

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO
DO RIO GRANDE DO NORTE

Jorge Flávio Aquino da Costa

MODELAGEM DE SISTEMAS ORIENTADOS A OBJETOS
UTILIZANDO DOMAIN-DRIVEN DESIGN

NATAL
2008

JORGE FLÁVIO AQUINO DA COSTA

MODELAGEM DE SISTEMAS ORIENTADOS A OBJETOS
UTILIZANDO DOMAIN-DRIVEN DESIGN

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para obtenção do grau de Especialista em Sistemas Corporativos.

Orientador:
David Ricardo do Vale Pereira, M. Sc..

Natal
2008

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Costa, Jorge Flávio Aquino da

Modelagem de sistemas orientados a objetos utilizando Domain-Driven Design / Jorge Flávio Aquino da Costa. – Natal: FARN, 2008.
77f.

Orientador: David Ricardo do Vale Pereira (M.Sc.)

Monografia (Especialização em Desenvolvimento de Sistemas Corporativos) – Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Desenvolvimento de Softwares – Monografia. 2. Programação de Sistemas – Monografia. 3. Sistemas Orientados a Objetos – Monografia. 4. Domain-Driven Design – Monografia. I. Título.

RN/FARN/BC

CDU 004.4



JORGE FLÁVIO AQUINO DA COSTA

MODELAGEM DE SISTEMAS ORIENTADOS A OBJETOS
UTILIZANDO DOMAIN-DRIVEN DESIGN

Monografia apresentada ao Curso de Especialização em Desenvolvimento de
Sistemas Corporativos da Faculdade Natalense para o Desenvolvimento do Rio
Grande do Norte, aprovada em ____ de ____ de _____, pela banca
examinadora composta pelos seguintes membros:

David Ricardo do Vale Pereira

David Ricardo do Vale Pereira, M. Sc.
Orientador

Gleydson de Azevedo Ferreira Lima

Gleydson de Azevedo Ferreira Lima, M. Sc.
Membro Examinador

Membro Examinador

AGRADECIMENTOS

Em primeiro lugar, ao Grande Arquiteto que nos concedeu o dom da vida e nos permite enxergar a Luz em meio à escuridão.

Aos meus pais, pelos exemplos de caráter e dedicação, os quais moldaram a minha visão de mundo.

À Cláudia, minha grande incentivadora e companheira de todas as horas.

Aos programadores, analistas e tradutores que colaboram doando o seu tempo e conhecimento para a comunidade do *software* livre, fazendo-a crescer em quantidade mas sem perder a qualidade.

Por fim, mas não menos importante, ao meu orientador David Pereira e o coordenador da pós Gleydson Lima pelo apoio dado durante o desenvolvimento desta monografia.

*Nunca se queixe, nunca explique,
nunca se desculpe. Aja ou saia.
Faça ou vá embora.
(ex-primeiro ministro inglês
Benjamim Disraeli)*

*Quando nada parece ajudar,
eu vou e olho o cortador de pedras
martelando sua rocha, talvez cem vezes,
sem que nem uma só rachadura apareça.
No entanto, na centésima primeira martelada,
a pedra abre em duas, e eu sei
que não foi aquela a que conseguiu,
mas todas as que vieram antes.
(Jacob Riis)*

RESUMO

Ao longo das últimas décadas, a área da Tecnologia da Informação relacionada à programação de sistemas evoluiu de forma rápida e constante, só perdendo em velocidade para a expansão acelerada da Internet, cujas velocidades e formas de acesso se multiplicam constantemente. Atualmente, utilizam-se conceitos e técnicas "estado-da-arte" no desenvolvimento de softwares, o que exige do programador um farto conhecimento sobre várias linguagens, assim como um ou mais Integrated Development Environments (IDE). Dentre essas técnicas, podemos citar os Padrões de Projeto (Design Patterns), Test-Driven Development (TDD), Metodologias Ágeis, Domain-Driven Design (DDD) e Linguagens de Domínio (Domain Specific Languages). Saber qual linguagem ou qual ferramenta utilizar no projeto e construção de um sistema tornou-se a principal dificuldade de um Analista de Software. Porém, as técnicas citadas acima formam um conjunto que visa aumentar de forma substancial a produtividade e eficiência dos mantenedores de grandes sistemas corporativos em produção na atualidade. Um dos problemas freqüentes que se apresenta aos responsáveis por esses sistemas é a sua manutenção. Cada vez mais, os projetistas percebem a necessidade de se elaborar uma modelagem elegante, eficiente e que torne simples a alteração no código e parâmetros do sistema. O Domain-Driven Design (Projeto Baseado no Domínio) permite ao Analista de Software modelar um sistema fiel o bastante ao mundo real, abstraindo suas peculiaridades, facilitando sobremaneira o trabalho dos desenvolvedores e melhorando a comunicação entre estes e os clientes. Pretende-se, com este trabalho, demonstrar as vantagens e características do DDD aplicado ao projeto de sistemas orientados a objetos. Este será iniciado com uma descrição mais detalhada sobre o DDD para, em seguida, apresentar um estudo de caso sobre sua utilização em um projeto de software.

Palavras-chave:

DDD. Domain-Driven Design. Engenharia de Software. Java. POO. Programação Orientada a Objetos. DSL. Linguagens de Domínio.

ABSTRACT

Over the last few decades, the Information Technology applied to systems programming evolved in a fast and constant way, staying behind only from the Internet accelerated expansion, which speeds and access methods are constantly being multiplied. Nowadays, state-of-the-art concepts and techniques have been used in software developing, what demands from programmer a strong knowledge about a lot of languages and Integrated Development Environments (IDE). Among those techniques, Design Patterns, Test-Driven Development (TDD), Agile Methodologies, Domain-Driven Design (DDD) and Domain Specific Languages (DSL) are frequently highlighted. Choose what language or framework to use in a system project or building became the most difficult concern of a Software Analyst. However, the techniques cited backwards form a set dedicated to substantially raise productivity and efficiency of modern large corporate systems maintainers. One of the recurrent problems presented to these systems coordinators is its maintenance. Designers have realized a need to elaborate some elegant and efficient modeling that will simplify modifications to system parameters and its source code. The Domain-Driven Design allows Software Analysts to model a system very similar to its counterpart in real world, by abstracting its peculiarities, resulting in an improved communication between clients and analysts and, as a side-effect, easing the last ones job.

This monograph main objective is to demonstrate the features and advantages of DDD applied to object-oriented systems design. At beginning, a detailed description about DDD will be presented and, in sequence, a case study will be developed by using it in a software project.

Keywords:

DDD. Domain-Driven Design. Software Engineering. Java. OOP. Object-Oriented Programming. DSL. Domain Languages.

LISTA DE ILUSTRAÇÕES

Figura 1 – Teoria dos Conjuntos (Operação de União)	15
Figura 2 – Modelo de desenvolvimento Cascata (waterfall)	17
Figura 3 – Modelo de Desenvolvimento Espiral	18
Figura 4 – Astrolábio Medieval	20
Figura 5 – Modelo de Pedido (versão 1)	21
Figura 6 – Modelo do Pedido (versão 2)	22
Figura 7 – Arquitetura do Model-Driven Design	23
Figura 8 – Arquitetura em Camadas no DDD	25
Figura 9 – Arquitetura em Camadas (com serviços)	29
Figura 10 – Modelo de Fábrica (Factory Method)	32
Figura 11 – Modelo de Fábrica Genérica	33
Figura 12 – Fluxograma de um Repositório	36
Figura 13 – Fluxograma de um Repositório (utilizando Fábrica)	39
Figura 14 – Etapas e Disciplinas do RUP	40
Figura 15 – Modelo da Concessionária (Revisão 1)	48
Figura 16 – Modelo da Concessionária (Revisão 2)	53
Figura 17 – Modelo da Concessionária (Revisão 3)	55
Figura 18 – Modelo da Concessionária (Revisão 4)	59
Figura 19 – Modelo da Concessionária (Versão Final)	63

LISTA DE ABREVIATURAS E SIGLAS

AJAX	<i>Asynchronous Javascript And XML</i> (Javascript Assíncrono e XML)
AOP	<i>Aspect-Oriented Programming</i> (Programação Orientada a Aspectos)
BD	<i>Banco de Dados</i>
CPF	<i>Cadastro de Pessoa Física</i>
CPU	<i>Central Processing Unit</i> (Unidade Central de Processamento)
DAO	<i>Data Access Objects</i> (Objetos de Acesso a Dados)
DDD	<i>Domain-Driven Design</i> (Projeto Baseado no Domínio)
DOS	<i>Disk Operating System</i> (Sistema Operacional de Disco)
DSL	<i>Domain-Specific Language</i> (Linguagem Específica do Domínio)
E/S	<i>Entrada/Saída</i>
EBS	<i>Evidence-Based Scheduling</i> Alocação Baseada em Evidências
GPS	<i>Global Positioning System</i> (Sistema Global de Posicionamento)
GSM	<i>Global System for Mobile Communications</i> (Sistema Global para Comunicações Móveis)
GUI	<i>Graphical User Interface</i> (Interface Gráfica do Usuário)
HTML	<i>HyperText Markup Language</i> (Linguagem de Marcação em HiperTexto)
ICMS	<i>Imposto sobre Circulação de Mercadorias e prestação de Serviços</i>
IDE	<i>Integrated Development Environment</i> (Ambiente de Desenvolvimento Integrado)
ISS	<i>Imposto Sobre Serviços (de qualquer natureza)</i>
JSF	<i>Java Server Faces</i>
KBPS	<i>Kilobits Por Segundo</i>

LCD	<i>Liquid Crystal Display</i> (Visor de Cristal Líquido)
MDD	<i>Model-Driven Design</i> (Projeto Baseado no Modelo)
MVC	<i>Model-View-Controller</i> (Modelo-Visão-Controlador)
OO	<i>Orientados a Objetos</i>
ORM	<i>Object-Relational Mapping</i> (Mapeamento Objeto-Relacional)
PDA	<i>Personal Digital Assistant</i> (Assistente Pessoal Digital)
POJO	<i>Plain Old Java Object</i> ("Simples" Objetos Java Antigos)
POO	<i>Programação Orientada a Objetos</i>
RAM	<i>Random Access Memory</i> (Memória de Acesso Aleatório)
RUP	<i>Rational Unified Process</i> (Processo Unificado da Rational™)
SQL	<i>Structured Query Language</i> (Linguagem de Consulta Estruturada)
TDD	<i>Test-Driven Development</i> (Desenvolvimento Baseado em Testes)
TI	<i>Tecnologia da Informação</i>
UID	<i>Universal IDentificator ou User IDentification</i> (IDentificador Universal ou IDentificação do Usuário)
UML	<i>Unified Modeling Language</i> (Linguagem de Modelagem Unificada)
VO	<i>Value Object</i>
XML	<i>eXtensible Markup Language</i> (Linguagem de Marcação Extensível)
XP	<i>eXtreme Programming</i> (Programação Extrema)

