está configurado para valer 2. Relembrando o diagrama de classes do sistema (Figura 4.8), essa classe `CardLimit` corresponde a uma das classes usadas na variabilidade de combinações de cartões, seguindo o padrão *Specification*. Em termos da linguagem do domínio, essa configuração diz que cada passageiro pode adquirir, no máximo, 2 cartões. Ao ler essa configuração, o Spring irá instanciar um objeto da classe `CardLimit` com o atributo `limit` valendo 2. Todos os outros *beans* que possuírem um atributo com esse nome o receberão por injeção.

Na linha 8 é declarado outro *bean* relacionado às regras de combinação de cartões. Nesse caso, o Spring instancia um objeto da classe `NoDuplicateCategories`, sem atributos, que implementa a regra do domínio que proíbe categorias duplicadas no conjunto de cartões de um passageiro. Esse *bean* recebe o nome de `noDuplicateCategories`.

Da linha 9 à linha 11, é declarado um *bean* chamado `cardCombination`, pertencente à classe `CardCombinationRules`. Essa classe possui uma única propriedade: `selected`. O valor dessa propriedade é uma lista de objetos que, em termos de domínio, corresponde às regras de combi-

nação de cartões selecionadas para estar presentes neste produto. Observe que essa lista contém dois elementos: `cardLimit` e `noDuplicateCategories`. Não por coincidência, esses são os mesmos nomes dos *beans* definidos nas duas linhas anteriores. Na linguagem do domínio, pode-se dizer que foram selecionadas duas regras de combinação de cartões: categorias não duplicadas e quantidade de cartões por passageiro restrita a dois. Caso se desejasse selecionar apenas a regra de categorias não duplicadas, por exemplo, bastaria que a propriedade `selected` recebesse uma lista contendo somente o *bean* `noDuplicateCategories`. Ou, se fosse necessário incluir uma nova regra, seria preciso declarar um novo *bean* para essa regra (nos moldes do que foi feito nas linhas 7 e 8) e incluir o nome desse *bean* na lista `selected`.

A classe `CardCombinationRules` é apenas uma classe auxiliar criada para facilitar a injeção de dependência. A classe `Passenger` possui um atributo chamado `cardCombination`. Assim, o Spring instancia um único objeto da classe `CardCombinationRules` e o injeta em todos os objetos da classe `Passenger`, que eventualmente forem criados. Os objetos `Passenger`, quando precisarem das regras de combinação selecionadas pelo usuário, consultarão esse objeto, pedindo a lista das especificações selecionadas.

Da linha 13 à linha 15, é declarado o *bean* `cardFactory`, da classe `CardFactory`, que possui um atributo do tipo String chamado `cardType`. Os valores aceitos para essa propriedade são `"regular"` e `"limited"`, para indicar à aplicação que tipo de cartão deve ser usado (comum ou com limite de passagens, respectivamente). A classe `CardFactory`, como o próprio nome sugere, é uma fábrica de cartões, ou seja, uma implementação do padrão *Simple Factory* (Cooper, 2003), cujo produto é um objeto da classe `Card` ou suas subclasses. Atualmente, há apenas uma subclasse: `LimitedCard`. Embora estejam sendo usados apenas dois tipos, a classe é extensível e pode abrigar um número qualquer de tipos que podem ser produzidos.

Da linha 17 à linha 19, é feita a configuração das demais funcionalidades opcionais da linha de produtos (já que limite de passagens é uma delas, mas possui uma forma de configuração especial). Atualmente, a única funcionalidade que se enquadra nessa categoria é a de usuários corporativos. Essas funcionalidades possuem em comum a característica de estarem associadas a um controlador MVC específico. Desse modo, a "porta de entrada" do sistema para a funcionalidade de usuários corporativos, por exemplo, é o controlador `CorporateUserController`. A funcionalidade somente estará presente em um produto se esse controlador estiver disponível para a aplicação como um todo, em particular para as views, do MVC.

A classe `OptionalFeatures` funciona como uma fábrica, de modo bastante similar à classe `CardFactory`. Seu único método público é o `selectedControllers`, que fornece uma lista de controladores MVC, que a aplicação tem à sua disposição para usar. Para decidir quais são esses objetos, a classe `OptionalFeatures` usa a lista selected, informada pelo usuário por meio do arquivo de configuração (linha 18). Com essa lista, a classe aplica a seguinte convenção de nomes: para cada String, possivelmente contendo várias palavras separadas por espaço, ela procura um controlador MVC com o mesmo nome, apenas removendo os espaços e transformando para maiúsculas todas as letras que vierem logo após um espaço em branco. O sufixo `Controller` é

anexado ao nome resultante. No exemplo apresentado aqui, a lista contém apenas um elemento: `'corporate user'`. Segundo a convenção, tal nome corresponde a um controlador chamado `corporateUserController`. O propósito dessa convenção é tornar o código de configuração mais próximo à linguagem do domínio, escondendo os detalhes de classes, controladores, etc.

Como nem todos os controladores são opcionais, ou seja, há determinados controladores que devem sempre estar presentes, seria um trabalho desnecessário (além de tornar o código um pouco mais confuso) listar no arquivo de configuração todos os controladores que se deseja selecionar. Por esse motivo, foi acrescentada a seguinte regra: por padrão, todos os controladores são obrigatórios. Para indicar que um controlador é opcional, basta incluir nele um atributo booleano optional, com valor true. Isso indica para a aplicação que tal controlador é opcional e somente irá ser disponibilizado se o seu nome (segundo a convenção abordada no parágrafo anterior) estiver na lista das funcionalidades opcionais selecionadas. Os controladores que não possuírem esse atributo serão sempre incluídos.

Em suma, o código da Listagem 4.9 é um exemplo de uma configuração possível para um produto da linha. Neste caso, o produto configurado possui as seguintes características: cada passageiro tem o direito de adquirir, no máximo, 2 cartões, e dentro do conjunto de cartões de cada passageiro não pode haver categorias duplicadas. A quantidade de passagens que os passageiros podem comprar no período de um mês também é limitada (o limite varia de categoria para categoria e pode ser definido na própria aplicação). Por fim, a aplicação permite que se façam cargas em lote para cartões de funcionários de empresas associadas (usuários corporativos).

Com esse arquivo de 20 linhas, é possível configurar todas as variabilidades de um produto, dentro do que já foi implementado. De modo geral, para se alterar a configuração de uma determinada variabilidade, o esforço requerido consiste em alterar apenas uma linha. Por exemplo, suponha que seja criada mais uma variabilidade opcional, relativa à geração de relatórios financeiros. Suponha, ainda, que o controlador responsável por essa funcionalidade se chame `financialReportController`. Para incluí-lo na lista das funcionalidades opcionais selecionadas, bastaria incluir o nome `'financial report'` na lista, da seguinte forma:

```
1 features(OptionalFeatures){
2   selected = ['corporate user', 'financial report']
3 }
```

Agora, considere que o objetivo é gerar um produto em que não haja limite de passagens, ou seja, os cartões devem ser comuns, permitindo a compra de qualquer quantidade de passagens, a qualquer momento. Nesse caso, a alteração consistiria em alterar uma String na linha 14, mudando o valor da propriedade `cardType` para `'regular'`:

```
1 cardFactory(CardFactory) {
2   cardType = 'regular'
3 }
```

Uma vez compreendida a semântica dessa linguagem, o trabalho de geração de uma aplicação executável é trivial: como os programas compilados em Groovy são executados por uma máquina

virtual Java e a aplicação neste caso é uma aplicação *web*, basta gerar um arquivo WAR (*web application archive*). Esses arquivos, por fazerem parte do padrão de aplicações *web*, definidos pela Sun (ver Jendrock (2006)), podem ser implantados em qualquer container *web*, como o Jetty ou o Tomcat. A geração desses arquivos também é simples. O próprio Grails possui um comando para geração do war da aplicação. Assim, a base de código como um todo compõe o conjunto de ativos centrais da linha de produtos. Para gerar um produto a partir desses ativos centrais, o trabalho necessário é editar o arquivo de configuração de dependências do Spring e gerar arquivo de aplicação *web* (ou um arquivo jar, no caso de uma aplicação *desktop*). O número de produtos que se pode gerar corresponde ao número de combinações possíveis das variabilidades.

A linguagem que configura como é feita a construção do grafo de objetos (por injeção de dependências) foi empregada neste trabalho com um objetivo específico: a instanciação de aplicações a partir de ativos centrais da linha de produtos, previamente implementados. Neste contexto, essa linguagem pode ser compreendida como um caso particular de AML, por compartilhar de suas características essenciais, isto é, ser específica ao domínio e permitir a abstração do processo de geração do produto.

A configuração da aplicação gerada, por meio da AML, determina o modo como os objetos se interconectam. Porém, mesmo as classes que não são utilizadas (responsáveis por uma variabilidade opcional, por exemplo) são empacotadas no arquivo WAR junto com as outras classes. Essa falta de controle sobre quais classes (ou arquivos JAR contendo várias classes) entram no pacote não chega a ser um problema, pois deixa o usuário final com mais liberdade de alterar o produto adquirido, caso ele tenha conhecimento técnico para tal.

Essa liberdade concedida ao usuário final é importante porque um dos grandes problemas do desenvolvimento de software é entregar aquilo que traz mais valor ao cliente, um objetivo que nem sempre é atingido apesar dos maiores esforços. Além disso, a agilidade na resposta a uma mudança nas necessidades do cliente é bem maior quando a tarefa de configurar um produto fica por conta do próprio dono desse produto.

4.6 Uso do Captor para geração de aplicações

Conforme discutido na seção anterior, o processo de geração de um produto começa com a seleção e configuração de suas variabilidades. Essa, inclusive, é a parte mais importante do processo, uma vez que a fase subseqüente — geração de uma aplicação executável — requer uma intervenção manual mínima. Logo, é bastante desejável que se reduza a probabilidade de erros durante a configuração das variabilidades, já que essa é a fase mais crítica.

Por esse motivo, neste projeto, tanto a escolha das tecnologias quanto as decisões arquiteturais foram feitas no sentido de simplificar a tarefa de configuração. O resultado deste trabalho, embora ainda esteja longe do ideal, já mostra sua utilidade. Com um arquivo de 20 linhas e uma linguagem relativamente simples, é possível configurar as variabilidades existentes. Mesmo para

funcionalidades ainda não implementadas, a estrutura do arquivo de configuração não precisa ser alterada. Basta acrescentar novos elementos aos que já existem. É o caso, por exemplo, das variabilidades opcionais. Para cada nova variabilidade opcional que se queira acrescentar ao sistema, basta incluir um novo elemento à lista, desde que as classes responsáveis por essa variabilidade opcional estejam implementadas e disponíveis para uso. É importante lembrar, também, que para essas funcionalidades opcionais, deve haver um controlador cujo nome siga a convenção explicada na Seção 4.5.

A edição desse arquivo pode ser feita diretamente, usando um editor de texto qualquer. Para tanto, o engenheiro de aplicações terá que conhecer: a) a linguagem específica de domínio (DSL, na sigla em inglês), criada neste projeto, para sistemas de transportes urbanos; b) a linguagem Groovy, que é uma linguagem de propósito geral, utilizada como hospedeira da linguagem específica de domínio. Os pormenores relativos a linguagens específicas de domínio fogem ao escopo deste trabalho. Uma bibliografia resumida sobre o assunto foi feita por van Deursen e Visser (2000).

Neste trabalho, a DSL para sistemas de transporte foi projetada com o objetivo de ser fácil e intuitiva, tanto em seu aspecto sintático quanto semântico. No entanto, apesar de toda a simplicidade que a linguagem oferece, é possível acrescentar mais uma camada de simplificação a esse modelo, tornando possível a edição das configurações por meio de uma interface gráfica específica para o domínio em questão. Essa camada adicional pode ser construída usando-se geradores de aplicações. Neste caso, foi usado o Captor (Shimabukuro *et al.*, 2006; Pereira e Braga, 2007).

O Captor é uma ferramenta que pretende dar apoio tanto à engenharia de domínio (conhecida como engenharia de linhas de produtos, no ESPLEP) quanto à engenharia de aplicações. Com ele o engenheiro de domínio pode definir os pontos de variabilidade do sistema, criando gabaritos (*templates*) para os artefatos variáveis. Tais gabaritos podem ser instanciados, isto é, os pontos de variabilidade podem ser substituídos por definições particulares para um determinado produto da linha. Assim, o engenheiro de aplicações é capaz de criar quantas instâncias forem necessárias a partir dos gabaritos produzidos pelo engenheiro de domínio.

O diagrama da Figura 4.10 mostra os artefatos envolvidos no processo de configuração do produto. Acima da linha horizontal tracejada estão os artefatos gerados pelo Captor. Abaixo, encontra-se o artefato final, resources.groovy, que é o arquivo de configuração de dependências da aplicação, discutido na seção anterior.

Para se gerar artefatos no Captor, primeiro é preciso criar um domínio e, a partir dele, criar projetos para cada aplicação particular. Neste exemplo, está sendo considerada somente uma aplicação: uma instância hipotética do sistema BET para a cidade de São Carlos, de modo a atender as regras de negócio específicas demandadas pelo sistema de transporte urbano daquela cidade. A criação de um domínio no Captor envolve, por sua vez, a criação de um projeto. Sendo assim, há três conjuntos de artefatos:

Projeto BET: projeto do Captor criado a partir da opção “New Captor Project”, na tela de seleção de *wizards* para novos projetos (Figura 4.11(a)).

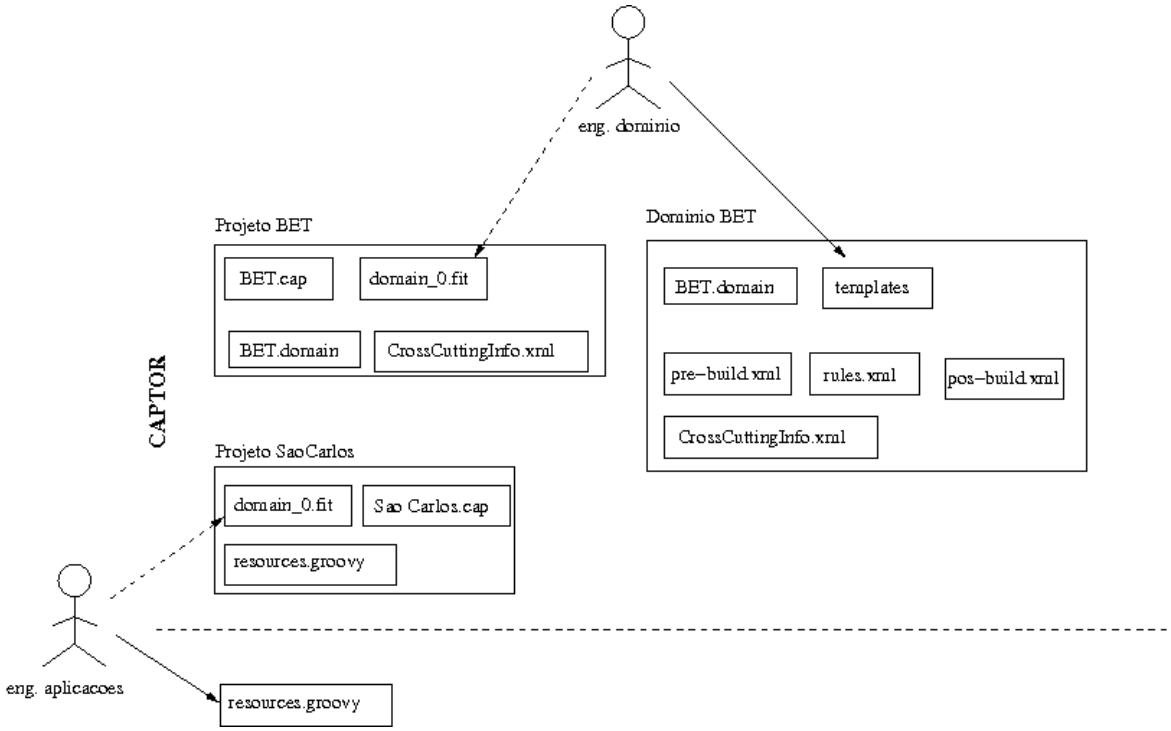
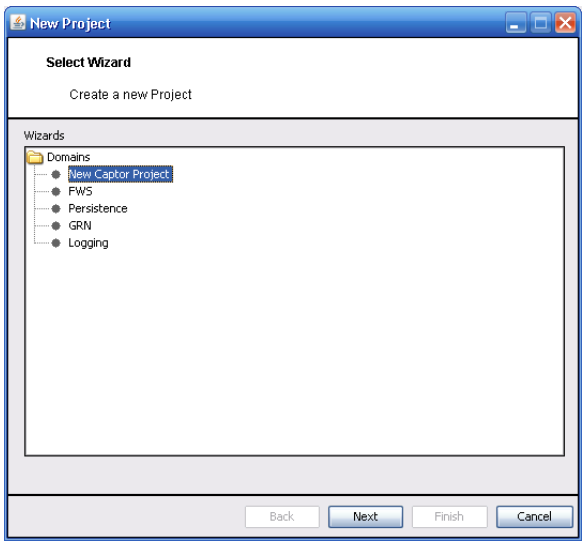


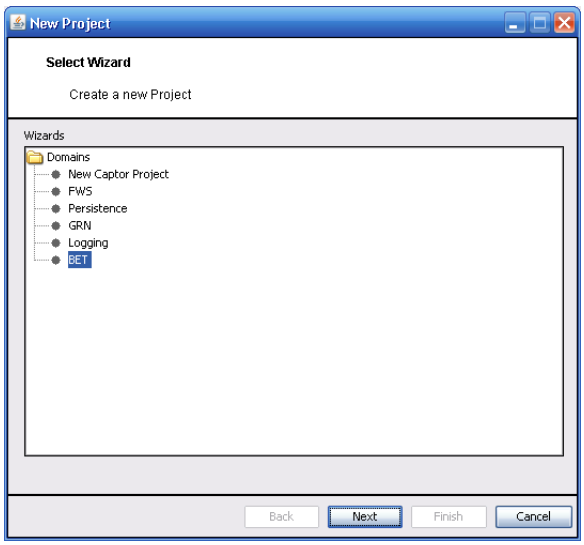
Figura 4.10: Artefatos para configuração de um produto.

Domínio BET: domínio gerado automaticamente a partir do projeto BET.

Projeto SaoCarlos: projeto do Captor criado a partir da opção "BET", na tela de seleção de *wizards* para novos projetos (Figura 4.11(b)).



(a) Projeto de domínio.



(b) Projeto de uma instância do sistema BET

Figura 4.11: *Wizards* para criação de projetos no Captor.

Ao se criar o projeto BET (ou qualquer novo projeto Captor), o Captor gera automaticamente dois arquivos: BET.cap, que contém metadados a respeito do projeto, e CrosscuttingInfo.xml, que

contém informações sobre domínios entrecortantes. Uma vez criado o projeto, é possível definir as informações sobre o domínio, tais como a descrição do projeto, suas funcionalidades variantes e os formulários para configuração dessas funcionalidades. No sistema BET, por exemplo, o tipo de cartão que o sistema cria pode ser configurado (os valores possíveis são `limited` e `regular`). Utilizando-se o Captor é possível criar um formulário contendo uma caixa de seleção com esses dois valores. Durante a instanciação do sistema para criar um novo produto, essa caixa de seleção é utilizada para a escolha do tipo de cartão com o qual aquele produto irá trabalhar.

Quando o projeto é salvo, o Captor gera o arquivo domain_0.fit, com informações sobre os formulários construídos pelo engenheiro de domínio. Neste momento, é possível construir (*build*) o projeto, isto é, gerar o quarto artefato desse conjunto: o arquivo BET.domain, com as mesmas informações, apenas estruturadas de maneira ligeiramente diferente. Pode-se dizer que o arquivo BET.domain contém as variáveis do domínio, como a lista de funcionalidades opcionais, o tipo de cartão etc.

No processo de construção é gerado também o segundo conjunto de artefatos: o domínio BET. Os arquivos BET.domain e CrosscuttingInfo.xml são cópias do projeto BET. Além desses, são criados mais quatro arquivos: três deles (pre-build.xml, rules.xml e pos-build.xml) são basicamente instruções para o próprio Captor sobre como gerar os artefatos finais. O quarto arquivo, indicado na figura genericamente sob o nome "templates" é, de fato, um arquivo XSL, que irá fazer a transformação dos dados específicos de cada projeto, dando origem aos artefatos pretendidos.

O projeto SaoCarlos, o terceiro conjunto de artefatos, é gerado pelo usuário a partir do *wizard* do domínio BET. O arquivo SaoCarlos.cap, similarmente ao BET.cap, é apenas um arquivo de metadados sobre o projeto. O usuário — neste caso, um engenheiro de aplicações — utilizando os *forms* gerados pelo Captor na fase de engenharia de domínio, pode editar os valores específicos para o produto que deseja gerar. Isto significa informar ao gerador, por exemplo, que a lista de funcionalidades opcionais consiste apenas da carga em lote para usuários corporativos e que os cartões que este produto irá tratar terão limite de passagens. Assim como o arquivo BET.domain contém as variáveis do domínio, o arquivo domain_0.fit contém os valores dessas variáveis para cada produto em particular.

Como todo projeto do Captor, é possível iniciar um processo de construção. No caso de projetos de produtos (e não de domínios) os artefatos gerados são aqueles que irão compor os produtos. São a meta final de todo o processo. Conhecendo as variáveis do domínio e seus valores particulares para cada produto, o Captor se encarrega de gerar os artefatos com base no gabarito, que contém informações sobre esse mapeamento.

No caso do sistema BET, há apenas um artefato: o arquivo resources.groovy, que é o arquivo de configurações de dependências do Spring, comentado na seção anterior. As duas instâncias do arquivo resources.groovy (acima e abaixo da linha horizontal tracejada) correspondem efetivamente ao mesmo artefato. Estão desenhadas separadamente na figura para ressaltar a diferença: o de cima é o arquivo gerado pelo Captor. O de baixo é o arquivo editado diretamente pelo engenheiro de aplicações.

Na Figura 4.10 há dois atores interagindo com os artefatos: o engenheiro de domínio e o engenheiro de aplicações. O engenheiro de domínio edita indiretamente o arquivo domain_0.fit, do projeto BET. Este último é gerado pelo Captor, mas é uma representação direta das entradas de dados do engenheiro. Essa "edição" é representada por uma seta tracejada. Além disso, o engenheiro de domínio precisa editar diretamente o arquivo de gabarito, neste caso denominado main.xsl, e apresentado na Listagem 4.10.

Listing 4.10: Arquivo de gabarito do domínio BET.

```
<?xml version="1.0"?>
  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:template match="/formsData/project"/>
  <xsl:template match="/formsData/forms/form">
  import specification.*
  import variability.*
  import passenger.*
  import fare.*

  beans = {
    <xsl:if test="form[@variant='CardCombinationRule']/data/textatt[@name='limit']!=0">
    cardLimit(CardLimit) {
      limit = <xsl:value-of
          select="form[@variant='CardCombinationRule']/data/textatt[@name='limit']"/>
    }
    </xsl:if>

    <xsl:if test="form[@variant='CardCombinationRule']/data/combo[@name='allowDuplicates']='no'">
    noDuplicateCategories(NoDuplicateCategories) {}
    </xsl:if>

    cardCombination(CardCombinationRules){
      selected = [
      <xsl:if test="form[@variant='CardCombinationRule']/data/textatt[@name='limit']!=0">
         cardLimit
      </xsl:if>
      <xsl:if test="form[@variant='CardCombinationRule']/data/combo[@name='allowDuplicates']='no'">
          , noDuplicateCategories
      </xsl:if>
      ]
    }

    cardFactory(CardFactory){
      cardType = "<xsl:value-of
          select="form[@variant='CardConfiguration']/data/combo[@name='cardType']"/>"
    }

    features(OptionalFeatures){
      selected = ['<xsl:value-of
          select="form[@variant='OptionalFeature']/data/textatt[@name='featureName']"/>']
    }
  }
  </xsl:template>
</xsl:stylesheet>
```

O arquivo main.xsl processa os elementos do arquivo domain_0.fit, que representam os elementos dos formulários e seus valores. Os valores preenchidos pelo engenheiro de aplicações

influenciam no arquivo resources.groovy gerado. Isso significa, por exemplo, que a lista das funcionalidades opcionais serão preenchidas com os valores que o engenheiro de aplicações preencheu em um campo de texto criado para este fim. Do mesmo modo, as regras de combinação de cartões que serão escritas no arquivo resources.groovy dependerão das escolhas do engenheiro de aplicações nas caixas de seleção destinadas à configuração dessa variabilidade.

O propósito final do engenheiro de aplicações é configurar as variabilidades dos produtos. De acordo com o que foi exposto na seção anterior, toda a configuração é feita pelo arquivo resources.groovy. Há duas maneiras de se fazer isso: editando diretamente esse arquivo ou "editando" indiretamente o arquivo domain_0.fit, do projeto SaoCarlos, de modo similar ao que o engenheiro de domínio faz no projeto BET. Os dois *screenshots* da Figura 4.12 ilustram como o engenheiro de aplicações pode usar o Captor para configurar as variabilidades de um determinado produto.

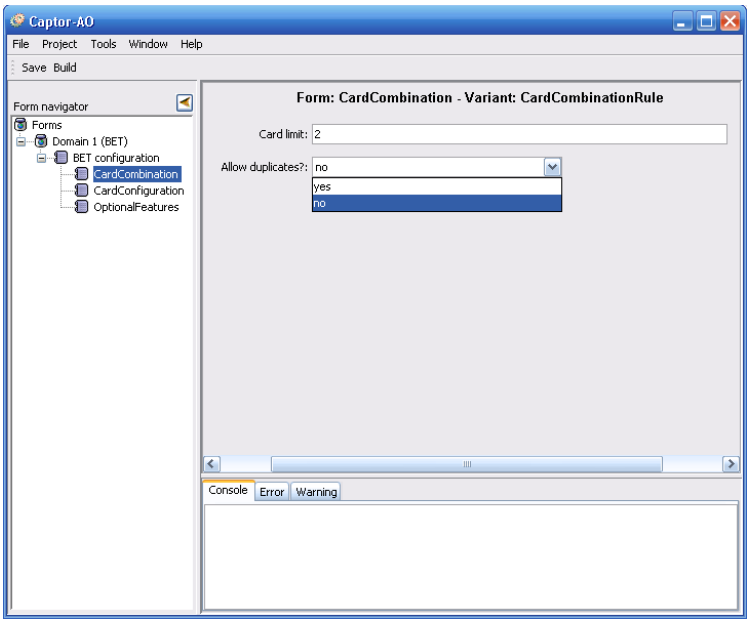
A Figura 4.12(a) mostra o formulário de configuração das regras de combinação (ou restrição) de cartões. O campo de texto rotulado "Card limit" permite ao engenheiro de aplicações definir quantos cartões o passageiro poderá ter. Caso não seja necessário definir um limite, ajusta-se o valor 0 neste campo. A definição a respeito de categorias duplicadas pode ser feita pela seleção de um dos ítens da caixa de combinação rotulada "Allow duplicates?". Na Figura 4.12(b) encontra-se uma caixa de combinação similar, cujo propósito é configurar o tipo de cartão que o produto irá gerenciar (comum ou com limite de passagens).

Em suma, as duas abordagens para geração de produtos são bem diferentes e cada uma apresenta suas vantagens e desvantagens, que estão sintetizadas na tabela 4.2.

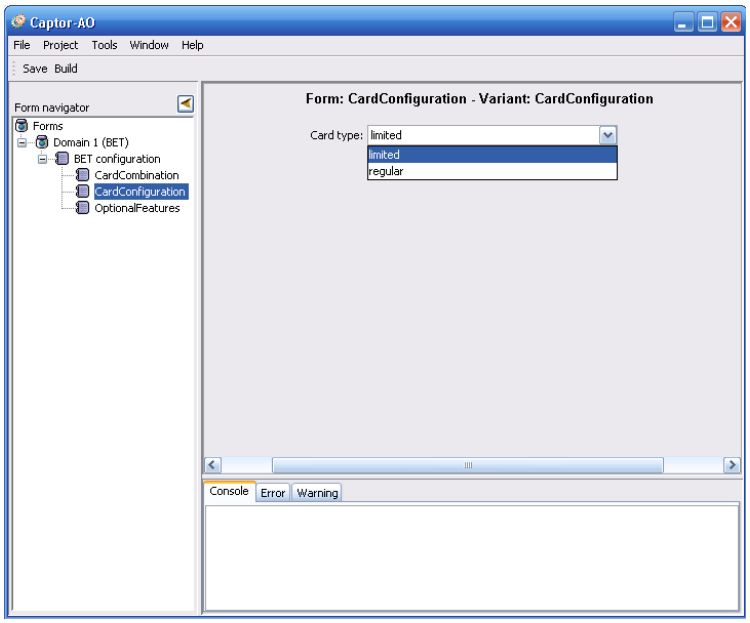
Tabela 4.2: Quadro comparativo das abordagens para engenharia de aplicações

	Com captor	**Sem Captor**
Vantagens	Interface mais amigável para o engenheiro de aplicações.	Um único artefato criado por aplicação.
Desvantagens	Para n projetos, são criados $3n + 8$ artefatos, no total. É necessário um trabalho adicional de engenharia de domínio. A cada alteração no domínio, será necessário um esforço de manutenção dos artefatos. O arquivo de gabaritos é mais ruidoso que o arquivo gerado.	É preciso editar diretamente o arquivo resources.groovy.

Logo, o uso do Captor como ferramenta de geração de aplicações mostrou-se desnecessário neste caso. A linguagem específica de domínio criada, juntamente com o mecanismo de injeção de dependência fornecido pelo Grails é simples o bastante para ser manipulada por um engenheiro de aplicações (considerando que qualquer engenheiro de aplicações possui uma experiência mínima com linguagens de programação).



(a) Definição das combinações de cartões.