SUMÁRIO

1 INTRODUÇÃO	11
2 DOMAIN-DRIVEN DESIGN	14
2.1 UBIQUITUOUS LANGUAGE	16
2.2 MODEL-DRIVEN DESIGN	20
2.2.1 Entidades	26
2.2.2 Value Objects	27
2.2.3 Serviços	27
2.2.4 Módulos	28
2.2.5 Aggregates	30
2.2.6 Fábricas	31
2.2.7 Repositórios	34
2.3 CONSIDERAÇÕES FINAIS	35
3 ESTUDO DE CASO	40
3.1 APRESENTAÇÃO DO DOMÍNIO	41
3.1.1 Personagens	42
3.2 INICIAÇÃO	43
3.2.1 Reunião Inicial	43
3.2.2 Coleta de Dados	44
3.3 ELABORAÇÃO	47
3.3.1 Criação da Linguagem Ubíqua	47
3.3.2 Componentes do Model-Driven Design	52
3.4 CONSTRUÇÃO	61
3.5 TRANSIÇÃO	64
3.6 CONSIDERAÇÕES FINAIS	69
4 CONCLUSÃO	70
4.1 PERSPECTIVAS FUTURAS	71
REFERÊNCIAS	72
APÊNDICE	75

1 INTRODUÇÃO

Considerando a evolução pela qual passou a informática nas últimas décadas, podemos descrever dois padrões que se sucederam até os dias atuais. No primeiro momento, existiam imensos computadores valvulados que ocupavam andares inteiros de prédios. Esses equipamentos tinham como características principais a limitação de memória, tanto a volátil (RAM) como a permanente, e capacidade de processamento reduzida. Nesse período, os programas eram sempre desenvolvidos visando à economia de espaço e eficiência da CPU¹. Ou seja, o programador devia buscar formas de otimização de seu código, ao mesmo tempo em que deveria se preocupar com a forma de gravação dos dados persistentes; estes últimos, em geral, eram salvos em formato binário², podendo ser codificados e comprimidos. Para se conseguir a eficiência desejada, os programadores utilizavam as linguagens de "baixo nível", sendo assim chamadas devido à sua proximidade com a linguagem de máquina.

Nesta fase inicial, não havia projeto formal de sistemas informatizados, e os programas eram escritos de forma estruturada, fazendo uso extensivo de saltos (*GOTO*³) e laços (*loops*⁴) de código. Esse período se perpetuou até o final da década de 70, quando máquinas mais modernas foram lançadas no mercado e a primeira linguagem Orientada a Objetos se difundiu mais fortemente. A *Smalltalk* revolucionou a forma de se programar, baseada num novo paradigma de que o software deveria ser projetado levando em consideração a reutilização de código, facilidade de manutenção e uma aproximação maior das estruturas de dados com o mundo real.

Chegou-se, então, ao segundo período da evolução da informática, onde o hardware se tornava cada vez mais poderoso, as mídias de armazenamento cada

1 Central Processing Unit: Unidade Central de Processamento de um computador, responsável por executar os programas e controlar os periféricos de Entrada (*mouse*, *teclado*) e Saída (*monitor*, *impressora* etc)

2 Não obstante todos os dados efetivamente gravados num computador, seja na memória volátil ou permanente, estejam no formato binário de 0s e 1s, utilizou-se essa denominação para diferenciar dos arquivos que possam ser lidos por editores de texto ASCII, tais como TXT ou XML.

3 Desvio incondicional na execução de um programa

4 Trechos de um código-fonte que são executados por um número de vezes determinado pelo programador, de forma pré-fixada ou através de variáveis

vez mais baratas e com maior capacidade, além da disseminação das redes interconectadas. Essa conjunção de fatores permitiu que os analistas e desenvolvedores de software explorassem mais a fundo a capacidade dos computadores, fazendo uso das linguagens de "alto nível" em geral, e das linguagens Orientadas a Objetos em particular.

A partir deste momento, o programador pôde se dedicar mais às regras do negócio, ao escopo do seu problema, sem se preocupar com as minúcias da arquitetura da CPU ou otimizações de código e acessos diretos a hardware. Nas linguagens de alto nível, esses detalhes foram abstraídos na forma de objetos e funções genéricas de E/S. Até mesmo os algoritmos mais comuns – tais como ordenamento de listas e *hashing*⁵ – foram incorporados à biblioteca de funções dessas linguagens.

Entre as metodologias de desenvolvimento, a que mais se beneficiou desta evolução do poder de processamento foi a Programação Orientada a Objetos (POO), pois tornou-se possível implementar linguagens dinâmicas, com tipagem fraca e interpretadas sem que o desempenho fosse comprometido. Conceitos como herança, poliformismo e encapsulamento foram introduzidos ao vocabulário e ao arsenal do analista de sistemas. O desenvolvimento baseado em Objetos permitiu uma maior reusabilidade de código (GAMMA et al, 2000), ao mesmo tempo que facilitou o seu entendimento, uma vez que se passou a trabalhar com estruturas de dados complexas, que permitiam simular melhor a realidade. Os objetos na POO, assim como suas contrapartes no mundo real, possuem comportamento – representado por suas funções – e estado – suas variáveis internas (RUMBAUGH et al, 1994) e, por conseguinte, reproduzem a dinâmica dos sistemas encontrados na maioria dos problemas computacionais cotidianos.

Entretanto, mesmo com essa inovação, alguns hábitos permaneceram e os programadores não souberam aproveitar toda a potencialidade da POO. Em sua maioria, e ainda nos dias atuais, o que se encontra são programas escritos em linguagens orientadas a objetos mas de forma quase exclusivamente procedural. A principal característica desse tipo de comportamento é a inexistência de atribuições

5 Estrutura de dados sequencial, que armazena as informações utilizando chaves codificadas, as quais identificam unicamente cada item da lista (pares chave-valor), diminuindo o tempo necessário na pesquisa e recuperação dos dados.

bem-definidas aos objetos e pouca interação entre eles. Nesta situação, o que se observa “[...] são apenas dados fluindo [...]” (CALÇADO, 2007) sendo controlados por uma função ou objeto denominado de Mestre dos Fantoches, concentrando o fluxo de ações do programa como se fosse uma grande função *main()*⁶.

É justamente para preencher essa lacuna que surge o *Domain-Driven Design* (DDD), com a sua proposta de auxiliar o analista no projeto de um sistema informatizado, a fim de aproveitar os benefícios da POO, unificando e estreitando a comunicação entre todos os envolvidos no processo de criação, inclusive os clientes. Seu objetivo primordial se concentra em projetar um sistema o mais fiel possível ao domínio do problema. Essa tarefa é facilitada enormemente quando se lida com objetos, ao invés de estruturas de dados isoladas que dificultam o entendimento da natureza do problema, acarretando em maiores esforços para manutenção do código por terceiros, prejudicando a escalabilidade e reusabilidade do sistema. Por isso, é tão necessário um sólido conhecimento sobre a Programação e Análise Orientadas a Objetos para a aplicação efetiva do DDD.

No próximo capítulo, serão abordados em mais detalhes os princípios e características fundamentais do *Domain-Driven Design*. Em seguida, será exposto um estudo de caso, simulando o projeto de um sistema utilizando o DDD. E, por último, seguem as conclusões.

6 A principal função e ponto de partida de um programa escrito na linguagem C.

2 DOMAIN-DRIVEN DESIGN

Ao contrário de outras metodologias de Engenharia de Software, como o Processo Unificado da Rational – *Rational-Unified Process (RUP)* – ou a Programação Extrema – *eXtreme Programming (XP)*, o Domain-Driven Design não é uma técnica ou uma ferramenta nas quais se produzem artefatos palpáveis, como os Casos de Uso (*Use Cases*) do RUP (AMBLER, 2005). O DDD é caracterizado e utilizado mais como filosofia de desenvolvimento, com idéias e conceitos que facilitam a vida dos programadores e clientes.

Essa peculiaridade do DDD é o que o torna tão abstrato, pois não existe um conjunto de regras fixas e passos seqüenciais que direcionem sua aplicação nos projetos. Entretanto, embora seja difícil aplicar os conceitos do DDD durante o desenvolvimento de um programa, é relativamente fácil identificar quando eles não estão sendo utilizados. Nesses casos, o que se encontram são funções inchadas, com dezenas de linhas de código, que concentram diversas tarefas – às vezes não relacionadas entre si. Ou, ainda, no campo da POO, destacam-se objetos mal projetados absorvendo poucas ou excessivas responsabilidades (Fantoques). Essas falhas de projeto são o que Fowler et al (1999, p.63) chamam de “cheiro ruim” (*bad smell*) dentro de um sistema, no qual à primeira vista têm-se a impressão que existe algo errado e, numa observação mais cuidadosa, percebe-se a ausência de boas práticas de projeto (FOWLER, 2002).

Para o primeiro caso, uma abordagem inteligente seria o uso dos Padrões de Projeto (*Design Patterns*). Os *Design Patterns* são soluções testadas e bem documentadas para os problemas computacionais mais comumente encontrados (GAMMA et al, 2000). Embora tenham sido compilados há alguns anos, apenas recentemente sua adoção se expandiu e começaram a ser utilizados de forma mais abrangente pelos desenvolvedores. No segundo caso, ocorre uma situação na qual se pode aplicar enfaticamente o DDD, auxiliado pela técnica da Programação Orientada a Aspectos – *Aspect-Oriented Programming (AOP)*, que consiste em separar do domínio o código que não lhe pertence exclusivamente (histórico, autenticação de usuário etc), isolando a camada de domínio apenas com o código relativo a ela. Neste contexto, o *Domain-Driven Design* evoluiu sobretudo após a

adoção generalizada dos sistemas orientados a objetos, como forma de melhorar o projeto destes pois, como o próprio nome sugere, é uma maneira de desenvolver focando no domínio (ou cerne) do problema.

Faz-se necessário então, a partir deste ponto, esclarecer o conceito de domínio. Traçando um paralelo com a Teoria dos Conjuntos da Matemática, o domínio de uma função é um subconjunto de um conjunto - ou um subconjunto de uma relação entre conjuntos - cujos elementos são possíveis argumentos para a função, conforme ilustrado na Figura 1. Fazendo uma analogia com um software, este seria a função e o domínio seriam os elementos com os quais o software pode trabalhar. Logo, o domínio de um sistema físico seria o subconjunto de todas as condições e variantes, representáveis de forma computacional, relevantes para o problema que se deseja resolver. Considerando que no mundo real existem objetos – animados ou inanimados – que interagem entre si, verifica-se que a POO é uma tecnologia que atende satisfatoriamente aos requisitos de modelagem e aproximação do código-fonte ao escopo do problema.