(b) Configuração do tipo de cartão.

Figura 4.12: Configuração das variabilidades pelo Captor.

Por outro lado, há casos em que os benefícios trazidos pelo uso do Captor podem superar seus custos. Um desses casos é a geração de arquivos grandes e com sintaxe ruidosa, uma situação bastante comum em arquivos de configuração no padrão XML. Caso fosse necessário algum tipo de manipulação de código na geração das aplicações, o uso do Captor também seria vantajoso, por ser menos suscetível a erros.

4.7 Comparações com a implementação de referência

O estudo de caso apresentado neste trabalho é uma aplicação prática dos fundamentos teóricos de DDD ao desenvolvimento de linhas de produtos de software. A implementação dessa linha de produtos foi contrastada com um outro estudo de caso para o mesmo domínio (Donegan, 2008), com as mesmas regras de negócio, baseado nos métodos tradicionais de desenvolvimento de linhas de produtos, apresentados no Capítulo 2.

A implementação baseada nos métodos tradicionais é citada neste trabalho como "implementação de referência". Nas próximas subseções são analisadas as funcionalidades comuns aos dois projetos.

4.7.1 Aquisição de cartões

As classes desenvolvidas na implementação de referência para a funcionalidade de aquisição de cartões são mostradas no diagrama de classes da Figura 4.13. Nessa arquitetura, baseada nas propostas de autores como Cheesman e Daniels (2001) e Gomaa (2004), todas as classes são encapsuladas em componentes de caixa-preta, que expõem ou requerem interfaces. Esses componentes foram implementados como pacotes Java. Do lado esquerdo são mostrados os componentes do sistema básico e do lado direito, os componentes das variabilidades. Para manter o diagrama simples e focado, somente a variabilidade de número (limite máximo) de cartões é mostrada.

No sistema básico há três componentes:

`GerenciaCtrl`: componente de interface com o usuário, que encapsula os controladores e as visões MVC. Embora esse componente possua vários controladores, são mostrados na figura apenas os controladores relacionados ao caso de uso em discussão.

`CartaoMgr`: encapsula as classes que gerenciam cartões e tipos de passageiros (chamados de categoria na implementação baseada em DDD). Expõe uma interface com vários métodos, dos quais o mais importante para esta discussão é o `buscarTiposPermitidos`. A classe que de fato implementa a interface é `CartaoMgr`, que contém implementação dos métodos de negócio, enquanto as tarefas de persistência são delegadas para os DAOs (Alur *et al.*, 2003).

`PassageiroMgr`: encapsula as classes que gerenciam passageiros. Considerando que no caso de passageiro, apenas um conjunto básico de operações é necessário, o componente possui apenas uma classe: `PassageiroDAO`, que cuida da persistência dos objetos da classe `Passageiro`.

Um ponto importante a ser observado é o tratamento dado aos conceitos de domínio, como `Passageiro`, `Cartao` e `TipoPassageiro`, que não estão representados na figura. Como esse

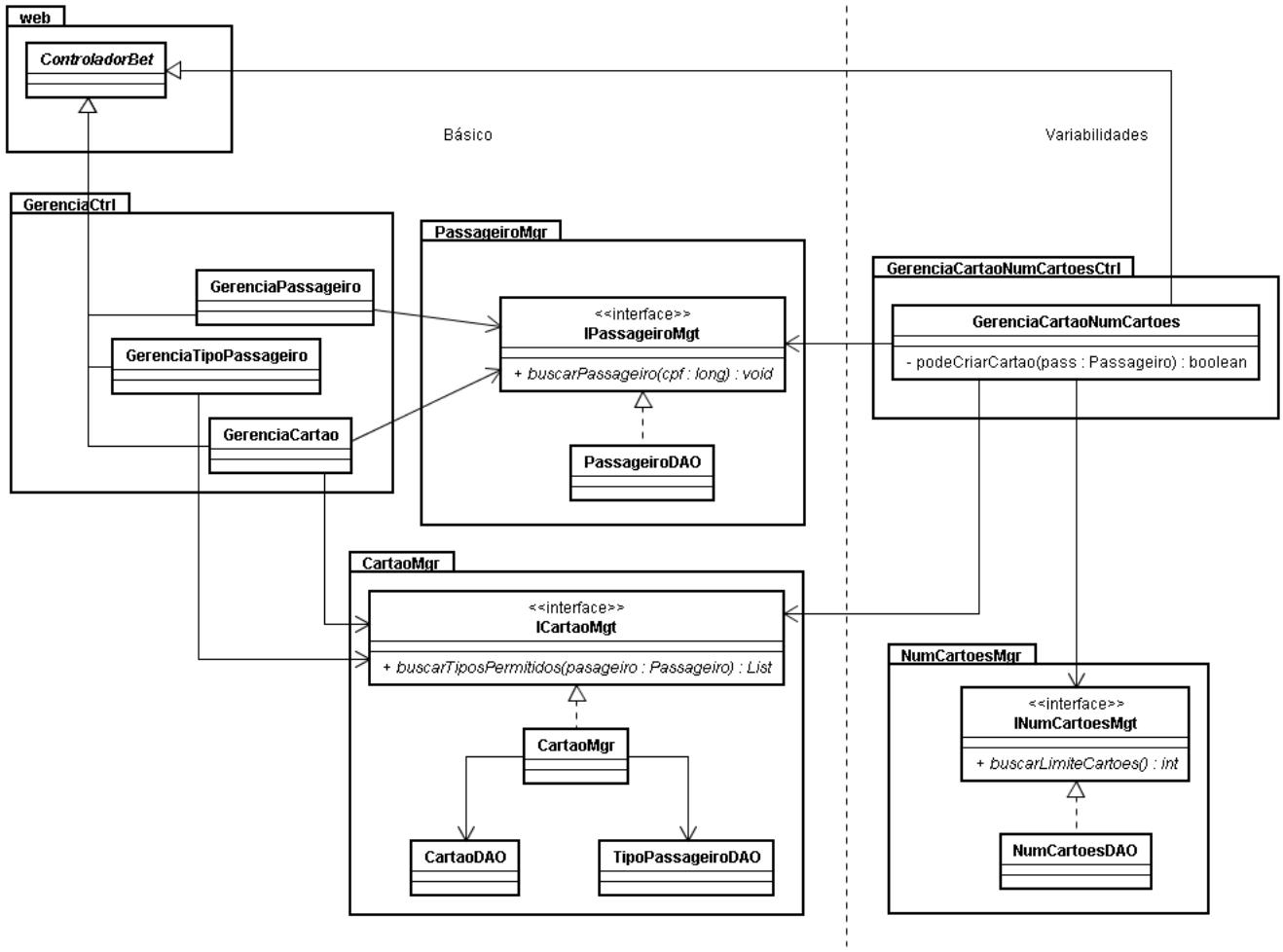


Figura 4.13: Diagrama de classes da implementação de referência para a funcionalidade de aquisição de cartões.

modelo prescreve que todos os objetos estejam encapsulados em componentes, é preciso definir alguma forma de organização para esses conceitos. O mais natural seria colocá-los dentro dos componentes que manipulam suas respectivas regras de negócio. Assim, `Passageiro` ficaria dentro `PassageiroMgr`; `Cartao`, dentro de `CartaoMgr` e assim por diante. Porém, essa forma de organização contraria as próprias regras desse modelo de componentização, já que viola o encapsulamento dos componentes. Em outras palavras, as classes internas a um componente ficariam visíveis a outros componentes.

Para evitar essa violação de encapsulamento, uma possibilidade seria forçar uma comunicação entre os componentes apenas por tipos primitivos, como inteiros, valores booleanos, cadeias de caracteres, etc. O problema dessa solução é que, além de ser muito difícil de implementar, torna o modelo ainda mais anêmico.

Uma terceira alternativa é criar um componente de caixa-branca (com todas as classes visíveis externamente), contendo apenas entidades do domínio. Esse componente deve ser acessado por todos os demais, como uma espécie de repositório de dados. Essa, inclusive, foi a solução adotada na implementação de referência. A vantagem é que causa um impacto pequeno no modelo de

componentização empregado (somente um componente é de caixa-branca), porém separa ainda mais os dados da lógica de negócios. Considerando que a implementação desse componente que contém as classes do domínio está exposta, o acoplamento entre os componentes de negócios e de sistemas é razoavelmente alto, pois qualquer alteração nesse componente de caixa-branca pode afetar os demais.

Dado que os componentes são caixas-pretas, a adição de novas variabilidades implica a inclusão de novos componentes. No caso da variabilidade de limite máximo de cartões, mostrada na figura, foi necessário incluir dois novos componentes: `GerenciaCartaoNumCartoesCtrl`, que contém o novo controlador MVC para esta variabilidade, e `NumCartoesMgr`, que contém classes com lógica adicional para gerenciar o limite de cartões por passageiro.

O novo controlador, `GerenciaCartaoNumCartoes`, possui um método específico para esta variabilidade: `podeCriarCartao`, que indica se o passageiro já atingiu o seu limite de cartões ou ainda pode adquirir um novo. Embora não seja uma boa prática de programação incluir métodos de negócio em controladores MVC (responsáveis por gerenciar a interface com o usuário), não há muitas alternativas usando esse tipo de componentização: mesmo que o código fosse refatorado, esse método seria colocado em um outro componente de controle, responsável por coordenar as tarefas entre o componente `NumCartoesMgr` e os demais já presentes no sistema básico.

Para implementar a nova variabilidade foram necessárias, no mínimo, duas novas classes e uma nova interface. De modo geral, para cada combinação de características dessa variabilidade, será necessário um novo componente de controle. Outra desvantagem dessa abordagem é que as fronteiras do domínio não estão bem delimitadas. Toda a lógica de negócios fica fora das classes que representam os conceitos de domínio e são movidas para os componentes. Os objetos do domínio são meras estruturas de dados, que trafegam pelos componentes.

Na implementação baseada em DDD, ao contrário, a relação é de apenas uma nova classe (que implementa a interface de especificação de cartões) para cada característica da variabilidade de combinação de cartões. Além disso, o escopo dessa classe é bastante estreito, focado apenas na regra do domínio. Todo o resto, incluindo fluxos de navegação do usuário e orquestração da funcionalidade não se alteram, já que tais interesses pertencem às camadas de apresentação e aplicação, respectivamente.

Como a camada de domínio está bem isolada das demais, a implementação de uma nova variabilidade consiste de uma alteração bastante restrita no código. Essa conseqüência é totalmente coerente com os princípios de DDD, que sugere que a modelagem do domínio seja feita diretamente no código. Como a variabilidade está relacionada apenas a regras do domínio, os únicos trechos que devem mudar (ou receber acréscimos de código e funcionalidades) são aqueles relacionados especificamente ao modelo desse domínio.

4.7.2 Crédito de cartões

O tratamento das variabilidades é a principal diferença entre o modelo de componentes usado na implementação de referência e o modelo de domínio baseado nos princípios de *Domain-Driven Design*. A variabilidade de Empresa Usuária (ou Usuário Corporativo), portanto, é o assunto da discussão desta seção.

Na arquitetura de referência, há diversas categorias de componentes. Em especial, na implementação desta variabilidade, são usados componentes de sistema e componentes de negócio. De acordo com Cheesman e Daniels (2000), componentes de sistema são identificados a partir das operações que emergem do diagrama de casos de uso e componentes de negócio são obtidos do refinamento do modelo conceitual. As operações dos componentes de sistema usam operações das interfaces de negócio.

O resultado desse refinamento do modelo conceitual é um conjunto de componentes que encapsulam a lógica de negócio e acesso a uma ou mais entidades do domínio. Por exemplo, o componente de negócio `CartaoMgr` é responsável por quatro classes que representam conceitos de negócio: `Cartao`, `Passageiro`, `TipoPassageiro` e `Pagamento`. Qualquer acesso a objetos dessas classes é intermediado por esse componente. Do mesmo modo, a classe `EmpresaUsuaria` está encapsulada pelo componente `EmpresaUsuariaMgr`.

Esse modo de segmentação dos conceitos do domínio apresenta vantagens, como por exemplo criar grupos de objetos isolados entre si. Assim, a navegação pelo grafo de objetos fica mais restrita e, conseqüentemente, a propagação de mudanças é mais controlada. O princípio é o mesmo que norteia a criação de agregados, em DDD. Em teoria, o uso desses componentes facilita a manutenção do sistema.

O problema surge quando há casos de uso que envolvem classes relacionadas a mais de um componente de negócio. A funcionalidade de crédito de passagens em lote (para usuários corporativos) é uma das várias funcionalidades que se enquadram nessa situação. Para realizar uma carga em lote, é preciso trabalhar com a Empresa Usuária, Passageiros e Cartões, de acordo com o pseudo-código do Algoritmo 1.

Algoritmo 1 Crédito em lote de cartões.

Parâmetros: `empresa`, `valor`, `tipo`

```
1: para todo Passageiro, p, associado à empresa faça
2:   para todo Cartão, c, pertencente a p faça
3:     se tipo = c.tipo então
4:       conceda um crédito de valor unidades monetárias ao cartão c
5:     fim se
6:   fim para
7: fim para
```

O algoritmo 1 não se encaixa em nenhum dos dois componentes citados acima. Em situações como essa, a solução é implementar essa lógica em um componente de sistema, neste caso, o

componente `CargaCartaoEmpresaUsuaria`. O diagrama de classes da Figura 4.14 mostra a organização dos componentes.

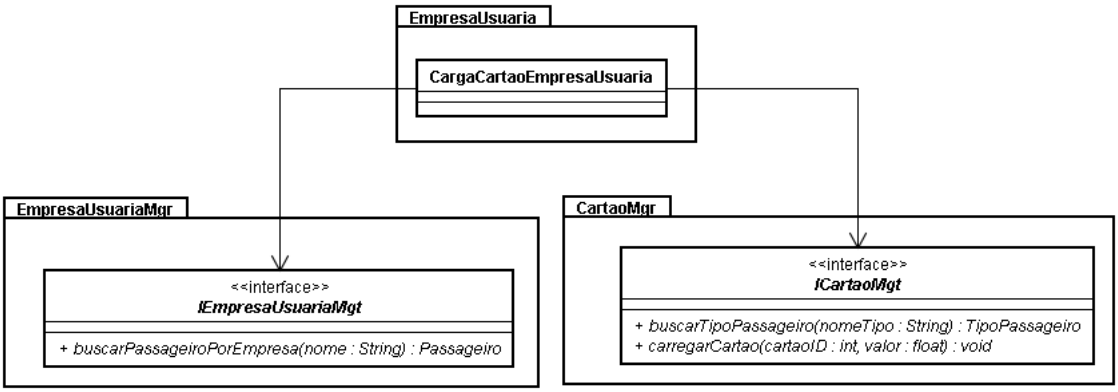


Figura 4.14: Classes e interfaces envolvidas na funcionalidade de crédito em lote da implementação de referência.

A interação entre essas classes para a execução do algoritmo está representada no diagrama de seqüência da Figura 4.15. Observe-se que todo o controle da execução é feito pela classe `CargaCartaoEmpresaUsuaria`. Nas mensagens 1 e 2, esse componente de sistema interage com os componentes de negócio para obter, respectivamente, a lista dos passageiros daquela empresa e o tipo (categoria) correspondente ao nome fornecido. Esses dois componentes de negócio são responsáveis por acessar a camada de persistência para buscar os objetos especificados.