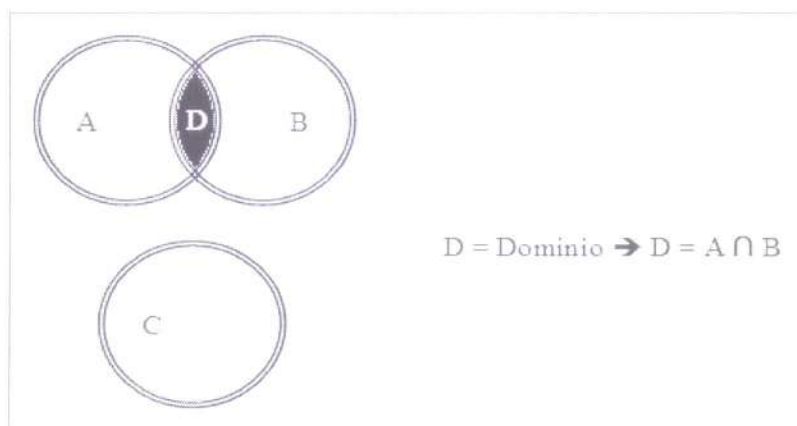


Figura 1 – Teoria dos Conjuntos (Operação de União)

Fonte: Dados da Pesquisa (2007)

Aliando a versatilidade da POO com a elegância do DDD, é possível projetar sistemas bastante fiéis às suas contrapartes físicas, mas este trabalho não é de exclusividade do desenvolvedor. Uma das características do DDD, que é compartilhada pela metodologia XP, são as reuniões freqüentes entre os analistas de software, desenvolvedores e clientes para discussões acerca do projeto e

modelagem do sistema (TELES, 2007). A diferença mais proeminente é a presença não somente de gerentes de Tecnologia da Informação (TI) ou dos usuários finais, diretamente interessados no sistema, mas também de especialistas que conhecem a fundo as metodologias e processos da empresa em questão.

Por exemplo, numa reunião fictícia para o projeto de um sistema automatizado destinado a uma empresa de *courrier* (transporte de encomendas), seria necessária a presença dos gerentes de TI, dos futuros operadores do software (motoristas, empacotadores, recepcionistas de balcão etc) e, principalmente, dos técnicos encarregados da logística de transporte pois estes, mais do que os demais, possuem um conhecimento mais minucioso do domínio relacionado ao processo produtivo da empresa. Cada participante possui um papel relevante a ser considerado: o gerente está consciente da necessidade do sistema; os operadores finais têm a visão de como o software deverá se comportar, no contexto da interação com o usuário; e os especialistas sabem como o sistema deve funcionar, em termos de entradas e saídas – para cada entrada de dados, uma saída correspondente e previsível.

Em algumas situações, os próprios usuários finais do sistema são os especialistas no domínio e, nesse caso, a etapa de projeto flui mais rapidamente uma vez que mais informações podem ser obtidas de menos pessoas, reduzindo assim a frequência e duração das reuniões, nas quais ocorre o que Evans (2003, p. 31) chama de trituração ou mastigação do conhecimento (*knowledge crunching*). É durante esta troca constante de informações entre o desenvolvedor e os especialistas que se molda a base do sistema final.

2.1 UBIQUITUOUS LANGUAGE

Uma das maiores queixas sobre a construção de software decorre da dificuldade que os desenvolvedores têm de entregar ao cliente exatamente aquilo que ele deseja e, mais ainda, dentro do prazo previsto. O segundo caso é mais difícil de corrigir porque envolve dados empíricos e bastante subjetivos sobre quanto tempo alocar para cada tarefa de codificação e implementação do software. Os programadores em geral costumam subestimar o tempo necessário para concluir

suas funções e componentes de código, desconsiderando os atrasos decorrentes de reuniões, testes, correções de *bugs*¹ entre outros. No entanto, Spolsky (2007) desenvolveu um método, chamado de Alocação Baseada em Evidências (*Evidence-Based Scheduling* – EBS), para prever o tempo necessário à realização das tarefas de programação. Seu método é baseado em dados históricos, consolidados pelos próprios programadores que registraram os tempos gastos durante a codificação de projetos anteriores, e ajuda a produzir um cronograma mais realista.

Quanto ao primeiro caso, a Figura 2 ilustra de forma bem-humorada o que acontece no Modelo de Desenvolvimento em Cascata, em que todas as etapas ocorrem de forma rígida e seqüencial.

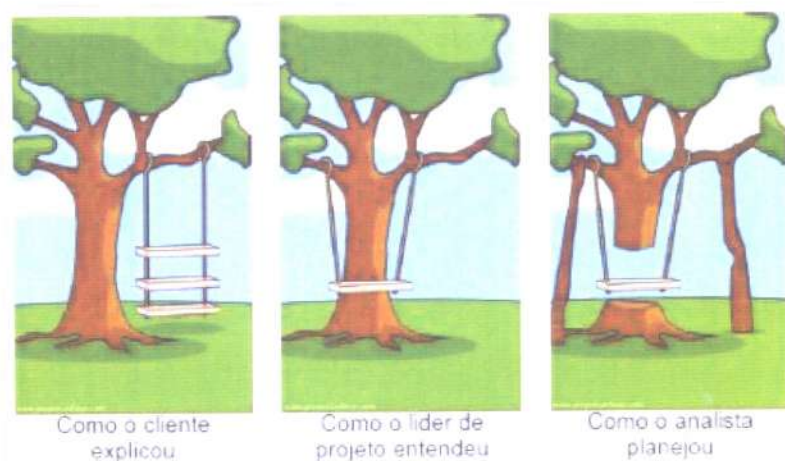


Figura 2 – Modelo de desenvolvimento Cascata (waterfall)

Fonte: Adaptado de Project Cartoon (2007)

O cliente comunica seu problema ao gerente de projeto, que repassa ao analista o qual, por sua vez, elabora uma especificação técnica e a transmite ao programador. Este último, então, inicia o processo de desenvolvimento utilizando apenas os documentos recebidos pelo analista, tendo pouca ou nenhuma interação com os demais envolvidos no projeto do sistema. Logo, é fácil perceber como existe uma enorme possibilidade de o produto gerado ser bastante diferente do esperado.

Cientes dessa dificuldade, os analistas e gerentes de projeto passaram a adotar técnicas de projeto iterativas, como a metodologia em *Espiral*, mostrada na

¹ Palavra inglesa cuja tradução literal é insetos. No jargão técnico da informática, denomina as falhas de programação que geram erros na execução de um programa.

Figura 3. Levando a idéia de ciclos ao extremo, os adeptos das metodologias Ágeis, tais como XP e SCRUM (SCHWABER, 2007), defendem *loops* realmente pequenos, em torno de horas e dias, e foco total na codificação. Com isso, obtêm-se protótipos funcionais do software logo nas primeiras iterações, facilitando as futuras correções de rumo no projeto e, como consequência adicional, maior satisfação ao cliente por visualizar logo de antemão como funcionará seu sistema.



Figura 3 – Modelo de Desenvolvimento Espiral

Fonte: Dados da Pesquisa (2007)

Porém, mesmo com iterações freqüentes e curtas, protótipos prematuros e reuniões com os clientes, não se pode garantir que o sistema final será mesmo o desejado devido, em parte, aos padrões de linguagem utilizados pelos analistas, desenvolvedores e clientes. Cada um desses atores é especialista na sua área e, portanto, tende a usar seus próprios termos e jargões técnicos, dificultando o entendimento entre eles. O cliente, em geral, sabe o quer, mas não sabe como quer e tenta explicar usando os termos da sua área. O analista traduz as necessidades do cliente para Casos de Uso, Diagramas em Linguagem de Modelagem Unificada (*Unified Modeling Language* – **UML**) e demais artefatos, a fim de transmitir formalmente as especificações do sistema para o desenvolvedor. Este, então, traduz a especificação formal em *Classes* (de Objetos), Funções e Tabelas de Banco de Dados dando forma ao sistema.

Essas diferenças de terminologias geram um entrave na comunicação e, inevitavelmente, informações são perdidas durante as “traduções” de uma linguagem para outra. Para contornar essa dificuldade, o *Domain-Driven Design* adota a idéia de uma linguagem universal, criada para facilitar o entendimento do domínio e agilizar a troca de conhecimentos entre os especialistas no domínio e os analistas de software. Em seu livro, Evans (2003) chamou tal linguagem de Linguagem Ubíqua (*Ubiquitous Language*²), cuja criação e evolução se dá ao longo das reuniões de *knowledge crunching*.

A Linguagem Ubíqua é derivada do domínio da aplicação e, por conseguinte, deve conter os termos mais utilizados pelos especialistas. Porém, ela não precisa necessariamente ser apenas textual, podendo se utilizar de gráficos e figuras para complementar a expressão das idéias, tais como diagramas Entidade-Relacionamento e diagramas de seqüência. É importante que todos os envolvidos, desde os clientes até os programadores, participem das reuniões de *knowledge crunching* e contribuam para a criação da Linguagem Ubíqua, para se familiarizarem com seus termos logo no início do projeto.

Conforme observação de (AVRAM (comp.); MARINESCU (org.), 2006), a estruturação da Linguagem Ubíqua não ocorre imediatamente, podendo necessitar de várias semanas ou até meses, dependendo da complexidade do domínio, para se chegar a uma aproximação satisfatória. Os especialistas do domínio, em conjunto com os analistas e programadores, precisarão constantemente revisar os conceitos e termos a fim de encontrar imprecisões na modelagem ou adicionar novos itens que não estavam previstos no projeto inicial. E, uma vez que a Linguagem Ubíqua está firmemente amarrada ao domínio, uma mudança neste deve refletir também naquela e vice-versa.

2 Alguns autores utilizam também o termo Onipresente no lugar de Ubíqua para traduzir o termo *Ubiquitous*. Neste trabalho, entretanto, foi escolhida a expressão Linguagem Ubíqua devido a sua similaridade com o original em inglês e, também, por já ser uma palavra encontrada no campo da informática, na área de pesquisa da Computação Ubíqua (Onipresente).

2.2 MODEL-DRIVEN DESIGN

O objetivo primordial do *Domain-Driven Design* é modelar da forma mais fiel possível o domínio de um sistema, de modo que este possa ser representado numa linguagem de programação. Esse processo, conhecido como *Model-Driven Design* (Projeto Direcionado ao Modelo), gera, entre outros produtos, a Linguagem Ubíqua discutida no tópico anterior. O modelo do sistema deve ser firmemente amarrado ao domínio e, como efeito, o código gerado irá implementar o modelo dentro de um ambiente computacional, numa linguagem de programação específica. Um exemplo de modelagem pode ser observado na Figura 4.



Figura 4 – Astrolábio Medieval

Fonte: Evans (2003)

Em seu livro, Evans (2003, p.58) compara o Astrolábio desenvolvido na Idade Média a um Computador moderno, quando diz que "[...] o astrolábio é uma implementação mecânica de um modelo do céu [...]" na medida em que, fornecendo como entrada uma data e hora qualquer, era possível se calcular a posição do sol e das estrelas no firmamento naquele instante de tempo escolhido. Neste caso, o domínio é o espaço sideral, e o modelo representa os corpos celestes e suas movimentações ao longo do tempo. É importante ressaltar que o modelo é um subconjunto do domínio, determinado pelas funcionalidades que os projetistas desejam abstrair do sistema em estudo.

Dependendo da complexidade do domínio, torna-se impossível modelar toda a estrutura do mesmo. Porém, todo domínio pode ser descrito na forma de diversos modelos, e cada modelo, por sua vez, também pode ser implementado de diferentes maneiras no código. Mas, apesar das afirmativas anteriores, "[...] a existência de um modelo analiticamente correto não implica que o mesmo possa ser diretamente expresso na forma de código funcional" (AVRAM (comp.); MARINESCU (org.), 2006, p. 32). A explicação para isso é que o modelo poderia violar alguns princípios do projeto ou adotar abstrações não permitidas pela plataforma ou linguagem de programação utilizada.