Para cada passageiro fornecido na mensagem 1, `CargaCartaoEmpresaUsuaria` envia as mensagens 3, 4 e 5. A mensagem 5 é enviada somente se o tipo fornecido for igual ao tipo fornecido na mensagem 4.

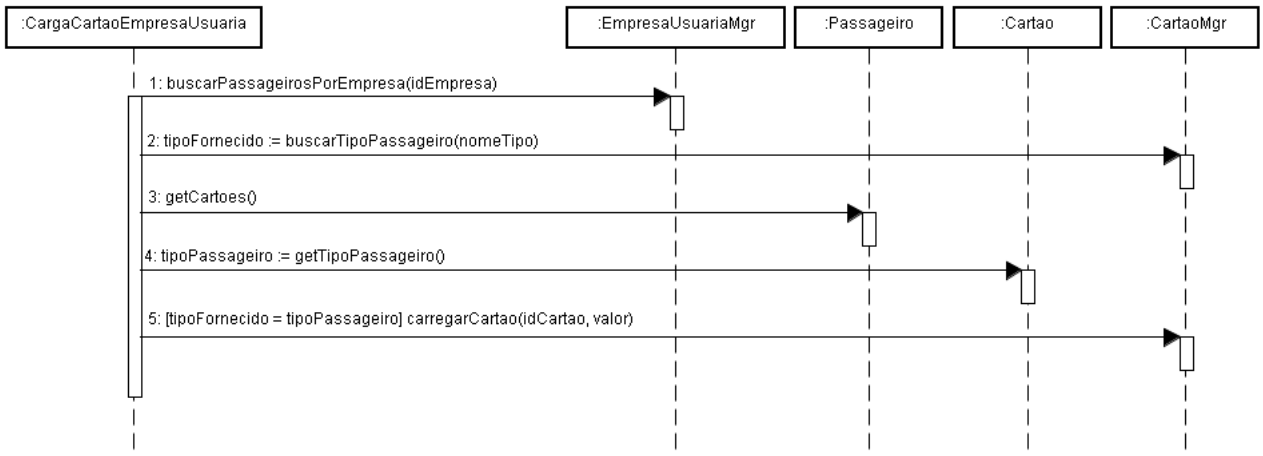


Figura 4.15: Seqüência de crédito em lote, baseada em componentes.

Na implementação baseada em DDD, existe a preocupação de manter o máximo possível de lógica de negócios dentro dos objetos da camada de domínio. Em conseqüência, há uma maior divisão de responsabilidades entre as classes, o que faz com que a execução do algoritmo de carga em lote fique distribuída por vários objetos.

O diagrama da Figura 4.16 mostra a seqüência de operações efetuadas para realizar uma carga em lote para um Usuário Corporativo. Cada objeto cuida de uma parte do processo, delegando as demais tarefas para outro objeto. Na mensagem 1, o controlador simplesmente delega a chamada para o serviço da camada de aplicação, enviando os parâmetros recebidos pela requisição do usuário. O serviço, por sua vez, obtém uma referência ao `CorporateUser` correspondente. Essa operação não está ilustrada na figura porque a referência é obtida através de metaprogramação. No código é feita uma chamada ao método estático `CorporateUser.findByCnpj()`.

De posse do objeto da classe `CorporateUser`, o serviço envia a mensagem 1.1 para esse objeto, indicando que ele deve efetuar crédito para todos os passageiros associados a ele como empregados. Para cada passageiro associado, o objeto `CorporateUser` envia a mensagem 1.1.1, para que esses passageiros creditem o valor fornecido no cartão pertencente a categoria cujo nome também é fornecido. Por fim, quando um objeto da classe Passenger recebe essa mensagem, ele envia a mensagem 1.1.1.1 para o cartão correspondente, delegando a operação de crédito para o Cartão, que efetivamente altera o seu saldo.

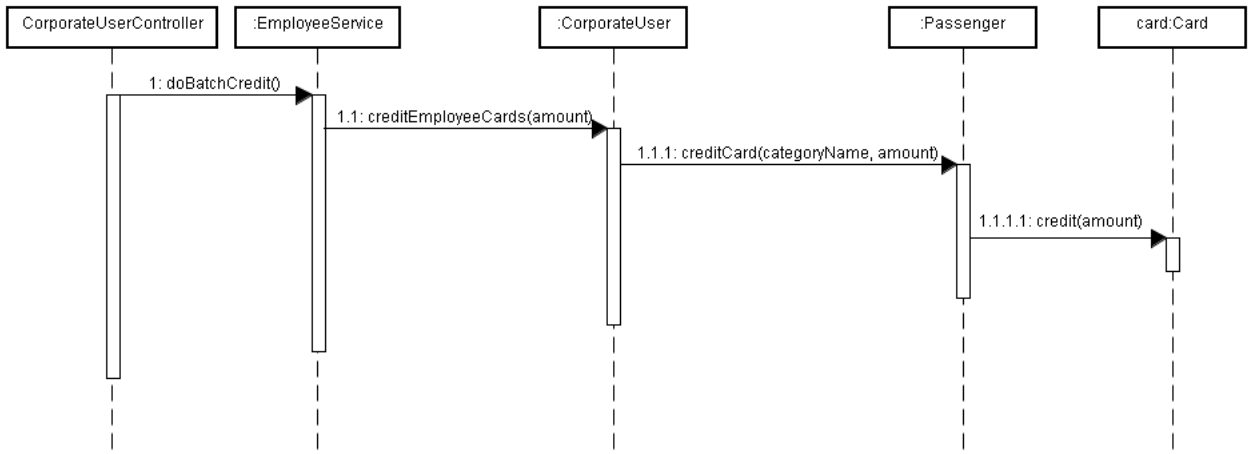


Figura 4.16: Seqüência de crédito em lote, baseada em DDD.

A questão do equilíbrio entre abrangência e profundidade, no que diz respeito ao conhecimento das regras de negócio pelos objetos, apresenta-se de maneira bem nítida no diagrama da Figura 4.16.

Note que o objeto mais à esquerda no diagrama, da classe `CorporateUserController`, inicia um processo de carga em lote através de uma chamada de método (`doBatchCredit`, da classe `EmployeeService`) e termina quando o processo como um todo foi concluído, seja de maneira bem-sucedida ou abortado em caso de falha. Entretanto, todo o conhecimento dessa classe sobre a operação de carga em lote restringe-se a chamada do método que deu origem ao fluxo de execução. A classe `CorporateUserController`, portanto, possui um conhecimento bem abrangente do processo, mas pouco aprofundado. A granularidade da classe é baixa.

No outro extremo está um objeto da classe `Card`, que detém conhecimento sobre o saldo, que é a propriedade que irá ser alterada, de fato. Por outro lado, essa classe não possui nenhum

conhecimento do processo como um todo. Conseqüentemente, a granularidade dessa classe é a mais alta possível. Ao caminhar da esquerda para a direita no diagrama (o caminho do fluxo de controle), nota-se um refinamento progressivo na granularidade das classes em relação à unidade de trabalho que está sendo executada.

O efeito dessa correlação é a maior possibilidade de reúso das classes. Por exemplo, o método `credit()`, da classe `Card`, pode ser reutilizado em outras circunstâncias, sem nenhuma alteração. É o caso da funcionalidade principal: efetuar uma carga de créditos em um cartão individual.

4.7.3 Limite de passagens por período de tempo

A restrição do número de passagens por período de tempo é uma funcionalidade razoavelmente complexa, que envolve várias regras e interações do usuário com o sistema. Para simplificar a comparação entre as implementações[4], decidiu-se abordar apenas um subconjunto dessa funcionalidade: verificação de possibilidade de inserção de crédito. Dado um valor de crédito, o sistema deve informar se aquele valor é permitido ou não para um dado cartão. O diagrama de seqüência da Figura 4.17 mostra as operações realizadas por cada componente na implementação de referência.

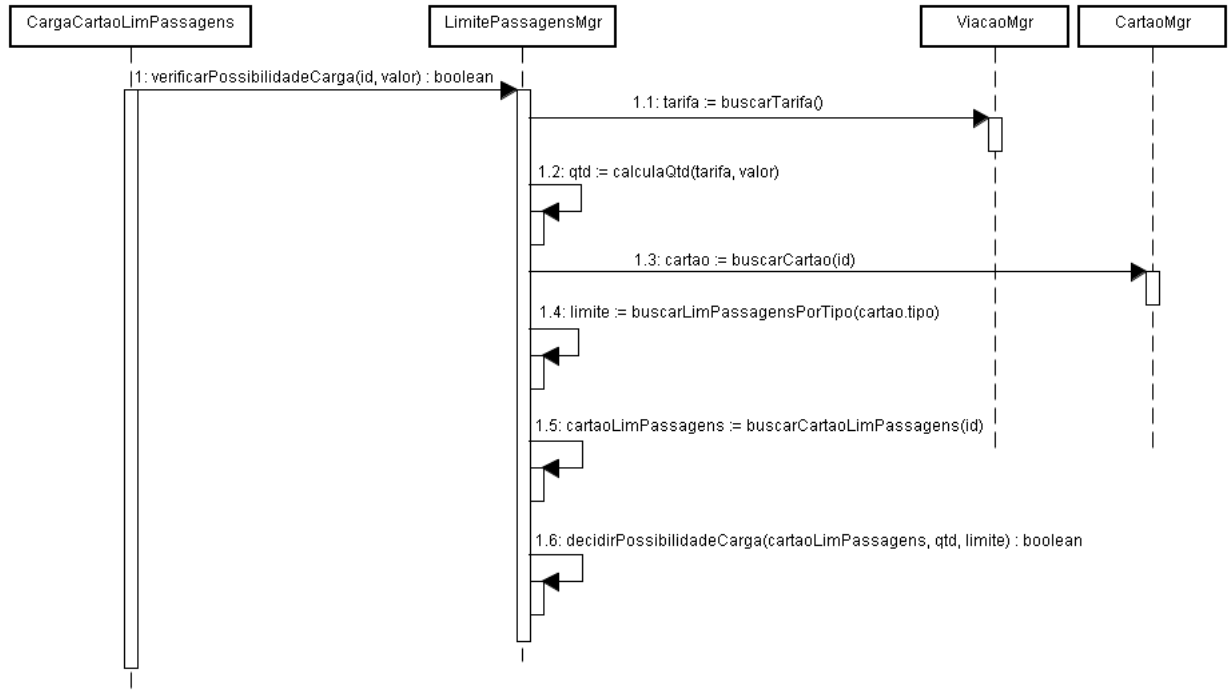


Figura 4.17: Diagrama de seqüência para a verificação da possibilidade de inserção de crédito, na implementação de referência.

Há quatro componentes envolvidos na implementação dessa funcionalidade na arquitetura de referência. `CargaCartaoLimPassagens` é o controlador MVC web responsável por tratar as requisições do cliente. Ao receber os dados mencionados acima (identificador do cartão e valor

[4]Todos os outros fluxos de controle para essa funcionalidade são semelhantes ao escolhido para comparação.

de crédito) ele delega a tarefa para o componente `LimitePassagensMgr`, invocando o método `verificarPossibilidadeCarga`, repassando os parâmetros que recebeu do cliente.

O componente `LimitePassagensMgr` comunica-se com dois outros componentes para completar a tarefa: `ViacaoMgr` e `CartaoMgr`. O primeiro fornece a tarifa vigente e o segundo devolve um cartão a partir do seu identificador. De posse da tarifa, o componente `LimitePassagensMgr` calcula a quantidade de passagens correspondente, ou seja, a quantidade que o passageiro deseja comprar. De posse do cartão, é possível descobrir seu tipo (ou categoria). O identificador do cartão é usado, também, pelo próprio `LimitePassagensMgr` para obter uma entidade chamada `CartaoLimPassagens`, usada para representar um cartão com limite de passagens. Com essas três informações (o cartão com limite de passagens, a quantidade de passagens e o limite), `LimitePassagensMgr` decide se há a possibilidade de carga ou não. O resultado dessa decisão é usado como valor de retorno do método `verificarPossibilidadeCarga`.

Esse padrão arquitetural, de separar componentes de negócio de componentes de sistema, já foi discutido na seção 4.7.2. No exemplo da Figura 4.17, a classe `CargaCartaoLimPassagens` é o controlador MVC, que delega a chamada para o componente de negócio `LimitePassagensMgr`. Este último implementa toda a lógica de negócios, usando os componentes de sistema `ViacaoMgr` e `CartaoMgr` como auxiliares, basicamente para tarefas de persistência das entidades, as quais carregam apenas dados e nenhum comportamento.

O componente `LimitePassagensMgr` é especializado nas tarefas relacionadas a cartões sujeitos a limite de passagens. Isso significa que tanto os detalhes específicos das regras de negócio — como calcular a quantidade de créditos que podem ser inseridos — quanto os detalhes relacionados ao fluxo de controle — como a seqüência em que as operações devem ser efetuadas — são tratadas pelo mesmo componente. Sabe-se, porém, que o fluxo de controle (pelo menos neste caso) independe da regra de passagens para os cartões. Como existe outro componente de negócios responsável pelas funcionalidades de crédito de cartões quando não há limite de passagens, há, necessariamente, uma duplicação de código. Tal duplicação pode até não ser literal (“copiar e colar”), mas pode ser uma outra implementação da mesma funcionalidade em outro ponto do código. Em qualquer destes casos, é um esforço duplicado e desnecessário, além de tornar mais complexa tanto a manutenção quanto o acréscimo de novas funcionalidades.

A mesma funcionalidade é implementada de maneira bem diferente usando-se os princípios de DDD. A seqüência das chamadas de métodos pode ser observada na Figura 4.18. Em primeiro lugar, nesta implementação não existe um processo explícito de consulta de possibilidade de crédito para um cartão com limite de passagens. Em vez disso, o usuário pode consultar quanto de crédito ainda pode ser inserido e, caso se tente inserir um valor maior, o sistema informa ao usuário que a operação não é permitida. No primeiro caso, o processo é bem trivial: dado um identificador de cartão, o controlador `CardController` recupera o objeto correspondente usando o mecanismo de persistência fornecido pelo Grails e mostra seus dados (incluindo a quantidade de crédito “inserível”).

Para efeito de comparação, foi escolhido o segundo caso por ser mais parecido com o da implementação de referência. Nesta situação, o cliente *web* envia uma requisição ao controlador `CardController`, que dispara a chamada ao método `credit`, passando os parâmetros enviados pelo usuário. O controlador, por sua vez, extrai o identificador do cartão e o valor de crédito a ser inserido e os repassa para o serviço de aplicação `CardCreditService`. Este serviço de aplicação consulta o serviço de domínio `FareService` a fim de obter a tarifa vigente. Com o valor a ser inserido e a tarifa vigente, o `CardCreditService` envia uma mensagem ao cartão correspondente ao identificador fornecido pelo usuário. Essa mensagem (1.1.2) dispara uma inserção de crédito no cartão.