Mais uma vez, Evans (2003) reforça a necessidade da presença dos programadores nas reuniões de *knowledge crunching*, a fim de garantir a viabilidade da implementação do modelo pois, em última instância, são eles que irão codificar o sistema e conhecem suas limitações práticas. De volta ao exemplo da empresa de *courrier*, um possível modelo para os objetos Cliente e Pedido foi criado pelos analistas de software, demonstrado na Figura 5.

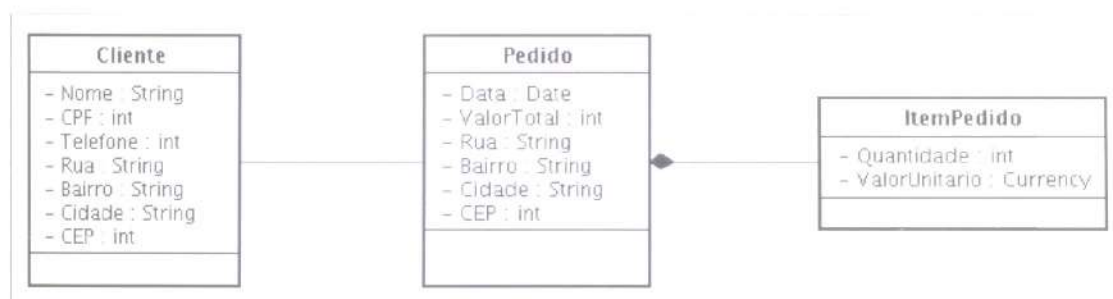


Figura 5 – Modelo de Pedido (versão 1)

Fonte: Dados da Pesquisa (2008)

O modelo desenvolvido na Figura 5 leva em consideração que existem dois endereços distintos que serão utilizados durante uma transação comercial da empresa: um endereço para entrega da encomenda, e outro para a cobrança, os quais podem ser diferentes para o mesmo cliente, daí a necessidade de separá-los em objetos distintos. Em geral¹ isso ocorre quando há entrega de presentes a terceiros, facultando ao cliente solicitar o envio apenas da encomenda ao destinatário, e a fatura no seu endereço residencial.

Após algumas iterações e reuniões, chegou-se à fase de projeto do módulo de pagamento, junto aos analistas e contadores da empresa. Durante as discussões (*knowledge crunching*), foi levantada a questão concernente ao modo de pagamento e o endereço de cobrança: os técnicos da empresa de *courrier* observaram que a maioria das encomendas é entregue no endereço próprio do cliente, assim como a suas faturas e notas fiscais. Sendo assim, o endereço de entrega e cobrança seriam idênticos.

Os programadores, que não estavam presentes às primeiras reuniões, alertaram para o fato de haver bastante duplicidade de informações no banco de dados. Os analistas, em conjunto com os técnicos da empresa e os programadores, chegaram então à uma nova proposição para o modelo do Cliente e da Encomenda, ainda considerando a possibilidade de endereços distintos para cobrança e entrega, e eliminando o espaço desnecessário oriundo da gravação duplicada.

A nova versão do modelo é demonstrada na Figura 6 e valida a afirmação anterior de que, mesmo encontrando um modelo tecnicamente correto, nem sempre ele se mostra a melhor opção, forçando o analista a freqüentemente reavaliar seu projeto.

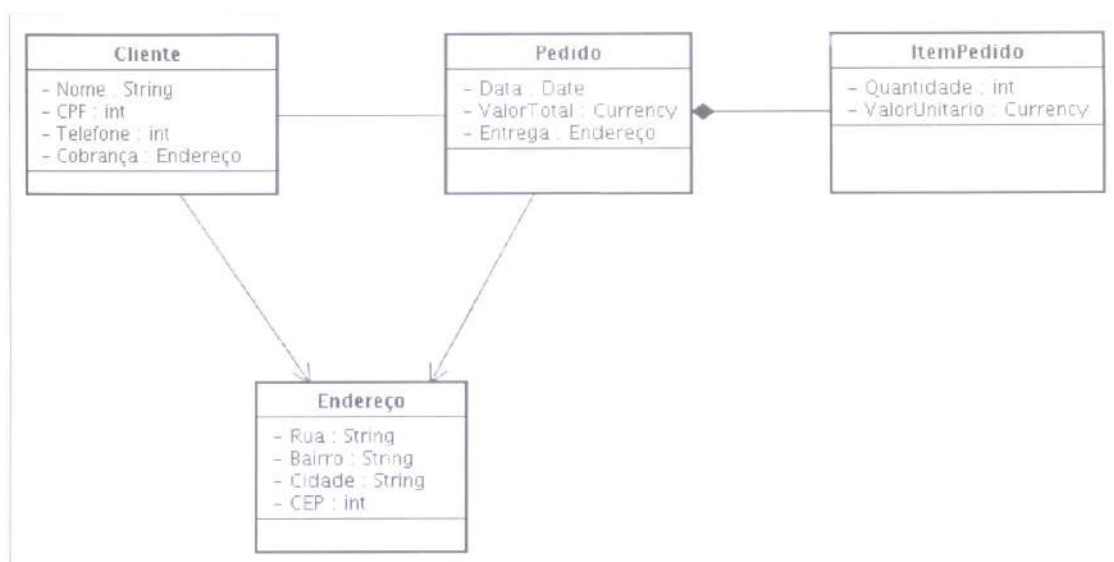


Figura 6 – Modelo do Pedido (versão 2)

Fonte: Dados da Pesquisa (2008)

Além dos objetos propriamente ditos (POJO's³), o *Model-Driven Design* faz uso de alguns conceitos auxiliares na modelagem de um sistema, os quais estão relacionados na Figura 7. Destes, apenas a Interface Gráfica com o Usuário (**GUI** – *Graphical User Interface*) e a Arquitetura em Camadas não fazem parte do MDD mas interagem diretamente com ele.

A interface gráfica (GUI) serve de ponte entre o operador de um sistema e o domínio da aplicação, permitindo ao usuário realizar ações e obter as saídas, ou resultados, desejados. Embora nem todas as interfaces sejam estritamente gráficas, para fins de generalização iremos considerá-las assim visto que, atualmente, até mesmo os sistemas de controle para plantas industriais adotam telas de cristal líquido (LCD) para comando e visualização dos processos.

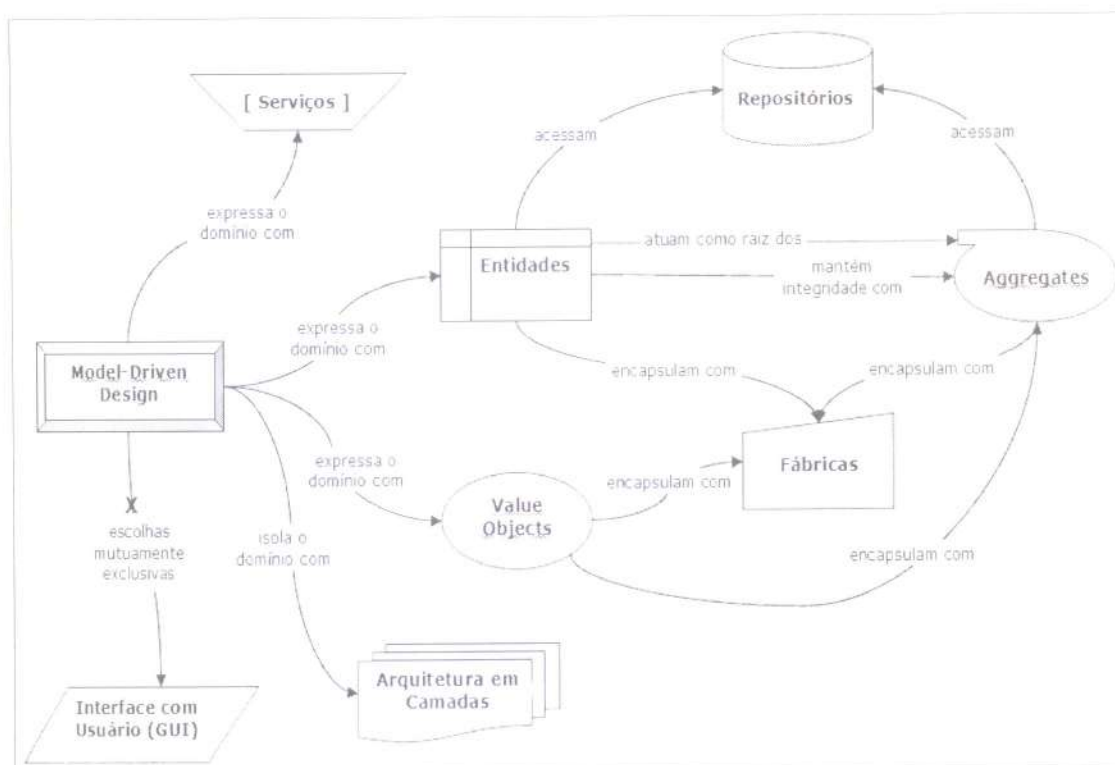


Figura 7 – Arquitetura do Model-Driven Design

Fonte: Adaptado de Avram (comp.); Marinescu (org.), (2006)

³ *Plain Old Java Objects*, ou seja, Objetos Java que não estendem nenhuma classe nem implementam interfaces

A Figura 8 expande o quadro da Arquitetura em Camadas, cuja função principal é isolar o domínio do restante da aplicação e definir claramente as responsabilidades de cada camada, de modo a diminuir o impacto de futuras modificações e diminuir o acoplamento do código (FOWLER, 2002).

Se o código relativo ao domínio está disperso entre diversas partes do programa, torna-se muito mais difícil entender o modelo do sistema, e mudanças superficiais na interface gráfica podem demandar grandes alterações na lógica de negócio ou no código de acesso ao banco de dados. As camadas são definidas por Evans (2003, p.74) da seguinte forma:

- **Interface com Usuário** (ou Camada de Apresentação): responsável por exibir as informações para o usuário e interpretar seus comandos, disparando as ações correspondentes. O ator externo pode ser também outro sistema computacional, ao invés de um usuário humano;
- **Camada de Aplicação**: define as tarefas que o software deve fazer e direciona as solicitações aos objetos de domínio responsáveis por atendê-las. Por ser uma camada de interligação, deve ser mantida concisa e leve, sem métodos de negócio, assim como não precisa guardar o estado de seus objetos;
- **Camada do Domínio** (ou Camada do Modelo): representa os conceitos, as informações e as regras do negócio. Aqui é onde se processam os estados dos objetos do sistema, ainda que as operações de salvar e recuperar esses estados sejam delegadas à camada de infra-estrutura. Segundo Evans (2003, p.74), “[...] esta camada é o coração de um sistema de software [...]”;
- **Camada de Infra-estrutura**: provê a base tecnológica para as camadas superiores, tais como envio de mensagens para a camada de Aplicação, persistência para o Domínio, desenho de controles para a Interface Gráfica entre outras.