Ao inserir o crédito no cartão, duas situações são possíveis:

1. O crédito é válido e a operação de inserção é bem sucedida
2. O crédito é inválido, o objeto `Card` joga uma exceção, que é pega pelo `CardCreditService`, que joga outra exceção para o controlador. Este transforma a exceção em uma mensagem amigável para o usuário, dizendo que aquele valor não pode ser inserido.

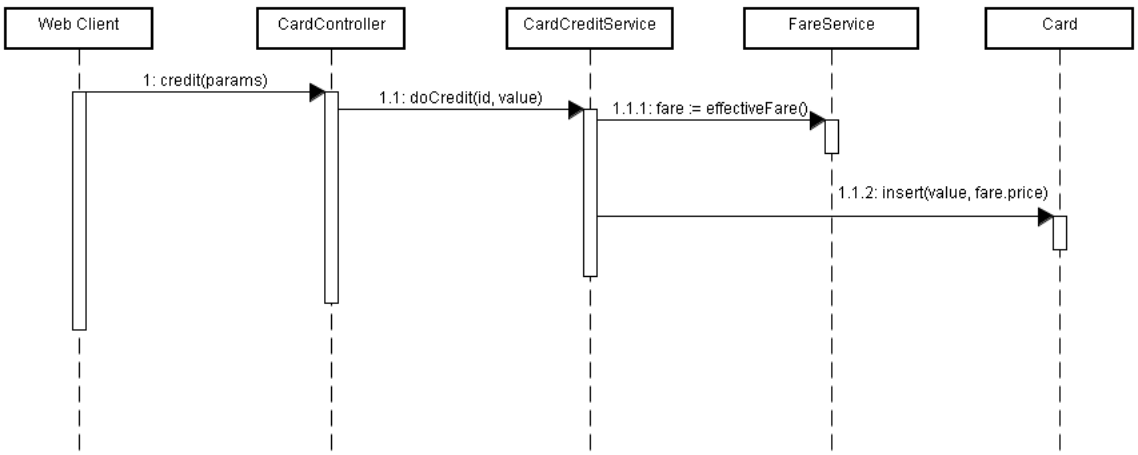


Figura 4.18: Diagrama de seqüência da implementação baseada em DDD para o crédito com limite de passagens.

É importante lembrar que esta funcionalidade trata do limite de passagens, que é uma variabilidade da linha de produtos. Portanto, o objeto `Card` nesse diagrama corresponde efetivamente à classe `LimitedCard`, uma subclasse de Card criada para encapsular os dados e comportamentos de cartões específicos para esta variabilidade. O mais interessante dessa seqüência de operações é que ela é *idêntica* (a menos das exceções) à seqüência de operações para inserção de crédito em um cartão básico, que não tem ciência das regras de limite de passagens. Conforme citado anteriormente neste texto, o polimorfismo da classe `Card` é o que suporta a implementação desta variabilidade.

4.7.4 Vantagens de DDD

Pela comparação entre as duas abordagens de desenvolvimento de linhas de produtos, verifica-se um fato recorrente: com DDD, o impacto que a implementação de uma nova funcionalidade causa à arquitetura é relativamente baixo, comparado ao modelo anêmico da implementação baseada em componentes. Essa robustez da arquitetura pode ser explicada pelos seguintes fatores:

Isolamento da camada de domínio: Um dos pilares de um projeto baseado em DDD é a separação da implementação em camadas. E a mais importante delas (no sentido de que deve merecer mais atenção e trabalho) é a camada de domínio, em que se definem todos (e somente) os conceitos e as regras de negócio. Os detalhes de infra-estrutura devem permanecer em outras camadas, de acordo com o escopo tecnológico (interface com o usuário, persistência, sistemas de notificação, comunicação de baixo nível com outros sistemas, etc.). Quantas e quais camadas devem ser usadas é uma decisão determinada pelas necessidades específicas de cada sistema. O importante é que haja uma camada de domínio devidamente protegida contra interesses alheios ao negócio.

Coesão e acoplamento: A preocupação com um modelo expressivo de domínio é outro fundamento de DDD. Esse modelo pode ser implementado usando-se qualquer paradigma de programação. Em termos de projeto orientado a objetos, essa expressividade pode ser alcançada por um princípio básico: aumento da coesão interna das classes e redução do acoplamento entre elas. No caso específico desta implementação, percebe-se que as classes do domínio são de granularidade fina, ou seja, cada classe é altamente especializada, servindo para representar um único conceito e implementar os comportamentos condizentes a esse conceito. Para produzir um comportamento mais abrangente em termos de regras de negócio, esses objetos são orquestrados por outros de granularidade mais baixa, como os serviços de aplicação.

Princípio da responsabilidade única: Relacionado ao item anterior, o princípio da responsabilidade única prescreve que cada classe deve ter uma, e apenas uma, responsabilidade. Considere-se o caso da classe `Card`. A responsabilidade dessa classe é exclusivamente a de gerenciar o estado desse cartão (com inserção e consumo de créditos, por exemplo). Os outros interesses, como o próprio valor de crédito, estão confinados a suas próprias classes. Do mesmo modo, para implementar a regra de limite de passagens, foi criada uma nova classe, responsável apenas por esse novo comportamento.

Princípio aberto-fechado: Como complemento do princípio anterior, este princípio de orientação a objetos diz que uma classe deve estar aberta para extensões e fechada para modificações. O conceito básico de cartão, por exemplo, não precisa ser modificado para que se introduzam regras adicionais, basta que seja estendido (neste caso, por herança).

Polimorfismo: Herança é uma característica primitiva das linguagens orientadas a objeto que deve ser usada com muita cautela. Isso deve-se ao fato de que ela introduz um grande acoplamento entre a classe base e suas sub-classes. Um dos critérios para se avaliar se o uso de herança está consistente é o princípio da substituição de Liskov, que diz que os subtipos devem respeitar os contratos dos métodos definidos no supertipo. Conforme já comentado, é o que ocorre com as clases `Card` e `LimitedCard`.

Paralela a todos esses fatores, encontra-se a refatoração, essencial para manter a base de código flexível e extensível. A refatoração tem um impacto direto no processo de engenharia de aplicações. Para compreender esta relação é preciso, antes, considerar o processo de geração de um produto da linha. O diagrama da Figura 4.19 mostra os principais elementos envolvidos no processo de geração de produtos em uma linha. Usando-se um gerador, deriva-se um conjunto de produtos a partir de um repositório de código (que contém o que chamaremos aqui, genericamente, de "componentes"). Guiado pela especificação do produto, fornecida por um engenheiro de aplicações, o gerador é capaz de selecionar os componentes corretos e montá-los da maneira correta, de modo a se derivar o produto desejado.

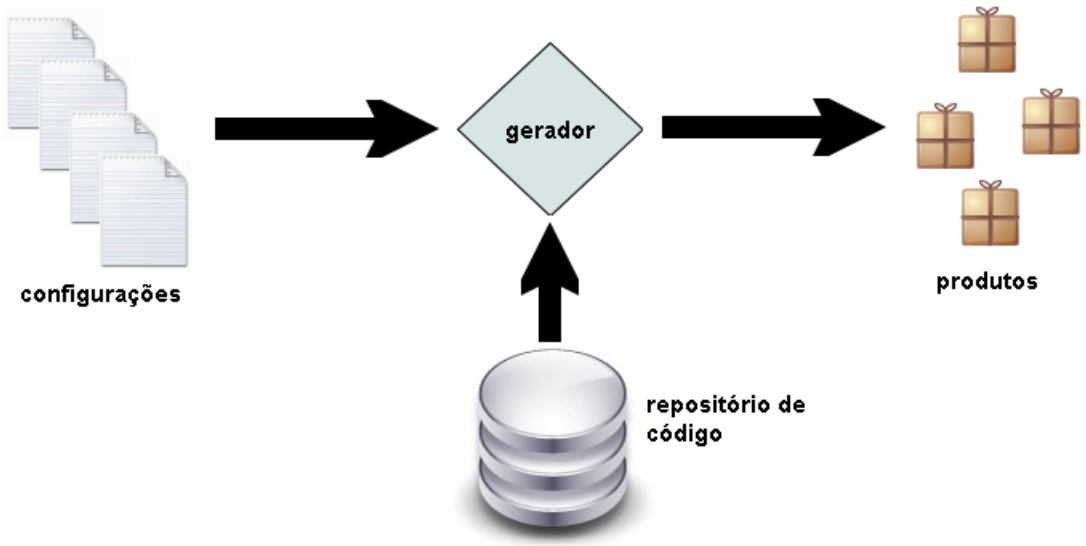


Figura 4.19: Visão geral do processo de geração de produtos em uma linha.

Como o número de combinações entre os componentes de uma linha de produtos é finito, o tamanho do conjunto de produtos deriváveis desse repositório também o será. Assim, na prática, existe a possibilidade de que exista algum cliente com alguma necessidade especial que não esteja contemplada pelos componentes existentes. Nesse caso, há duas soluções possíveis:

1. Implementar um novo componente e acrescentá-lo ao repositório, de modo que seja possível derivar o produto pretendido usando o gerador.

2. Gerar, a partir dos componentes existentes, um produto que seja o mais parecido possível com aquele que se pretende efetivamente entregar, fazendo em seguida, adaptações *ad-hoc* a esse produto gerado.

Por motivos de reúso, robustez e facilidade de manutenção, a solução 1 é mais desejável que a solução 2, a longo prazo. Porém, é preciso considerar que a tarefa de acrescentar um novo componente ao repositório de código pode demandar algum tipo de refatoração na base de código já existente. Isso ocorre porque geralmente esse código não terá sido projetado para suportar todo tipo de alterações imagináveis, o que é, por sinal, uma boa prática. Por outro lado, dentro dos princípios de DDD, uma refatoração é considerada como parte do custo de desenvolvimento de um produto (ou linha de produtos, neste caso) e não um custo extra.

Além disso, DDD prega que as refatorações devem ser feitas ao longo de todo o desenvolvimento. Como se sabe, refatorações frequentes e em pequenos passos ao longo do projeto reduzem o custo de desenvolver uma nova funcionalidade. Logo, em DDD, o custo de desenvolvimento de uma nova funcionalidade é baixo (o próprio exemplo comentado neste texto, é uma evidência disso). A partir dessa constatação, aliada ao fato de que o acréscimo de um novo componente ao repositório é parte dos custos estimados, nos permite concluir que a inclusão de um novo componente (incluindo o custo de refatoração) representa um baixo custo dentro de um projeto baseado em DDD. Essa vantagem se apresenta mesmo a curto prazo.

Se a primeira solução apresentada acima é melhor que a segunda a longo prazo, e a inclusão de um novo componente, utilizando os princípios de DDD, é de baixo custo a curto prazo, deduz-se que o desenvolvimento baseado em DDD é bastante apropriado para linhas de produtos, já que reduz os custos tanto a curto quanto a longo prazo. É importante fazer duas ressalvas: 1) o escopo de aplicação de DDD é bastante amplo, mas o presente estudo foca no desenvolvimento de linhas de produtos. 2) não se pretende, com este trabalho, sustentar a tese de que DDD é o melhor método para se construir uma linha de produtos com qualidade, mas tão somente mostrar o quão adequado ele é para este fim.

4.8 Proposta de abordagem para desenvolvimento de linhas de produtos

Nesta seção é desenvolvida uma proposta para desenvolvimento de linhas de produtos de software com base em métodos ágeis e DDD. Essa proposta é resultado da aplicação de princípios gerais de desenvolvimento (tanto em termos de processo quanto de projeto) e de uma análise posterior à implementação do estudo de caso deste trabalho. Esta última permitiu um exame retrospectivo e distanciado dos eventos que se sucederam durante o trabalho. Com isso, um processo semi-formal emergiu a partir da observação dos pontos positivos e negativos do trabalho. O processo é considerado semi-formal porque há muitos pontos em aberto, cujas decisões competem a cada equipe em particular.

4.8.1 Gerenciamento do projeto

De modo geral, o processo utilizado começa com um entendimento superficial do domínio, seus principais conceitos e regras. Neste ponto, o conhecimento obtido não é suficiente para prever todos os desafios que o projeto apresentará. No entanto, esse contato inicial é de extrema importância. A partir dele, o conhecimento pode ser refinado, de modo a dar origem a um modelo de domínio iterativo.

Tendo esse conhecimento superficial sobre o domínio, os desenvolvedores podem começar a planejar a primeira iteração. É importante que a implementação comece o mais cedo possível, porque a existência de software funcionando (mesmo que incompleto e sem interface amigável, por exemplo) é o que dá chance ao cliente ou especialista no domínio de dar uma resposta e guiar os desenvolvedores no caminho correto. Assim, quaisquer erros ou não-conformidades podem ser detectados tão logo sejam criados. A cada iteração, portanto, as seguintes atividades devem ser desempenhadas:

Elaboração e priorização das histórias: O conhecimento superficial, obtido no início, deve ser refinado ao se escrever as histórias de usuário, contrastando-as de modo a se medir a importância relativa de cada uma. O resultado deve ser armazenado em um *backlog* da linha de produtos e deve estar sempre disponível para consultas e possíveis alterações. Um subconjunto dessas histórias deve ser escolhido para a implementação durante a iteração.

Implementação vertical: Como a ordem em que as funcionalidades são implementadas depende da prioridade de negócios de cada uma, o desenvolvimento da linha é feito de modo vertical, isto é, não existe a imposição de que todas as funcionalidades do núcleo sejam implementadas antes das variabilidades. Isso ocorreu, por exemplo, em relação à aquisição de cartões. Logo após implementada a aquisição, propriamente dita, foram implementadas as regras de restrição de cartões, que, embora sejam características variáveis, estão intimamente relacionadas à aquisição. Essa abordagem é coerente com os princípios ágeis de entregar o que tem mais valor primeiro.

Evolução do modelo de *features* da linha de produtos: Como cada história corresponde biunivocamente a uma *feature*, a identificação destas é direta. Os modelos de *features*, porém, não são apenas reproduções gráficas do backlog. Eles são o primeiro esforço de modelagem dentro de uma iteração. Ao identificar as funcionalidades e o modo como elas se relacionam, ganha-se conhecimento adicional sobre as regras de negócio envolvidas. Este é o momento apropriado para, por exemplo, identificar quais são as variabilidades da linha e a quais tipos elas pertencem.

Implementação das histórias: Embora a evolução do modelo de *features* seja uma forma de modelagem, o principal trabalho neste sentido, de acordo com os princípios de DDD, deve ser feito diretamente no código. É na tarefa de implementação que o modelo de domínio se

refina e todos os detalhes são abordados. O exercício da linguagem ubíqua, neste momento, é muito importante: o código deve refletir a linguagem usada pelos especialistas e ser o mais claro possível. Boas práticas de projeto são também fundamentais, de modo que o resultado seja um código flexível e extensível.

Implementação de testes automatizados: O princípio da construção de um modelo de domínio diretamente no código pode ser aplicado de uma outra forma: a escrita de testes automatizados em vários níveis de abstração, como testes de unidade, testes funcionais, testes de integração e testes de interface de usuário. A escrita de testes automatizados, além de servirem ao propósito de criar uma "rede de proteção" para o código, também ajudam na tarefa de processamento do conhecimento.