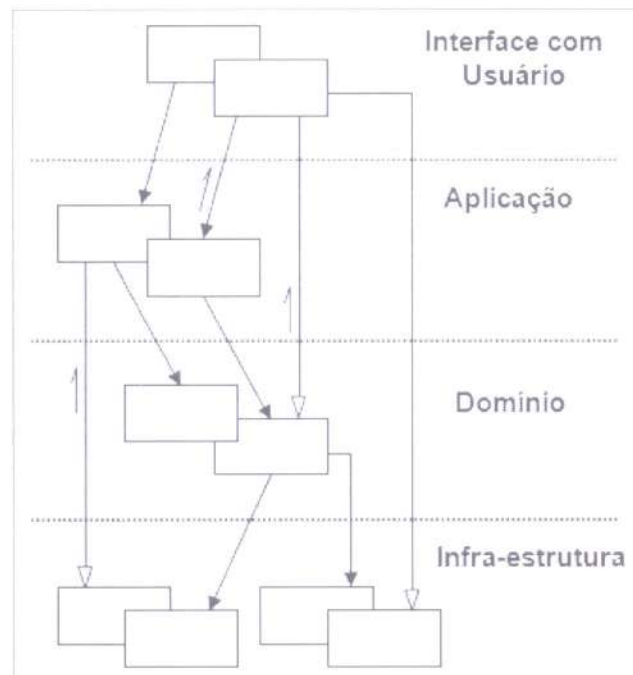


Figura 8 – Arquitetura em Camadas no DDD

Fonte: Evans (2003)

Alguns projetistas defendem a unificação entre a camada de Aplicação com a de Infra-estrutura, culminando no que se denomina o padrão de projeto MVC, descrito em (GAMMA et al, 2000). Porém, o isolamento explícito da camada de Domínio é o que realmente caracteriza o *Model-Driven Design*. Nesta linha de pensamento, Evans (2003, p.74) faz a seguinte recomendação:

Particione um programa complexo em camadas. Projete um design coeso, em que cada camada dependa somente das camadas inferiores. Siga os padrões de projeto arquiteturais a fim de prover um baixo acoplamento às camadas superiores. Concentre todo o código relativo ao domínio em uma camada e isole-o das demais. Os objetos do domínio, isentos da responsabilidade de se desenharem, armazenarem, gerenciarem as tarefas da aplicação e etc, podem se focar na implementação do modelo do domínio. Isto permite a um modelo evoluir e se tornar rico e claro o suficiente para capturar as informações essenciais de um negócio e colocá-las para funcionar.

Nos próximos subitens, serão explicados os demais componentes do *Model-Driven Design*, exemplificando alguns casos de uso.

2.2.1 Entidades

Entidades (*Entities*) fazem parte de uma categoria de objetos, cuja característica mais notória é possuir uma identidade própria, que os identificam unicamente entre os demais objetos derivados da mesma classe. Ou seja, dois objetos criados a partir da mesma classe (definição) não necessariamente serão idênticos. Para diferenciar esses objetos, apenas o conjunto de seus atributos não é suficiente. É necessário que exista “[...] *uma linha de continuidade e identidade, que atravessa o ciclo de vida de um sistema e pode até se estender além dele*” (AVRAM (comp.); MARINESCU (org.), 2006, p.39), para que se possa aplicar o conceito de Entidade a um determinado objeto.

Na Figura 5, *Cliente* é uma Entidade, pois este objeto possui identidade e continuidade ao longo do tempo e, além disso, seus atributos não o identificam exclusivamente. Diferentes clientes podem compartilhar o mesmo nome, endereço, idade e, mesmo assim, ser em diferentes no contexto da aplicação. O mesmo se aplica ao objeto *Pedido*, pois dois ou mais pedidos podem conter os mesmos itens, mas estarem associados a clientes distintos. Porém, a dificuldade nestes casos é justamente como distinguir cada objeto.

Considerando a Entidade *Cliente*, no nosso país é possível utilizar o número do Cadastro de Pessoa Física (CPF) para diferenciá-los; nos Estados Unidos, utiliza-se o número do Seguro Social (*Social Security*) e assim por diante. Contudo, quando não existem atributos que possam diferenciar os objetos, recomenda-se incluir um identificador único e externo à Entidade. Pode-se utilizar números inteiros auto-incrementados ou, em aplicações mais recentes, chaves alfanuméricas denominadas Identificadores Universais – *Universal IDentifiers (UID)*.

As Entidades representam um conceito importante no domínio da aplicação e, por isso, sugerimos cautela ao defini-las. Porém, nem sempre elas serão necessárias, e essa escolha deve ser feita tão breve quanto possível, de preferência logo no início do processo de modelagem.

2.2.2 Value Objects

Ao lado das Entidades, os *Value Objects*⁴ formam o cerne do MDD porém, ao contrário delas, não compartilham do conceito de identidade. Definindo-os de forma sucinta, "[...] esses objetos descrevem determinada característica de uma coisa" (EVANS, 2003, p.97). Justamente por esse motivo, são bem mais simples de se criar e manter, pois não precisam de uma monitoração e acompanhamento de seu estado ao longo do tempo. Essa peculiaridade lhes permite ser reutilizáveis e compartilhados entre diversos outros objetos.

Como exemplo, considerando um objeto *Pneu*, que é parte constituinte do objeto *Carro*. Os principais atributos de *Pneu* são o raio, largura da banda de rodagem e capacidade de carga (em kg). Um carro, para se locomover, necessita de quatro pneus preferencialmente de características idênticas, pelo menos aos pares (dois iguais nas rodas dianteiras e dois iguais nas traseiras). Caso algum dos pneus fure ou se estrague, basta substituir por outro com os mesmos atributos ou, ainda, retirar de um carro e colocar em outro com as mesmas características.

Analogamente, num *software* de computador, os objetos são criados livremente, utilizados pelo tempo necessário e destruídos pelo coletor de lixo (*garbage collector*). Quando solicitados por outros objetos ou funções, são repassados por valor, afinal o custo de criar um VO é relativamente baixo. E, como não precisam de um registro histórico das suas transições de estado, os VOs melhoram o desempenho do sistema como um todo. Além disso, *Value Objects* podem conter outros VOs, assim como referências a Entidades. O agrupamento entre VOs e Entidades será explicado mais a frente, em subitem posterior.

2.2.3 Serviços

Durante a análise de um modelo, surgem algumas características que se mostram difíceis de modelar como objetos, sejam eles Entidades ou VOs. Em geral, elas se apresentam como operações cujo efeito se aplica a dois ou mais objetos.

⁴ Não confundir com *Transfer Objects*, que são os objetos de transferência de dados na API *Java Beans* (ALUR; CRUPI; MALKS, 2001) e que, no início, eram também denominados *Value Objects*. Por não possuir uma tradução comumente aceita, foi mantida a versão original em inglês

Quando a ação diz respeito a apenas um objeto, ou pode ser realizada somente por ele, é simples incorporá-la ao seu comportamento. Ao contrário, se fosse incluída em um objeto qualquer ou distribuída entre vários outros, em diferentes camadas, acarretaria uma falha de projeto – acoplamento de código – gerando um *bad smell*.

Contudo, conforme observado em (AVRAM (comp.); MARINESCU (org.), 2006), numa linguagem OO, tais ações que não pertencem a um determinado objeto ainda assim precisam ser modeladas como objetos (ou interfaces, como sugere Evans (2003)). Essas interfaces seriam declaradas como Serviços (*Services*) no modelo do sistema, e devem constar também no vocabulário da Linguagem Ubíqua. Sua principal função é prover funcionalidades às Entidades e VOs. A seguir, são listadas as três principais características que definem um Serviço (AVRAM (comp.); MARINESCU (org.), 2006, p.46):

1. A funcionalidade fornecida por um Serviço se refere a um conceito do domínio que não pertence naturalmente a uma Entidade ou VO;
2. A operação realizada se refere a outros objetos no domínio;
3. A operação é *stateless*⁵.

Os Serviços são frequentemente encontrados nos *frameworks* (ferramentas de apoio) na camada de Infra-estrutura, mas podem fazer parte tanto das camadas de Domínio como também da Aplicação, como demonstrado na Figura 9.

2.2.4 Módulos

À medida que o domínio de uma aplicação cresce, seja pela complexidade natural do mesmo, ou por solicitações externas de clientes ou desenvolvedores, aumenta também a dificuldade em se entender as várias relações entre os objetos e suas funções dentro da aplicação. Para facilitar a compreensão do domínio, visto a partir de uma perspectiva macro, os projetistas organizam o modelo em módulos, cuja motivação principal é “[...] unir conceitos e tarefas relacionadas a fim de reduzir a complexidade do domínio” (AVRAM (comp.); MARINESCU (org.), 2006, p.48).

⁵ São operações onde não se mantém o estado dos objetos envolvidos, descartando a hipótese de persistência destes. A tradução literal de *stateless* é “sem estado”.

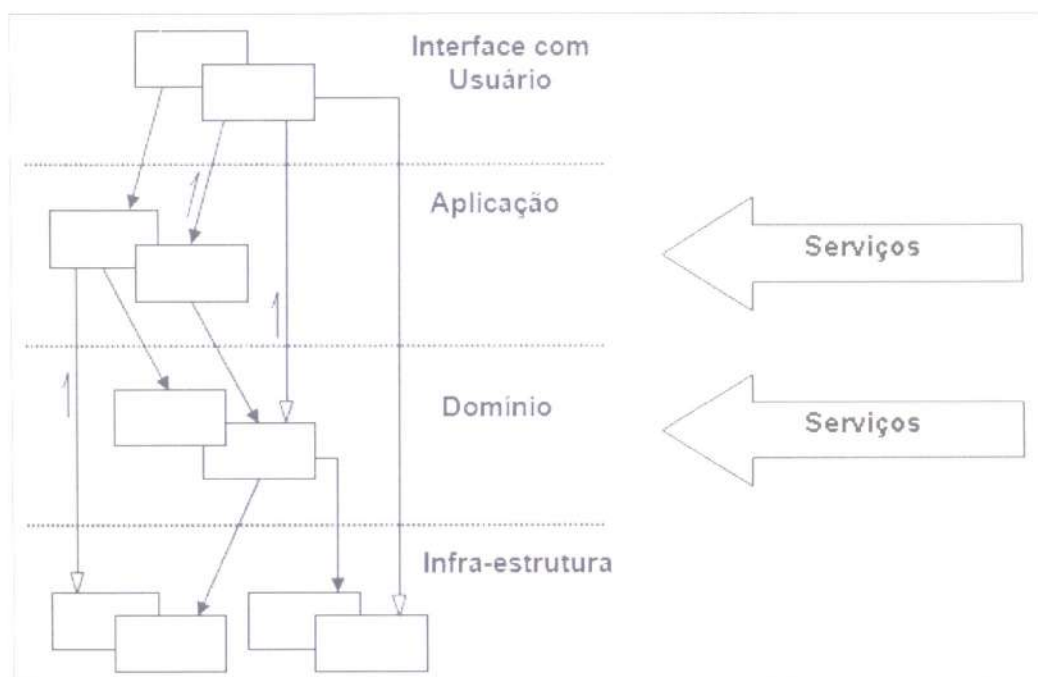


Figura 9 – Arquitetura em Camadas (com serviços)

Fonte: Adaptado de Evans (2003)

Como benefício adicional, a adoção de Módulos no modelo diminui o acoplamento entre os objetos, enquanto se aumenta a coesão dentro dos próprios módulos, o que gera código de melhor qualidade. Existem vários tipos de coesão, dos quais se destacam os dois principais:

- **Coesão Comunicativa** (*Communicational Cohesion*): ocorre quando partes do módulo atuam sobre os mesmos dados, executando diferentes operações;
- **Coesão Funcional** (*Functional Cohesion*): os objetos pertencentes ao módulo cooperam para realizar a mesma tarefa. Este tipo é considerado o mais coeso.

Como boa prática de projeto, os Módulos devem ter uma interface bem definida para que outros objetos possam acessá-lo. É mais simples acessar apenas uma interface, ao invés de vários objetos, pois as dependências entre os módulos serão menores, reduzindo assim o acoplamento. Acoplamento tênue reduz a complexidade do sistema, e facilita sua manutenção e compreensão por parte dos programadores que desconhecem sua estrutura.

Embora não apareçam no diagrama da Figura 7, os Módulos desempenham um papel importante no MDD, pois eles “[...] *contam a história do sistema e contêm um conjunto coeso de conceitos*” (EVANS, 2003, p. 108). Assim como os Serviços, devem constar no repertório da Linguagem Ubíqua. São compostos por Entidades, VOs e *Aggregates*, em qualquer proporção, e inclusive por Fábricas, sendo que estes dois últimos componentes possuem peculiaridades a serem discutidas mais adiante.

2.2.5 *Aggregates*

Numa primeira análise, os *Aggregates* (Agregados) se parecem bastante com os Módulos, explorados no subitem anterior. A própria definição destes componentes induz a essa semelhança. Segundo Evans (2003, p. 119):

Aggregate é um agrupamento de objetos associados que são tratados como uma unidade, no que se refere às alterações dos dados. Cada *Aggregate* possui uma raiz e um limite. O limite define o que está dentro do *Aggregate*. A raiz é o único membro do *Aggregate* ao qual os objetos externos podem guardar referências, embora objetos dentro do limite possam referenciar a si mesmos.

Apesar de compartilharem o mesmo princípio de encapsulamento das suas partes constituintes e do baixo acoplamento com objetos externos, os Módulos e *Aggregates* diferem nos objetivos. Enquanto os primeiros agrupam Entidades e VOs para cooperarem em tarefas ou dados semelhantes, os últimos são formados para garantir a integridade dos dados de seus objetos. Esta integridade, regida por regras de consistência denominadas de *invariantes*⁶, deve ser mantida após qualquer modificação nos dados – mudança de estado – dos objetos do *Aggregate*, eventualmente gerando um efeito cascata, necessitando de controle transacional.

⁶ Em outras palavras, invariantes são um conjunto de regras que os objetos devem obedecer. Este princípio é o que Meyer, citado por Calçado (2005), chama de *Design by Contract* (Projeto por Contrato), no qual os objetos “assinam” um contrato estabelecendo seus direitos (condições de entrada) e suas obrigações (condições de saída).

Uma maneira de simplificar o controle das alterações dentro do *Aggregate* é diminuir as associações muitos-para-muitos entre os objetos internos e, na medida do possível, criar associações unidirecionais. O custo de verificar todas as invariantes envolvidas no processo tende a impactar no desempenho geral do sistema e, se feito de forma incorreta, gera corrupção nos dados ou lixo nos bancos de dados, em especial registros órfãos.

Aggregates agrupam objetos do domínio que possuem associações de dependências, que podem se tornar complexas, e cujo acesso ao mundo exterior deve ser bloqueado. Apenas Entidades podem ser a raiz de um *Aggregate*, permitindo aos objetos externos acessá-las diretamente, enquanto elas mesmo têm direito de acesso a qualquer objeto interno (outras Entidades ou VOs) e fornecem referências temporárias destes objetos, quando necessário. Por isso, quando a Entidade raiz de um *Aggregate* é excluída no contexto da aplicação, todos os outros objetos pertencentes àquele conjunto também o serão, pois não existirá mais nenhum objeto com referências a eles. Os objetos internos do *Aggregate*, mesmo não oferecendo acesso direto a objetos externos ao limite, possuem a liberdade de referenciar Entidades-raiz de outros *Aggregates*.

2.2.6 Fábricas

Todas as linguagens OO modernas fornecem meios ao programador para iniciar o estado dos objetos, via métodos denominados construtores, que são os primeiros a serem executados após a alocação de memória para o objeto. Alguns objetos, seja por sua própria complexidade ou por dependerem de fatores externos a eles, demandam diversas tarefas para que sua rotina inicial de configuração seja concluída. Em alguns casos (AVRAM (comp.); MARINESCU (org.), 2006), o código para iniciar um objeto da camada de domínio pode se misturar a outro contido na camada de aplicação, violando o princípio de isolamento das camadas e aumentando o acoplamento.

Para encapsular o processo de criação desses objetos complexos, é necessário introduzir um novo elemento ao projeto: as Fábricas (*Factories*). As Fábricas não são um conceito novo criado pelo *Domain-Driven Design*, sendo

bastante discutidas por (GAMMA et al, 2000) e mencionadas em (FOWLER et al, 1999). Inclusive, Gamma et al (2000) descrevem um tipo ainda mais genérico, as *Abstract Factories* (Fábricas Abstratas), que seriam “fábricas de fábricas”, para objetos de mais alta complexidade.

No contexto do MDD, as Fábricas são responsáveis por iniciar corretamente as Entidades, VOs e especialmente úteis na criação dos *Aggregates*. Quando a raiz do *Aggregate* é criada, todos os objetos que dependem dela também o são, permitindo assim um maior controle sobre as invariantes (AVRAM (comp.); MARINESCU (org.), 2006), garantindo a integridade de estado para os objetos recém-criados. Sua operação deve ser atômica, ou seja, ou todos os objetos são criados de uma só vez, ou a execução será suspensa gerando uma exceção.

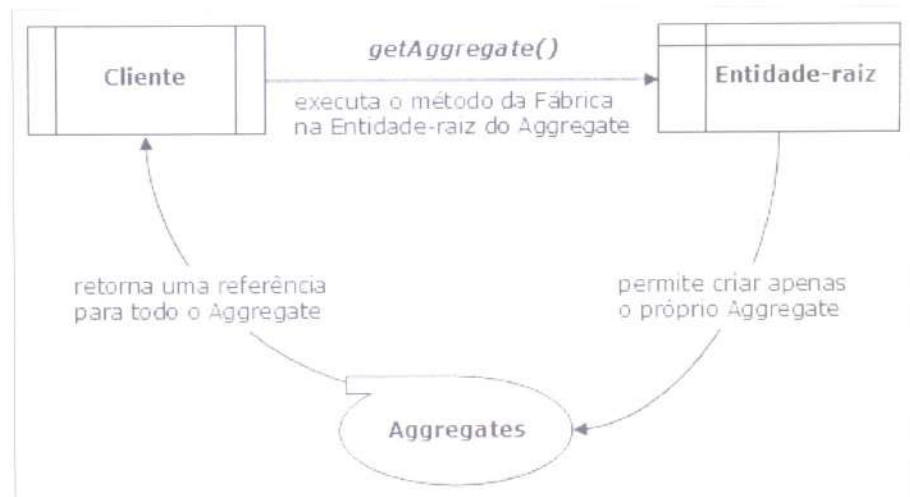


Figura 10 – Modelo de Fábrica (*Factory Method*)

Fonte: Dados da Pesquisa (2007)

Ainda que as Fábricas não façam parte do modelo do sistema, são ferramentas pertencentes à camada do domínio (EVANS, 2003) e auxiliam a reduzir a complexidade do projeto. Por exemplo, para a obtenção de um *Aggregate*, uma possibilidade seria criar um método na sua Entidade-raiz que controlasse o processo de criação dos objetos dependentes e retornasse uma referência da raiz, como observado na Figura 10. Caso se deseje desacoplar ainda mais o código, um objeto separado poderia ser adicionado ao domínio para assumir as responsabilidades da Fábrica, demonstrando na Figura 11.

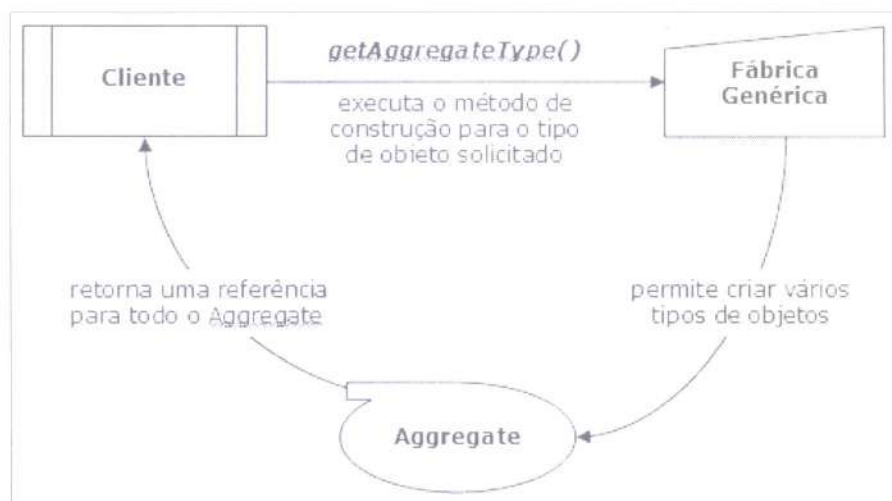


Figura 11 – Modelo de Fábrica Genérica

Fonte: Dados da Pesquisa (2007)

Além de criar objetos a partir do zero, as Fábricas talvez precisem recuperá-los de um estado dormente, provavelmente persistidos em um banco de dados. Este caso é um pouco diferente do primeiro, pois o novo objeto criado já possui uma identidade e cabe à Fábrica reconstruí-lo adequadamente, sem violar as invariantes (AVRAM (comp.); MARINESCU (org.), 2006). No caso de um objeto novo, e sem histórico anterior, a quebra de qualquer condição invariante implica no lançamento de uma exceção.

Mas, embora sejam de relevante utilidade, nem sempre é necessário o uso de Fábricas para iniciar os objetos, ressaltando o fato da perda de performance em utilizá-las. Evans (2003, p. 131) lista alguns casos em que os métodos construtores nativos são suficientes:

- A classe é o próprio tipo. Não é parte de nenhuma hierarquia relevante, e polimorficamente não é necessária para implementar uma interface;
- O objeto cliente se importa com a implementação interna, talvez uma melhor opção fosse usar o padrão *Strategy* (Estratégia) (GAMMA et al, 2000);
- Todos os atributos do objeto estão disponíveis ao cliente;
- A construção do objeto não é complicada;
- Um construtor público deve seguir as mesmas regras de uma Fábrica: operação atômica e invariantes preservadas.

2.2.7 Repositórios

Em ambientes não orientados a objetos, a simples declaração de uma variável já a torna pronta para uso, podendo ser utilizada como retorno de funções ou resultado de operações matemáticas. Mas, esta declaração apenas cria um ponteiro que armazenará um endereço de memória da máquina. Caso ocorra uma tentativa de acesso à variável antes de lhe atribuir um endereço de memória válido, ocorreria um erro de Ponteiro Nulo (*Null Pointer*). Nos sistemas OO, os ponteiros são chamados de referências e, da mesma forma, apontam para um endereço na área de memória. Para que se possa efetivamente trabalhar com um objeto, é necessário obter uma referência válida para ele, ou seja, um ponteiro não-nulo.