Execução de testes manuais: Apesar de todo o esforço que deve ser direcionado para a automação de testes, é preciso lembrar que eles não eliminam a necessidade de testes manuais. Evidentemente, os testes manuais não podem ser apenas uma repetição dos testes automatizados. Eles devem ser de natureza exploratória, a fim de identificar possíveis cenários de teste não cobertos pela automação. Quando um cenário desses é encontrado, ele deve ser incluído na suíte de testes automatizados.

Refatorações: Dado que o modelo é iterativo e feito diretamente no código, há um risco de que o código se degenere rapidamente. Porém, cada nova funcionalidade requisitada é uma oportunidade de rever o modelo, de modo a tentar torná-lo mais genérico. Sempre que for possível, cada novo *insight* sobre o domínio deve vir acompanhado de uma refatoração. O caminho contrário também deve ser explorado, isto é, seqüências de pequenas refatorações para tornar o código mais limpo podem levar a um conhecimento mais profundo sobre o domínio ou a grandes *insights*. Portanto, as refatorações devem ser feitas continuamente, tanto antes quanto após desenvolvimento de código novo.

Lançamento de novas versões: Muitas funcionalidades somente fazem sentido para o usuário final quando disponíveis em conjunto. Por isso, é importante fazer um planejamento de lançamento. A cada iteração, a linha de produtos evolui, ganhando novas funcionalidades. De tempos em tempos, esse acúmulo de funcionalidades atinge um marco no projeto, tornando a linha madura o suficiente para ser disponibilizada no mercado ou para os clientes que possam se beneficiar dela. Durante um projeto, vários lançamentos podem ser feitos, mas em geral eles são menos freqüentes que as iterações.

O diagrama de atividades da Figura 4.20 mostra as atividades propostas para o processo de desenvolvimento de linhas de produtos com base em DDD e métodos ágeis. A atividade representada pela caixa "Iteração N", no canto superior direito da figura, representa a n-ésima iteração do projeto. Associado a esta atividade, há um comentário listando as tarefas realizadas em cada iteração. Embora elas estejam listadas em uma determinada ordem, é importante lembrar que tais

tarefas não ocorrem seqüencialmente. Ao longo da iteração, todas as tarefas são executadas na ordem que for mais conveniente, dadas as circunstâncias específicas do trabalho.

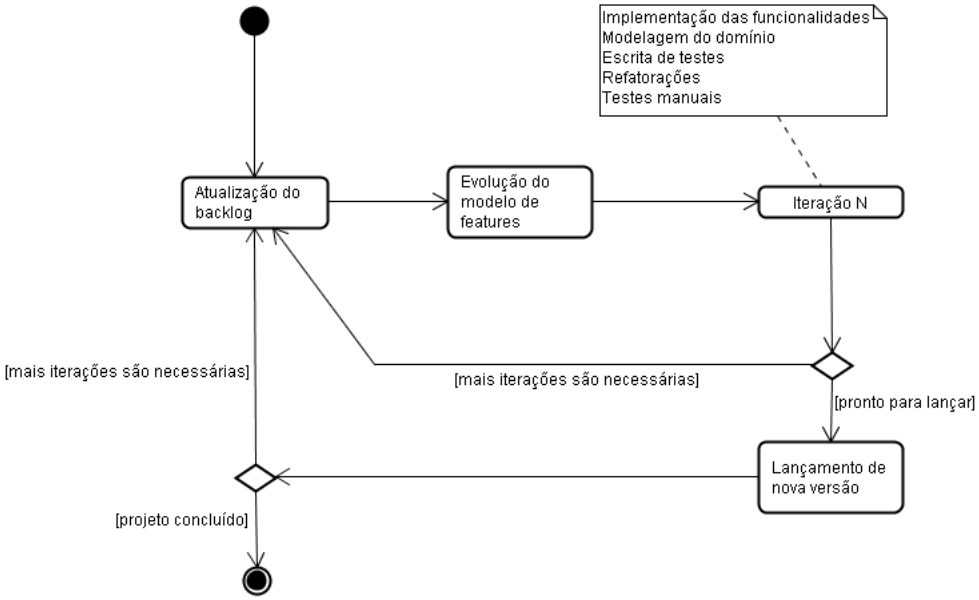


Figura 4.20: Atividades de desenvolvimento de linhas de produtos.

As iterações no desenvolvimento de linhas de produtos, segundo proposto neste método, são de grande importância. Elas permitem maior garantia de atendimento dos requisitos, já que o fim de cada iteração é um ponto de checagem do projeto. É o momento certo para que os *stakeholders* do projeto conheçam o que já está pronto e possam contestar os resultados, caso não as funcionalidades implementadas não atendam às suas necessidades. É também o momento apropriado para opiniões e sugestões de melhoria, quando for o caso.

A combinação de desenvolvimento iterativo com implementação vertical da linha também ajuda no atendimento dos requisitos propostos e na qualidade do produto final. Ao desenvolver uma funcionalidade e suas variabilidades em uma mesma iteração, a equipe de desenvolvimento tem a capacidade de antecipar problemas que, de outro modo, somente apareceriam ao final do projeto, tornando os custos de reparo bem maiores. Este é um dos pontos em que há grande divergência entre o método proposto e os métodos tradicionais. Nestes últimos, a implementação das variabilidades é abordada somente após a conclusão do núcleo da linha.

Outro ponto em que o processo ágil proposto neste trabalho difere grandemente dos métodos tradicionais é a importância dada a documentos textuais de requisitos. Conforme se pode observar, além das histórias de usuários, não se prescreve a redação de nenhum outro documento dessa espécie durante o desenvolvimento. A decisão de se escrever um documento de requisitos ou qualquer exposição escrita sobre as regras do domínio cabe exclusivamente aos desenvolvedores. Cada projeto deve ter a liberdade de criar esse tipo de documentação de alto-nível de maneira que se ajuste às suas necessidades e objetivos.

Em qualquer caso, o exercício da linguagem ubíqua não deve ser esquecido. Seja em conversas informais com os especialistas, seja em documentos de texto ou qualquer outra forma de comunicação em que o conteúdo envolva o conhecimento sobre o domínio, é preciso prestar atenção às expressões utilizadas no sentido de evitar ambigüidades e interpretações equivocadas.

4.8.2 Práticas de engenharia de software

Além das prescrições e sugestões relacionadas ao gerenciamento do projeto, é preciso prestar atenção às práticas de engenharia de software empregadas. Essas duas áreas de conhecimento devem se integrar, de modo que uma complemente a outra. As práticas empregadas no estudo de caso são examinadas em detalhes nesta seção. O intuito desse exame é estabelecer — ainda que de modo incipiente — um corpo de conhecimento que possa ser reutilizado em novos projetos de linhas de produtos de software.

A separação em camadas, segundo o que já foi analisado anteriormente, é um dos primeiros passos na implementação da linha de produtos. A idéia geral é que se trabalhe com quatro camadas: apresentação, aplicação, domínio e infra-estrutura. Porém, as condições particulares de cada projeto podem variar, como a tecnologia utilizada, a complexidade da interface de usuário, o número e as características das variabilidades, etc. Todas essas particularidades podem influenciar na tomada de decisão quanto ao número efetivo de camadas que serão adotadas.

No estudo de caso apresentado neste trabalho, por exemplo, o uso do Grails como *framework* de desenvolvimento tornou desnecessário o uso da camada de infra-estrutura, por conta da sua abordagem simplificada em relação à persistência. Ainda assim, caso houvesse a demanda por funcionalidades que exigissem o uso de outros serviços de infra-estrutura, como envio de *emails* ou acesso a sistemas de cobrança externos, o uso dessa quarta camada seria inevitável.

Sugere-se, ainda, que a implementação comece a ser feita a partir da camada de domínio, dado que esse é o ponto focal do desenvolvimento. Isso ajudará a manter a concentração dos desenvolvedores nas regras de negócios e na modelagem do domínio. Por esse motivo, a escrita e execução de testes automatizados (em vários níveis) é de suma importância. Eles permitem avaliar o estágio do desenvolvimento da linha de produtos, por permitirem uma medição objetiva de quantas funcionalidades estão implementadas e funcionando corretamente. Quanto maior for a cobertura dos testes, melhor será a acuidade dessa medida de progresso. Por conseguinte, sugere-se o uso de ferramentas de medição de cobertura de testes (Yang *et al.*, 2007).

Para facilitar a escrita de testes automatizados, é importante que a cada história de usuário esteja associada uma lista de testes de aceitação informais, de acordo com a sugestão da diretriz 1, da Seção 4.4.1. Evidentemente, os testes automatizados devem possuir determinadas qualidades que os testes informais das histórias não possuem, como rigor na especificação dos comportamentos, declaração do contexto (*fixture*) do teste, detalhes dos resultados esperados, entre outros. A lista de testes informais serve, contudo, como ponto de partida e é uma boa fonte de referência sobre o que precisa ser feito em termos de testes. A mesma idéia vale para os testes manuais.

Um efeito colateral de uma boa cobertura de testes é que a implementação das demais camadas pode ser postergada o máximo possível. O comportamento principal da aplicação pode ser verificado pelos testes de unidade e funcionais. Nos pontos em que haja interação com objetos de outras camadas, estes podem ser substituídos por objetos dublês, conhecidos como *mock objects* (Freeman *et al.*, 2004).

O próximo passo é identificar quais classes pertencem a quais camadas. Partindo da premissa de que cada classe possui apenas uma responsabilidade, o principal fator que deve ser levado em conta nessa identificação é o nível de granularidade dessas responsabilidades. Como se pode perceber pelo exemplo da linha de produtos desenvolvida neste trabalho, as responsabilidades das classes da camada de apresentação são de granularidade grossa ao passo que as classes do domínio são de granularidade fina.

Considere-se, por exemplo, a concessão de crédito em lote para os funcionários associados a um determinado usuário corporativo. Ao utilizar essa funcionalidade, o usuário dispara um processamento razoavelmente complexo, que envolve a troca de mensagens entre vários objetos, começando por um dos controladores. Os controladores são responsáveis por receber os dados submetidos pelo usuário e redirecionar o fluxo de navegação da maneira correta. Dependendo do fluxo, diferentes telas e mensagens serão mostradas ao usuário, tarefa essa que compete às visões. Os controladores e as visões, portanto, possuem conhecimento sobre toda a funcionalidade de crédito em lote, que é vista pelo usuário como uma unidade de trabalho. Por outro lado, essas classes não contêm nenhuma lógica de negócios. Ou seja, o conhecimento dessas classes sobre o domínio é bastante abrangente e pouco aprofundado.

À medida que se vai descendo na pilha de camadas da aplicação, o que se observa é uma inversão dessa relação entre abrangência e profundidade em relação ao conhecimento do domínio. Considerando a mesma funcionalidade de crédito em lote, pode-se constatar que a classe `Card` encontra-se no extremo oposto; os objetos dessa classe são os que efetivamente alteram o saldo, o que significa que eles possuem um conhecimento bastante especializado e aprofundado sobre o domínio, ao passo que não têm ciência do processo de crédito em lote ou qualquer outro processo relevante ao usuário. Isso os torna altamente reusáveis. A relação entre abrangência e profundidade é mostrada esquematicamente no gráfico da Figura 4.21.

Outro ponto que deve receber atenção especial é o encapsulamento das variabilidades. Para que o projeto seja flexível, é preciso que as funcionalidades variáveis estejam completamente desacopladas entre si e também desacopladas das funcionalidades do núcleo. O desacoplamento das funcionalidades em conjunto com o uso de injeção de dependência em todas as classes facilita muito a tarefa de configuração dos produtos que se deseja instanciar, durante a fase de engenharia de aplicações, além de aumentar o reúso das classes do núcleo. A diretriz 7 trata desse tipo de problema, sugerindo o encapsulamento da variabilidade opcional, isolando-a do núcleo.

Um exemplo de aplicação desse princípio pode ser encontrado na implementação da funcionalidade de crédito em lote. Como essa é uma funcionalidade opcional, todas as suas classes foram encapsuladas em um pacote (`corporation`), de tal modo que nenhuma classe externa a

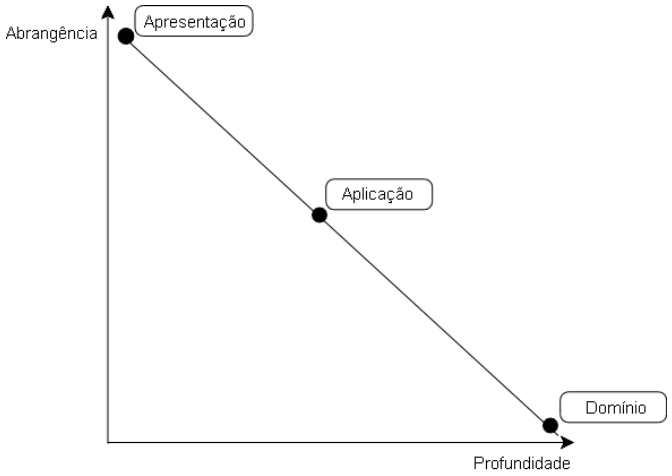


Figura 4.21: Abrangência versus profundidade no conhecimento sobre as regras do domínio.

esse pacote possui referência a alguma classe interna. O resultado é a total independência das demais classes em relação ao pacote `corporation`, que pode não estar presente em virtude de sua natureza opcional. Porém, as demais classes não sofrem qualquer impacto por conta dessa variabilidade.

Observe-se que a funcionalidade de crédito em lote envolve todas as camadas, já que ela proporciona novas unidades de trabalho para o usuário, como associar (e desassociar) um passageiro a um usuário corporativo, além da execução da transação de crédito, que é a principal. Conseqüentemente novos fluxos de navegação precisam ser gerenciados e novas regras de negócio, tratadas. Para simplificar a comunicação entre a camada de domínio e de aplicação, um serviço é interposto entre as duas.

As regras de restrição de cartões, ao contrário, envolvem exclusivamente regras de negócio. Nenhum outro interesse é afetado pelas regras de restrição escolhidas. Por esse motivo, é desejável que a variabilidade esteja encapsulada somente nas classes de domínio, como sugere a diretriz 4. A implementação da funcionalidade de limite de passagens por período de tempo segue igualmente essa diretriz. Esses dois exemplos podem ser generalizados e o princípio pode ser aplicado a todas as demais variabilidades restritas ao domínio. Essa decisão de projeto favorece novamente o reúso e, conseqüentemente, minimiza o trabalho de codificação para variabilidades desse tipo.

Na Tabela 4.3 é feita uma comparação entre os métodos abordados neste trabalho, levando em consideração os principais fatores relacionados ao desenvolvimento de aplicações e reúso de partes de software. Entretanto, alguns fatores, como a estratégia de implementação da linha de produtos, não se aplica a todos os métodos. Em outros casos, os autores de alguns métodos não chegam a discutir alguns fatores, como a abordagem em relação aos testes.