A primeira forma de se obter referência para um objeto é, simplesmente, criando um novo. Esta operação, que pode ser realizada por uma Fábrica, retornará a referência para o objeto desejado. Outra maneira de se conseguir uma referência válida é percorrendo uma associação (AVRAM (comp.); MARINESCU (org.), 2006), onde se começa por um objeto conhecido e solicita-lhe que repasse a referência para um dos objetos sob sua dependência. Estas associações remetem diretamente aos *Aggregates*, e suas Entidades-raiz, que são objetos públicos e aos quais se pode acessar diretamente. Mas, ainda assim, é necessário conhecer a raiz.

Porém, a situação mais freqüente é mesmo a recuperação de objetos que foram persistidos em banco de dados, ou outra forma de armazenamento permanente. Um objeto, após ser utilizado durante o tempo necessário, poderá ser excluído definitivamente ou armazenado para posterior reutilização. Mesmo aqueles que não possuem identidade própria – VOs, por exemplo – necessitam de registro histórico.

Portanto, apresenta-se bastante tentadora a idéia de obter diretamente do banco de dados as referências para os novos objetos. Essa opção não se torna atraente quando se considera a quantidade de código relativo à infra-estrutura de acesso ao banco que ficará dispersa nas classes de domínio, quebrando o isolamento das camadas proposto por (EVANS, 2003). Ainda assim, há quem advogue o uso do padrão de projeto *Table Data Gateway* (FOWLER, 2002), também conhecido como *Data Access Objects (DAO)* (ALUR; CRUPI; MARKS, 2001).

A abordagem do padrão DAO – ou do *Table Data Gateway* – realmente preserva o isolamento das camadas, agindo em conjunto com o último elemento do MDD discutido neste capítulo: o Repositório, cuja função é encapsular a lógica necessária para obter referências (EVANS, 2003), que se estende além do simples Mapeamento Objeto-Relacional (*Object-Relational Mapping* – **ORM**). Isto porque o Repositório pode também salvar referências para alguns dos objetos que, após serem criados, são salvos no próprio Repositório e recuperados de lá mesmo, como visto no fluxograma da Figura 12 (AVRAM (comp.); MARINESCU (org.), 2006).

É público que qualquer operação envolvendo acesso a disco, ou mesmo à rede, gera um atraso maior na execução do programa, pois a CPU precisa esperar a transferência dos dados para continuar o seu processamento. Essa estratégia de *cache*⁷ dos objetos recém-criados agiliza bastante o desempenho do sistema e, na eventualidade deles não estarem disponíveis em memória quando forem solicitados pelo cliente, ainda assim poderão facilmente ser recuperados do banco.

Alguns sistemas de ORM (BAUER; KING, 2006) implementam ainda o cache dos dados oriundos do próprio banco de dados, aumentando ainda mais a performance da aplicação. Ainda assim, é importante ressaltar que um Repositório não é um ORM, mas pode se utilizar deste para realizar suas funções.

2.3 CONSIDERAÇÕES FINAIS

Ao longo deste capítulo, foram abordados de maneira sucinta os principais conceitos do *Domain-Driven Design*, assim como a importância da interação contínua entre os clientes e os desenvolvedores, de modo a agilizar as correções de rumo necessárias ao projeto. Uma das técnicas da Programação Extrema (*eXtreme Programming* – **XP**) recomenda justamente isso: reuniões frequentes entre clientes e programadores, além da criação de protótipos funcionais do sistema logo nas primeiras iterações (TELES, 2007).

⁷ É uma técnica que consiste em armazenar na memória volátil (RAM) do computador os dados acessados mais recentemente. Na próxima requisição a esses dados, o programa irá acessá-los diretamente da memória RAM, que é muito mais veloz do que a memória permanente (Disco Rígido – HD), tanto para leitura quanto para escrita.

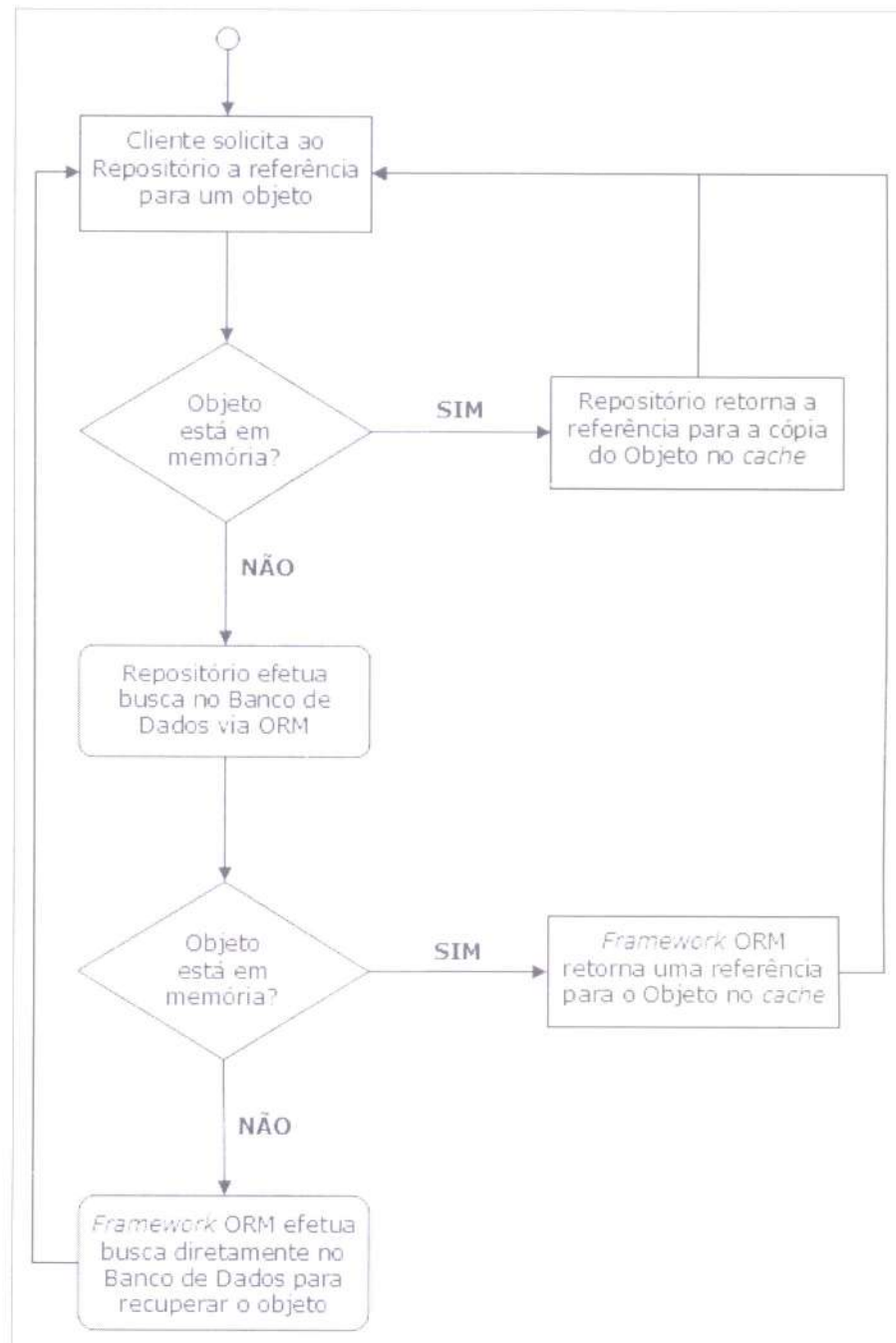


Figura 12 – Fluxograma de um Repositório

Fonte: Dados da Pesquisa (2008)

Na mesma linha, foi citada a necessidade de um vocabulário comum entre os participantes do projeto. A criação e evolução da Linguagem Ubíqua devem-se dar naturalmente, respeitando as peculiaridades do domínio da aplicação. Sempre que possível, utilizar os termos e verbos empregados pelos especialistas no domínio. Dependendo da complexidade, sempre é possível simplificar a modelagem abstraindo o funcionamento dos elementos mais complicados. Estes elementos seriam agrupados em *Aggregates*, encapsulando sua funcionalidade em VOs mais simples, fornecendo apenas um ponto de acesso, por meio de uma Entidade.

As Entidades, por sua própria característica de possuir identidade própria, em geral são persistidas no banco de dados. Enquanto isso, os objetos que não possuem identidade são implementados por *Value Objects*, os quais na maioria das vezes não necessitam de persistência. Fowler (2002) ressalta um aspecto interessante sobre os VOs: seus métodos *hash*⁸ devem ser sobrescritos, pois dois VOs com atributos do domínio iguais são considerados idênticos, para fins de aplicação. Por exemplo, um objeto do tipo Dinheiro é identificado unicamente por seus atributos valor e moeda, que indicam respectivamente a quantidade numérica de dinheiro representada e sua correspondente moeda (Euro, Dólar, Real etc) no mercado financeiro internacional. Esta característica faz com que os VOs possam ser reutilizados dentro do sistema, evitando a duplicação desnecessária dos mesmos.

Como regra geral, se os VOs serão compartilhados durante o seu ciclo de vida na aplicação, eles devem ser imutáveis. Se um VO é solicitado por outro objeto, ele pode ser enviado por valor, ou uma cópia sua seria criada e repassada (AVRAM (comp.); MARINESCU (org.), 2006). O mesmo não ocorre com as Entidades que, quando solicitadas por outros elementos do domínio, devem ser repassadas por referência, uma vez que a duplicação de Entidades com a mesma identidade pode ocasionar operações inválidas ou a sobreposição indevida de dados.

⁸ São funções criptográficas que recebem uma entrada de tamanho aleatório e a transformam num código de *bits* com tamanho fixo, o chamado *hash*, cuja tradução é picar ou comprimir. A transformação é de mão-única: dois códigos de entrada distintos sempre irão gerar saídas diferentes. Todo objeto Java possui uma função de *hash* nativa e que pode ser sobrescrita, o que possibilita identificar dois objetos com conteúdos idênticos usando essa função, ao contrário do método *equals* que apenas verifica se dois objetos apontam para a mesma referência.

Prosseguindo na exposição do *Model-Driven Design*, dois componentes se destacam: os Módulos e os Serviços. Enquanto os Módulos servem como auxílio no entendimento do modelo, na medida em que os projetistas agrupam logicamente determinadas partes, os Serviços facilitam a manutenção e evolução do sistema, pois retiram dos objetos de domínio algumas responsabilidades que pertenceriam às demais camadas. Talvez se torne difícil identificar quais serviços são mais propensos à serem incluídos na camada de aplicação ou na de domínio, mas Evans (2003, p. 104) detalha algumas situações que ajudam na escolha.