Tabela 4.3: Comparação de métodos de desenvolvimento e reúso

	ESPLEP/PLUS	**FAST**	**Componentes UML**	**DDD/métodos ágeis**
Representação dos requisitos	Casos de uso e modelos de *features*	Modelo econômico e documento de similaridades	Casos de uso e visão do sistema	Histórias de usuário e modelos de *features*
Modelo de domínio	Diagramas de classe, colaboração e statecharts	Vários artefatos (figura 2.5)	Modelo conceitual e de tipos de negócio	Diretamente no código
Evolução das aplicações	Planejada desde o início	Planejada desde o início	Planejada desde o início	Incremental
Implementação da LPS	Horizontal	Vertical	—	Vertical e iterativa
Estratégia de reúso	Componentes de granularidade grossa	Geração de código a partir de *templates*	Componentes de granularidade grossa	Reúso de classes e interfaces (granularidade fina)
Uso de AML	Não	Sim	Não	Sim
Testes	Durante a fase de Transição	—	—	Escrita de testes ao longo do desenvolvimento
Organização da lógica de domínio	*Script* de transação	—	*Script* de transação	Modelo de domínio
Divisão arquitetural	Camadas, cliente/servidor	—	Camadas	Camadas, cliente/servidor

4.9 Considerações Finais

Neste trabalho foi feito um estudo de caso envolvendo a implementação de uma linha de produtos de software para sistemas de transporte urbano. O projeto dessa linha de produtos foi comparado ao de uma implementação de referência para o mesmo domínio e os resultados mostraram-se bastante satisfatórios, especialmente em termos de flexibilidade e extensibilidade da arquitetura. Vantagens secundárias, como menor quantidade de classes e menos código voltado à infra-estrutura também foram observadas. Porém, essas vantagens decorrem, sobretudo, da escolha da tecnologia e pouco se relacionam com decisões de projeto (*design*), propriamente.

O processo empregado neste trabalho também teve uma influência forte sobre a linha de produtos desenvolvida. A partir desta experiência foi possível delinear um processo inicial para desenvolvimento de linhas de produtos com base em métodos ágeis e *Domain-Driven Design*. É importante ressaltar que este processo foi elaborado a partir de uma compilação de princípios de projeto e boas práticas de desenvolvimento, observando-se as lições aprendidas durante a condução de um estudo de caso em particular. Outros estudos de caso poderiam originar novos *insights* e modificações no processo proposto.

Os principais aspectos que caracterizam o processo utilizado neste projeto são a escrita de testes de unidade, desenvolvimento iterativo, refatorações freqüentes, modelagem de domínio no código e separação de interesses em camadas. Em um nível mais baixo de abstração, relacionado especificamente a questões de implementação da linha de produtos, encontram-se as diretrizes de desenvolvimento, citadas ao longo da discussão sobre as iterações do projeto.

Essas diretrizes são regras gerais abstraídas dos problemas concretos encontrados ao longo do desenvolvimento. Mesmo sendo regras generalizadas a partir da experiência, essas diretrizes são apenas guias para o desenvolvimento de novas linhas de produtos, considerando que o objetivo seja alcançar as boas práticas prescritas pelo processo. As diretrizes têm um caráter essencialmente prático, o que torna fácil a sua aplicabilidade em situações particulares.

Deste modo, as diretrizes estão subordinadas ao processo e não devem ser tomadas como regras rígidas ou absolutas, porquanto as condições particulares de cada projeto variam bastante e devem ser levadas em conta. Mais importante que a observância estrita às diretrizes é a aplicação destas aliada ao bom-senso.

O código-fonte da aplicação implementada neste estudo de caso é livre (licença GPL) e está hospedado no endereço `http://sourceforge.net/projects/bet`. O código está sob controle de versão (subversion) e disponível para *checkout*.

CAPÍTULO

5

Conclusões

5.1 Resumo do Trabalho

O trabalho apresentado nesta dissertação tem como interesse principal o desenvolvimento de linhas de produtos de software baseado em *Domain-Driven Design* e métodos ágeis. Para coletar lições sobre esse tipo específico de desenvolvimento foi realizado um estudo de caso para o domínio de sistemas de transportes urbanos. A linha de produtos foi desenvolvida verticalmente em quatro iterações, aplicando-se em cada iteração os vários princípios de DDD e diversas práticas de métodos ágeis. Ao fim de cada iteração, um incremento de funcionalidades (tanto comuns quanto variáveis) foi acrescentado à linha de produtos, tornando-a potencialmente entregável. Para os principais problemas de projeto, as soluções encontradas são abstraídas e enunciadas na forma de diretrizes de desenvolvimento.

A linha de produtos foi projetada de modo a simplificar ao máximo a geração de aplicações, tomando vantagem do paradigma de orientação a objetos e da técnica conhecida como injeção de dependência. Para criar uma nova aplicação, basta configurar o modo como os objetos se inter-relacionam, utilizando, para isso, uma linguagem específica de domínio. Uma linguagem específica de domínio pode ser usada em conjunto com geradores de aplicação, como o Captor, provendo uma camada adicional de abstração.

Uma implementação de referência para o mesmo domínio, baseada nos métodos tradicionais de desenvolvimento de linhas de produtos de software foi usada para efeitos de comparação. As soluções de modelagem encontradas nas duas implementações foram analisadas com respeito à expressividade da arquitetura, padrões de reúso e separação de interesses.

5.2 Contribuições

A principal contribuição desta dissertação é a proposta de um conjunto de diretrizes para o desenvolvimento de linhas de produtos usando DDD e métodos ágeis. A construção desse conjunto de diretrizes foi feita tanto a partir da adaptação de princípios ágeis e de DDD para o contexto de LPS, quanto por meio da abstração dos problemas particulares encontrados na implementação da linha de produtos, que foi objeto do estudo de caso empreendido neste trabalho.

O estudo de caso sugere que é possível construir linhas de produtos fundamentadas em um modelo de domínio mais expressivo, comparado ao que propõem os métodos tradicionais. A expressividade do modelo tem um grande impacto na facilidade de manutenção de uma linha de produtos e, conseqüentemente, na sua evolução. Com isso, a adaptabilidade a mudanças ao longo do tempo é bem maior. Além disso, um modelo de domínio expresso diretamente no código da aplicação reduz a necessidade de documentação não executável, como diagramas UML e documentos de casos de uso.

Durante o desenvolvimento do projeto, houve uma preocupação em aumentar a coesão das classes e reduzir o acoplamento entre elas. Para tal, foram utilizadas várias técnicas e princípios, como polimorfismo e princípio da responsabilidade única. O *design* resultante mostrou-se bastante flexível e extensível. A inclusão ou alteração de regras de negócio ou mesmo de fluxos inteiros de atividade (como a funcionalidade de crédito em lote) podem ser feitas facilmente. A configuração das funcionalidades requeridas para um produto consiste apenas de configuração do grafo de dependências, por meio de uma linguagem específica de domínio.

Do ponto de vista de linhas de produtos, essa flexibilidade resulta em baixo custo na geração de novos produtos. Dado que a refatoração é feita com bastante freqüência, com vistas à manutenção da qualidade do projeto, a inclusão de uma nova funcionalidade, ainda que implique alterações de arquitetura, não é considerada custosa, e sim parte do processo utilizado. Os benefícios, neste caso, superam os custos a longo prazo.

A aplicação de DDD ao desenvolvimento de linhas de produtos de software mostrou-se bastante adequada. A razão disto é o enfoque do desenvolvimento baseado em uma linguagem ubíqua do domínio. Destarte, os conceitos do domínio revelam-se com muito mais clareza no código da LPS. O efeito mais relevante dessa transparência é uma maior facilidade de manutenção da linha a curto e longo prazo.

As principais vantagens observadas na abordagem proposta em relação aos métodos tradicionais são: menor esforço na implementação de uma nova regra de negócios (em geral, uma nova classe, comparada a vários componentes e classes no modelo de componentes da implementação de referência) e maior capacidade de reúso, pela atribuição adequada de responsabilidades, pela divisão em vários níveis de granularidade (compatíveis com o nível de abstração da classe) e pela aplicação de princípios de orientação a objetos, como o princípio de substituição de Liskov e o princípio aberto-fechado. Quanto ao processo, o desenvolvimento iterativo e as refatorações

freqüentes contribuem para um projeto emergente, de modo que o modelo de domínio mantém-se sempre atualizado.

5.3 Limitações e Trabalhos Futuros

Este estudo de caso foi conduzido sobre apenas uma linha de produtos. É preciso que se façam outros estudos similares aplicando DDD e métodos ágeis ao desenvolvimento de linhas de produtos, para que as diretrizes apresentadas neste trabalho possam ser generalizadas e se convertam em um método rigoroso. É importante, também, analisar aplicações para domínios de caráter bem diverso das aplicações corporativas, como aplicações científicas, jogos, telefonia e processamento de texto.

Outro ponto que merece atenção é o impacto da linguagem utilizada na flexibilidade e extensibilidade da linha de produtos. Neste trabalho, foi utilizada uma linguagem de tipagem dinâmica (Groovy), que oferece diversos recursos que uma linguagem estática, como Java, não oferece. Muitos desses recursos, como os fechamentos e a metaprogramação ajudaram a simplificar o projeto, mas é preciso investigar se há uma relação intrínseca ou apenas acidental entre esses dois fatores.

Em algumas situações, o uso de uma linguagem ubíqua pode trazer problemas, embora estes não tenham sido experimentados neste trabalho. Trata-se dos casos em que as linguagens utilizadas no contexto de cada produto sejam diferentes. Por exemplo, suponha-se que termo "cartão" ou o termo "crédito" não fosse de emprego unânime nos sistemas de transporte urbano de todas as cidades. Nesse caso, a adoção de uma linguagem ubíqua seria difícil, dando margem a ambigüidades e uma maior dificuldade de compreensão do modelo. Haveria sempre a necessidade de uma "tradução" entre conceitos, algo que vai de encontro à própria idéia de linguagem ubíqua. Como DDD não foi pensado inicialmente para linhas de produtos, essa questão fica ainda em aberto.

Nesta dissertação, a comparação com a implementação de referência levou em conta apenas aspectos qualitativos, como a expressividade do modelo, a facilidade de reúso e relação entre responsabilidades e níveis de granularidade. Mas foi feita também uma comparação de aspectos quantitativos, baseados em métricas bem conhecidas de qualidade de código, como número de classes, número de métodos, estabilidade e métodos ponderados por classe (*weighted methods per class*).

Os resultados, porém, revelaram uma enorme discrepância entre os valores obtidos a partir da implementação baseada em DDD e aqueles obtidos a partir da implementação de referência. Tal discrepância — embora confirme as observações qualitativas — sugere que as duas implementações não podem ser confiavelmente comparadas usando-se esta abordagem. Embora se tenha procurado fazer medições de fatores que não dependessem da linguagem de programação, o que se observa é que as linguagens utilizadas (Java e Groovy) diferem muito entre si, o que as leva a interferir indiretamente no resultado dessas medições. De qualquer forma, os dados obtidos são apresentados no Apêndice C, para apreciação do leitor.

APÊNDICE

A

Backlog do sistema BET

O *backlog* do sistema BET é apresentado abaixo, contendo as histórias ordenadas segundo sua prioridade de negócios. Para cada história, além de seu enunciado, é mostrado um título, a iteração em que foi implementada e alguns comentários adicionais, quando necessário.

Tabela A.1: Backlog do produto para a linha de produtos BET

Título	**História de usário**	**Iteração**	**Comentários**
Aquisição de cartão	Como passageiro, quero adquirir um cartão de uma determinada categoria, para armazenar meus créditos.	1	
Restrição de cartões	A empresa de transporte municipal quer restringir as combinações de cartões permitidas por passageiro, para evitar perdas financeiras	1	Variabilidade. Pode-se implementar qualquer número de regras para combinação.
Inserção de créditos	Como atendente, quero efetuar operações de crédito no cartão de um passageiro, para que a empresa de viação possa fazer arrecadação pré-paga de passagens.	2	

Tabela A.1 – Continuação

Título	História de usário	Iteração	Comentários
Usuários corporativos	Como usuário corporativo, quero conceder crédito de passagens a todos os meus funcionários, como forma de benefício trabalhista (vale-transporte).	2	Variabilidade.
Cadastro de tarifas	A empresa de transporte municipal precisa cadastrar tarifas para que possa cobrar pelas passagens.	3	
Limite de passagens	A empresa de transporte municipal deseja limitar a quantidade de passagens que um usuário pode comprar por mês, para evitar perdas financeiras.	3	Variabilidade.
Consumo de créditos	A empresa de transporte municipal deseja que os créditos armazenados nos cartões sejam consumidos a cada viagem, para manter atualizados os saldos dos passageiros.	4	Parte desta história foi reusada da implementação de referência. Trata-se da aplicação cliente, que simula o validador de um ônibus.
Concessão de descontos	A empresa de transporte municipal precisa conceder descontos diferenciados por categoria, para atender requisições legais e de negócio.		
Gerenciamento de linhas	Como administrador, quero gerenciar as linhas, ônibus e horários do sistema de transporte municipal, para atender às demandas dos passageiros.		
Geração de dados de receita	O departamento financeiro da empresa de transporte precisa que sejam gerados dados de receita, para gerenciamento financeiro e contábil.		
Linhas de integração	Como passageiro, quero fazer mais de uma viagem pagando apenas uma passagem, para economizar meu saldo de créditos.		

Tabela A.1 – Continuação

Título	História de usário	Iteração	Comentários
Autenticação e autorização	Como administrador, quero restringir o acesso ao sistema de informações, de modo que somente as pessoas autorizadas possam operá-lo.		
Impressão de extrato	Como passageiro, quero ver um extrato de uso dos meus cartões, para controlar melhor os gastos com transporte.		
Gerenciamento de terminal e validadores	Como administrador, quero cadastrar e alterar dados de terminais e de validadores, para manter os dados atualizados em relação à rede de transporte.		

APÊNDICE

C

Comparação de medidas das implementações

Várias medidas de qualidade de código foram extraídas tanto da implementação de referência quanto da implementação baseada em DDD e métodos ágeis. As tabelas deste apêndice apresentam uma comparação entre as duas implementações para todas as medidas. A maioria das medidas possui nomes auto-explicativos. No entanto, a medida WMC (Weighted Methods per Class) merece uma explicação mais detalhada. Esta medida é a soma das complexidades dos métodos definidos na classe. Neste caso, a medida de complexidade dos métodos é a Complexidade Ciclomática de McCabe (McCabe, 1976).

Tabela C.1: Resumo das medidas feitas nas implementações

Medida	**Referência**	**DDD**
Número de classes	20	23
Número de métodos	249	66
WMC (Weighted Methods per Class)	271	128
Número de pacotes	9	5

Tabela C.2: Métodos ponderados por classe

Referência	**271**	**DDD**	**128**
Classe	**Valor**	**Classe**	**Valor**
GerenciaCartao	24	CardController	26
PassageiroDAO	14	PassengerController	15
GerenciaCartaoNumCartoes	28	CategoryController	15
CartaoMgr	32	CardAcquisitionService	4
CartaoDAO	14	Passenger	8
TipoPassageiroDAO	14	Card	5
NumCartoesDAO	13	Category	3
Passageiro	9	NoDuplicateCategories	2
Cartao	3	CardLimit	1
TipoPassageiro	15	CPF	4
GerenciaPassageiro	21	CorporateUser	4
GerenciaTipoPassageiro	19	CorporateUserController	21
EmpresaUsuariaDAO	11	EmployeeService	7
EmpresaUsuariaMgr	16	LegalIdentification	4
EmpresaUsuaria	15	CardCreditService	2
CargaCartao	12	CNPJ	5
AquisicaoCartao	11	CorporateUserNotFoundException	0
		EmployeeAssociationException	1
		CardNotFoundException	0
		InvalidCreditException	1
		CategoryNotAllowedException	0
		InvalidCpfException	0
		PassengerNotFoundException	0

Tabela C.3: Número de métodos por classe

Referência	**249**	**DDD**	**66**
Classe	**Valor**	**Classe**	**Valor**
GerenciaCartao	12	CardController	10
PassageiroDAO	13	PassengerController	8
GerenciaCartaoNumCartoes	15	CategoryController	8
CartaoMgr	29	CardAcquisitionService	3
CartaoDAO	11	Passenger	3
TipoPassageiroDAO	13	Card	4
NumCartoesDAO	5	Category	3
Passageiro	9	NoDuplicateCategories	1
Cartao	18	CardLimit	1
TipoPassageiro	14	CPF	2
GerenciaPassageiro	10	CorporateUser	4
GerenciaTipoPassageiro	9	CorporateUserController	9
EmpresaUsuariaDAO	10	EmployeeService	3
EmpresaUsuariaMgr	12	LegalIdentification	4
EmpresaUsuaria	21	CardCreditService	1
CargaCartao	7	CNPJ	2
AquisicaoCartao	9	CorporateUserNotFoundException	0
CargaCartaoEmpresaUsuaria	9	EmployeeAssociationException	0
GerenciaEmpresaUsuaria	10	CardNotFoundException	0
GerenciaPassageiroEmpresaUsuaria	13	InvalidCreditException	0
		CategoryNotAllowedException	0
		InvalidCpfException	0
		PassengerNotFoundException	0

Tabela C.4: Métricas de estabilidade

Pacote	Ca (Acoplamento aferente)	Ce (Acoplamento eferente)	I (Instabilidade)	A (Abstração)	D (Distância da seqüência principal)
Referência					
gerenciaCtrl	10	1	0,09	0,33	0,41
passageiroMgr	13	2	0,13	0,5	0,26
gerenciaCartaoNumCartoesCtrl	0	4	1	0	0
cartaoMgr	1	5	0,83	0	0,12
numCartoesMgr	3	1	0,25	0,5	0,18
web.aquisicaoCartao	0	1	1	0,5	0,35
web.cargaCartao	0	1	1	0,5	0,35
empresaUsuariaMgr	5	3	0,38	0,33	0,21
variabilidades.web.empresaUsuaria	0	4	1	0	0
DDD					
default	0	4	1	0	0
passenger	7	1	0,13	0,11	0,54
specification	1	0	0	0,33	0,47
corporation	1	3	0,75	0	0,18
fare	1	1	0,5	0	0,35

Referências Bibliográficas

ALLEN, D. *Seam in action*. Manning Publications Co., 2008.

ALUR, D.; CRUPI, J.; MALKS, D. *Core J2ee Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003.

BAUER, C.; KING, G. *Hibernate in action*. Manning, 2005.

BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

BLAKELEY, J.; CAMPBELL, D.; MURALIDHAR, S.; NORI, A. The ADO .NET entity framework: making the conceptual level real. *ACM SIGMOD Record*, v. 35, n. 4, p. 32–39, 2006.

BOEHM, B. A spiral model of software development and enhancement. *Computer*, v. 21, n. 5, p. 61–72, 1988.

BOOCH, G.; RUMBAUGH, J. J. I. *The unified modeling language user guide*. Addison-Wesley, 2000.

BOOTH, D.; HAAS, H.; MCCABE, F.; NEWCOMER, E.; CHAMPION, M.; FERRIS, C.; ORCHARD, D. Web Services Architecture. *W3C Working Group Note*, v. 11, p. 2005–1, 2004.

BRAGA, R. T. V.; RÉ, R.; MASIERO, P. C. A process to create analysis pattern languages for specific domains. In: *Sexta Conferência Latino-americana em Linguagens de Padrões para Programação*, 2007, p. 251–265.

BROOKS, F. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, v. 20, n. 4, p. 10–19, 1987.

BROOKS, F. P. *The mythical man-month and other essays on software engineering*. 1st ed. University of North Carolina, 1975.

BROWN, D.; DAVIS, C.; STANLICK, S. *Struts 2 in action*. Manning Publications Co., 2007.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

CHEESMAN, J.; DANIELS, J. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2000.

CHEESMAN, J.; DANIELS, J. *UML components*. Addison-Wesley Boston, 2001.

CLEMENTS, P.; NORTHROP, L. *Software product lines: Practices and patterns*. Addison-Wesley, 576 p., 2001.

CLEMENTS, P.; NORTHROP, L. *Software product lines*. Addison-Wesley Boston, 2002.

COCKBURN, A. *Writing effective use cases*. Addison-Wesley Boston, 2001.

COCKBURN, A. *Agile software development*. Boston: Addison-Wesley, 2002.

COCKBURN, A. *People and Methodologies in Software Development*. Tese de Doutoramento, University of Oslo Norway, 2003.

COHN, M. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 2004.

COHN, M. *Agile estimating and planning*. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005.

COOPER, J. *C# Design Patterns: A Tutorial*. Addison-Wesley Professional, 2003.

VAN DEURSEN, A.; VISSER, J. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, v. 35, n. 6, p. 26–36, 2000.

DIJKSTRA, E. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, p. 60–66, 1982.

DONEGAN, P. M. *Geração de famílias de produtos de software com arquitetura baseada em componentes*. Dissertação de Mestrado, Universidade de São Paulo, 2008.

DUB, J.; SAPIR, R.; PURICH, P. Oracle Application Server TopLink application developers guide, 10g (9.0. 4). *Oracle Corporation*, 2003.

EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.

EVANS, E.; FOWLER, M. Specifications. *Proceedings of the 1997 Conference on Pattern Languages of Programming*, p. 97–34, 1997.

FORRESTER *The total economic impact of using thoughtworks' agile development approach*. Relatório Técnico, Forrester Research, Inc., 2004.

FOWLER, M. Anemic domain model. 2003a.
Disponível em http://martinfowler.com/bliki/AnemicDomainModel.html

FOWLER, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003b.

FOWLER, M. Inversion of control containers and the dependency injection pattern. Acessado em setembro, 2008, 2004.
Disponível em http://martinfowler.com/articles/injection.html

FRAKES, W.; KANG, K. Software Reuse Research: Status and Future. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, p. 529–536, 2005.

FREEMAN, S.; MACKINNON, T.; PRYCE, N.; WALNES, J. Mock roles, not objects. In: *Companion to the ACM conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2004.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

GOLDBERG, A.; ROBSON, D. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1983.

GOMAA, H. *Designing software product lines with UML*. Addison-Wesley Boston, 2004.

GOMES, L. A. F.; BRAGA, R. T. V. Uso de padrões em linhas de produtos de software: Uma revisão sistemática. In: *7a Conferência Latino-Americana em Linguagens de Padrões para Programação*, 2008, p. 123–137.

GOOGLE Google guice. 2008.
Disponível em http://code.google.com/p/google-guice

GRISS, M. Implementing Product-Line Features by Composing Component Aspects. *First International Software Product-Line Conference*, 2000.

HOLOVATY, A.; KAPLAN-MOSS, J. *The definitive guide to Django: Web development done right*. Apress, 2007.

HU, Y.; PENG, S. So we thought we knew money. In: *Conference on Object Oriented Programming Systems Languages and Applications*, ACM, 2007, p. 971–975.

IEEE IEEE Recommended Practice for Software Requirements Specifications. *IEEE Computer Society Press*, v. 5, p. 35–37, 1998.

JACOBSON, I. Object-oriented development in an industrial environment. *ACM SIGPLAN Notices*, v. 22, n. 12, p. 183–191, 1987.

JACOBSON, I.; BOOCH, G.; RUMBAUGH, J. *The unified software development process*. Addison-Wesley Longman, Inc., 1999.

JENDROCK, E. *The Java EE 5 Tutorial*. Addison-Wesley Professional, 2006.

JOHNSON, R.; FOOTE, B. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1988.

KANG, K.; COHEN, S.; HESS, J.; NOVAK, W.; PETERSON, A. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *SEI, CMU, Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-21, Nov*, 1990.

KOENIG, D. *Groovy in action*. Manning Publications Co., 2007.

KOSKELA, L. *Test driven - practical tdd and acceptance tdd for java developers*. Manning Publications Co., 470 p., 2007.

KROLL, P.; KRUCHTEN, P. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley Professional, 2003.

LANDRE, E.; WESENBERG, H.; OLMHEIM, J. Agile enterprise software development using domain-driven design and test first. In: *OOPSLA 2007*, 2007.

LANDRE, E.; WESENBERG, H.; RØNNEBERG, H. Architectural improvement by use of strategic level domain-driven design. *Conference on Object Oriented Programming Systems Languages and Applications*, p. 809–814, 2006.

LARMAN, C. *Applying UML and patterns*. Prentice Hall PTR Upper Saddle River, NJ, 2002.

LARMAN, C. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2004.

LARMAN, C.; BASILI, V. Iterative and Incremental Development: A Brief History. *COMPUTER*, p. 47–56, 2003.

LISKOV, B.; WING, J. Behavioural subtyping using invariants and constraints. In: BOWMAN, H.; DERRICK, J., eds. *Formal Methods for Distributed Processing: A Survey of Object-Oriented Approaches*, Cambridge University Press New York, NY, USA, 2001, p. 254–280.

MACKENZIE, C.; LASKEY, K.; MCCABE, F.; BROWN, P.; METZ, R. Reference Model for Service Oriented Architecture 1.0. *Committee Specification, OASIS*, 2006.

MARTIN, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2003.

MCCABE, T. A Software Complexity Measure. *IEEE Trans. Software Engineering*, v. 2, n. 12, p. 308–320, 1976.

MCILROY, M. D. Mass-produced software components. In: BUXTON, J. M.; NAUR, P.; RANDELL, B., eds. *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, NATO Science Committee, 1968, p. 88–98.

MEYER, B. Applying design by contract. *Computer*, v. 25, n. 10, p. 40–51, 1992.

MILLER, J.; MUKERJI, J.; *et al.* MDA Guide Version 1.0. 1. *Object Management Group*, p. 03–06, 2003.

NILSSON, J. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*. Addison-Wesley Professional, 2006.

OMG Common object request broker archictecture. 1995.

OMG, O. M. G. Unified Modeling Language (UML), version 2.1.2. Superstructure Specification: formal/07-11-02, 2007.

PALMER, S.; FELSING, J. *A practical guide to feature-driven development*. Prentice Hall, 2002.

PANDA, D.; RAHMAN, R.; LANE, D. *EJB 3 in Action*. Manning Publications Co., 2007.

PARNAS, D. Designing Software for Ease of Extension and Contraction. *Transactions on Software Engineering*, p. 128–138, 1979.

PENG, S.; HU, Y. IAnticorruption: a domain-driven design approach to more robust integration. 2007.

PEREIRA, C. A. F.; BRAGA, R. T. V. Composição e geração de aplicações usando aspectos. In: *Proceedings of XII WTES in SBES '07*, 2007.

PIATELLI-PALMERINI, M. *Inevitable Illusions: How Mistakes of Human Reason Rule Our Minds*. John Wiley, 1996.

PICOCONTAINER 2008.
Disponível em http://www.picocontainer.org

PISCITELLO, D.; CHAPIN, A. *Open Systems Networking: TCP/IP and OSI*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1993.

REENSKAUG, T. *THING-MODEL-VIEW-EDITOR: an Example from a Planning System*. Relatório Técnico, Xerox PARC, 1979.

RICHARDSON, C. *POJOs in action*. Manning, 2006.

RICHARDSON, L.; RUBY, S. *Restful web services*. O'Reilly, 2007.

ROCHER, G. *The definitive guide to Grails*. Apress, 2006.

SCHWABER, K. Scrum Development Process. In: *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 1995), Austin, Texas, USA*, 1995, p. 117–134.

SCHWABER, K. *Agile Project Management with Scrum*. Microsoft Press, 2004.

SHIMABUKURO, E.; MASIERO, P.; BRAGA, R. Captor: Um Gerador de Aplicações Configurável. *Anais da XIII Sessão de Ferramentas do XX Simpósio Brasileiro de Engenharia de Software*, p. 121–128, 2006.

STONE, A.; RUSSELL, R.; PATTERSON, K. Transformational versus servant leadership: a difference in leader focus. *Leadership & Organization Development Journal*, v. 25, n. 4, p. 349–361, 2004.

SUN JSR 220: Enterprise JavaBeans - Version 3.0, JCP Standard. 2006.

SZYPERSKI, C.; BOSCH, J.; WECK, W. *Component Oriented Programming*. Springer, 1998.

TAKEUCHI, H.; NONAKA, I. The New New Product Development Game. *Harvard Business Review*, v. 64, n. 1, p. 137–146, 1986.

TELES, V. M. *Um estudo de caso da adoção das práticas e valores do extreme programming*. Dissertação de Mestrado, Universidade Federal do Rio de Janeiro, 2005.

THOMAS, D.; HANSSON, D. H. *Agile web development with rails*. 2nd. ed. Pragmatic Bookshelf, 2006.

THOMAS, M. IT Projects Sink or Swim. *British Computer Society Review*, 2001.

VERSIONONE *The state of agile development (3rd annual report)*. Relatório Técnico, VersionOne, 2008.

VOIGT, B. J. J.; GLINZ, M.; SEYBOLD, C. *Dynamic system development method*. Relatório Técnico, Department of Information Technology – University of Zurich, 2004.

WALLS, C.; BREIDENBACH, R. *Spring in Action*. Manning, 2007.

WEISS, D.; LAI, C. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

WESENBERG, H.; LANDRE, E.; RØNNEBERG, H. Using domain-driven design to evaluate commercial off-the-shelf software. *Conference on Object Oriented Programming Systems Languages and Applications*, p. 824–829, 2006.

WIRFS-BROCK, R.; WILKERSON, B. Object-oriented design: a responsibility-driven approach. In: *Conference on Object Oriented Programming Systems Languages and Applications*, 1989, p. 71–75.

WITTHAWASKUL, W.; JOHNSON, R. *Specifying Persistence in Platform Independent Models*. Relatório Técnico, University of Illinois at Urbana-Champaign, 2004.

YANG, Q.; LI, J.; WEISS, D. A Survey of Coverage-Based Testing Tools. *The Computer Journal*, 2007.

YEMELYANOV, A. *Using ado.net entity framework in domain-driven design: A pattern approach*. Dissertação de Mestrado, IT University of Göteborg, 2008.

YOURDON, E. *Death March*. Prentice Hall Ptr, 2004.