Tanto os Módulos como os Serviços se beneficiam das facilidades de empacotamento das linguagens OO modernas. O agrupamento físico em pacotes – *Packages* em Java ou *Assemblies* na C# – facilita a manutenção do código, e o agrupamento lógico auxilia na evolução e refatoração (FOWLER et al, 1999), por diminuir o acoplamento e aumentar a coesão (LARMAN, 1998) dentro dos Módulos ou Serviços.

Concluindo os elementos responsáveis diretamente pela expressão do modelo, na sequência foram abordados aqueles que lidam com o ciclo de vida dos objetos: as Fábricas e Repositórios. Descrevendo-as resumidamente, as Fábricas criam novos objetos em memória, e os Repositórios se encarregam de armazená-los e recuperá-los posteriormente. Porém, existe uma pequena diferença entre as Fábricas de Entidades e as de *Value Objects*, discutida com mais detalhes por Evans (2003, p. 133). Basicamente, diz respeito à inserção da identidade nas Entidades, visto que os VOs não a possuem. No caso de um objeto reconstruído, essa identidade deve ser preservada, e não gerada novamente.

Quanto aos Repositórios, observa-se que eles auxiliam no isolamento das camadas e permitem ao desenvolvedor abstrair o meio de persistência dos objetos, utilizando *frameworks* de ORM. Ganhos de performance podem ser obtidos com o uso de *cache* para o sistema de ORM, e para o próprio Repositório. Por fim, este último padrão permite ainda o funcionamento conjunto com uma Fábrica, diante da necessidade em recuperar objetos mais complexos, que seriam restaurados preservando as invariantes, como visto na Figura 13. E, no próximo capítulo, será desenvolvido um estudo de caso utilizando os conceitos estudados até o momento para, em seguida, finalizar o trabalho com as conclusões.

3 ESTUDO DE CASO

Em geral, todo projeto de informatização de um sistema segue uma seqüência de fases pré-definidas (AMBLER, 2005): Iniciação (*Inception*), Elaboração (*Elaboration*), Construção (*Construction*) e Transição (*Transition*), que visam alcançar um mínimo de qualidade e segurança ao processo. Estas etapas fazem parte do processo de desenvolvimento RUP, mas podem ser encontradas em diversas outras metodologias de projeto (Espiral, Cascata, XP etc), embora venham a utilizar outras denominações ou apareçam de forma iterativa (em laço). Como pode ser deduzido do diagrama demonstrado na Figura 14, a tarefa de Modelagem do Negócio (*Business Modeling*) está praticamente concentrada apenas nas fases de Iniciação e Elaboração. Assim, como o foco deste trabalho é a modelagem do domínio, serão priorizadas as duas primeiras etapas, em detrimento da Construção e Transição.

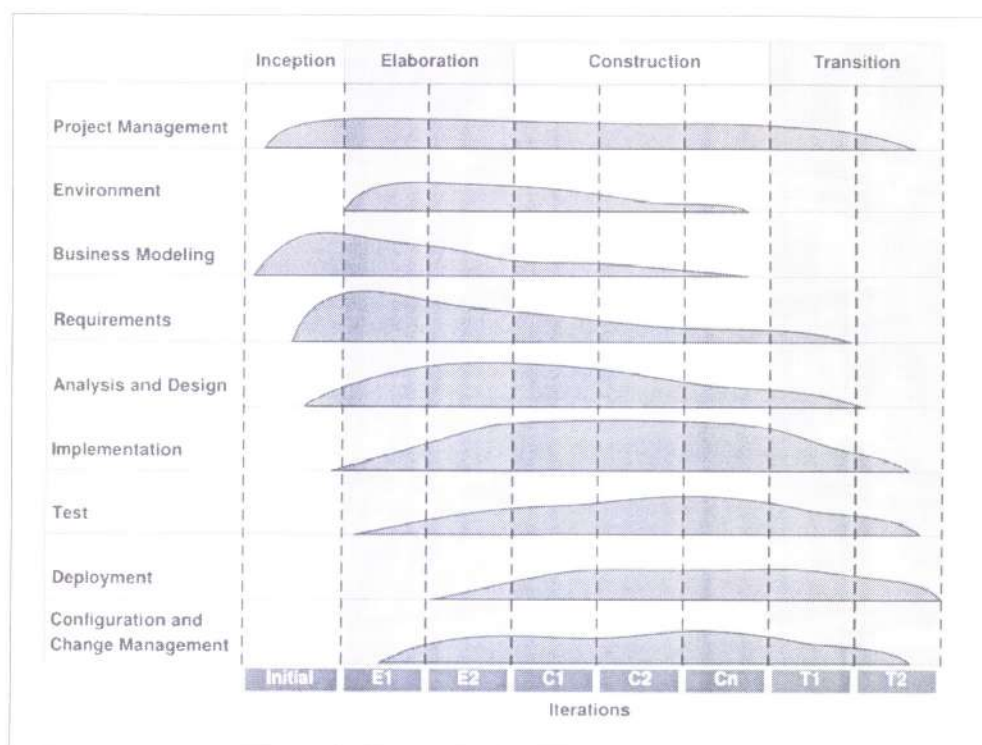


Figura 14 – Etapas e Disciplinas do RUP

Fonte: Wikipedia (2007)

Neste capítulo será desenvolvido um estudo de caso sobre a modelagem de um sistema computacional, utilizando os conceitos do *Domain-Driven Design*. O foco do projeto é apenas a discussão do domínio do sistema, não levando em conta a codificação e implementação de um programa funcional, pois para estas tarefas já existem ótimas referências: (GAMMA et al, 2000); (FOWLER et al, 1999); (FOWLER, 2002). Além disso, encontra-se disponível uma extensa variedade de *frameworks* – a maioria em código-livre – que fornecem as ferramentas e objetos para a resolução dos problemas mais corriqueiros de programação.

O objetivo principal é construir um modelo que seja fiel ao domínio, contando com as vantagens intrínsecas do DDD – facilidade de manutenção e evolução. Por esse motivo, a camada de domínio será mais explorada, em detrimento da aplicação, IU e infra-estrutura. Os componentes principais do DDD serão usados para descrever o modelo e representar o domínio de forma simplificada.

3.1 APRESENTAÇÃO DO DOMÍNIO

O sistema escolhido para o estudo de caso é uma Concessionária de Automóveis, cuja finalidade comercial é vender e efetuar manutenção nos veículos produzidos pela Fábrica que ela representa. Resumidamente, a Concessionária recebe o pedido de um veículo novo e repassa a solicitação para a Fábrica. Esta, por sua vez, produz o veículo com as características e opcionais solicitados e o envia para a Concessionária, a qual emite uma fatura ao cliente para que este realize o pagamento do veículo. No caso de uma manutenção, a Concessionária recebe o automóvel do cliente, aplica as correções necessárias e, da mesma forma, emite fatura com a discriminação das peças utilizadas e da mão-de-obra necessária.

A evolução do processo de modelagem será mostrada seguindo o método aplicado em (PRESSMAN, 2006). Serão apresentados personagens e diálogos que, na medida do possível, tentam simular as situações mais freqüentes que ocorrem durante o projeto e desenvolvimento de um software. Este método permite que se demonstrem, no decorrer do processo, as vantagens obtidas ao projetar usando o DDD. As explicações mais detalhadas acerca do domínio serão introduzidas paulatinamente durante os diálogos dos personagens.

Porém, antes de continuar com a descrição dos personagens, é importante avisar que o Autor não recebeu nenhum tipo de financiamento ou patrocínio para citar as Marcas Registradas ou Tecnologias expostas nos diálogos, sendo fruto apenas de suas preferências pessoais ou meramente acaso. Os personagens e suas respectivas opiniões e idéias são meramente fictícios e não remetem à qualquer empresa ou produto no mercado.

3.1.1 *Personagens*

Para que houvesse uma diversidade de idéias e pontos de vista, optou-se por escrever os diálogos através de quatro personagens, cujas características são as seguintes:

- **Gerente da Concessionária (GC):** é o responsável pela gerência dos recursos humanos e financeiros da concessionária, possuindo conhecimentos na área administrativa;
- **Supervisor de Tecnologia da Informação (STI):** coordena a aquisição e implantação dos equipamentos e serviços de informática para a concessionária. Possui formação técnica na área de Tecnologia da Informação (TI);
- **Analista de Sistemas (AS):** coordenador técnico da empresa terceirizada, contratada para automatizar os processos de venda e manutenção da concessionária. Possui formação superior na área de computação;
- **Técnico em Programação (TP):** chefe da equipe empenhada na automação da concessionária. Dotado de formação técnica e diversos certificados de qualidade na área de programação de sistemas.

Conforme explicado acima, os dois primeiros personagens são os responsáveis técnicos da concessionária, e os principais contatos para os demais que representam a empresa terceirizada, contratada para desenvolver o software de controle de vendas e manutenção. Durante os diálogos, cada participante será identificado por suas iniciais, a fim de resultar numa melhor fluidez do texto.

3.2 INICIAÇÃO

Esta é a primeira etapa do projeto, e existe em todas as metodologias. É neste ponto que ocorre o primeiro contato entre o cliente e o desenvolvedor. Os subitens a seguir iniciam o processo.

3.2.1 Reunião Inicial

- **Local:** sala de reunião da empresa terceirizada;
- **Participantes:** gerente da concessionária e analista de sistemas;

AS: – Bom dia, GC, em que posso ajudá-lo?

GC: – Bom dia, AS, estou interessado em melhorar meu sistema de atendimento na concessionária.

AS: – Em que sentido?

GC: – Atualmente, nós contamos apenas com um programa que se conecta ao sistema da fábrica e nos passa a situação do estoque. Então, nós abrimos um pedido para o automóvel que o cliente deseja, e o próprio sistema já abre a solicitação na fábrica e fornece um prazo para a entrega.

AS: – E, pra vocês, não está sendo suficiente?

GC: – Bem, o software já é arcaico, foi enviado pela Fábrica para nossa revenda há muitos anos atrás. E, hoje em dia, a Fábrica até tem uma versão mais atual, que roda na Internet, mas não possui uma opção para controlar as manutenções, os serviços, entende?

AS: – Entendo. Quer dizer que você necessita de um programa que se comunique com o sistema da Fábrica, e ainda forneça um controle sobre os serviços realizados pela Concessionária?

GC: – Exato! Quando podemos começar?

AS: – A princípio, preciso visitar a sua empresa e conhecer a estrutura atual. Existe alguém cuidando da área de informática?

GC: – Sim, temos um Supervisor encarregado da parte técnica.

AS: – Ótimo! Gostaria que ele estivesse presente na nossa próxima reunião.

GC: – Tudo bem, irei agendar com minha secretária.

AS: – OK, até mais.

3.2.2 Coleta de Dados

- **Local:** sala de reunião da concessionária;
- **Participantes:** gerente da concessionária, analista de sistemas e supervisor de TI;

GC: – Bom dia, senhores! Vamos dar início à reunião, conforme planejado. AS, este é o Supervisor de TI (STI), nosso responsável pelo bom funcionamento dos sistemas de informática da concessionária.

AS: – Bom dia, GC! Bom dia, STI! Podemos começar os trabalhos?

STI: – Então, AS, em que posso te ajudar?

AS: – Você pode me explicar, de forma geral, como funciona o seu software de vendas, e o que vocês esperam do novo sistema integrado de venda e manutenção de veículos...

STI: – OK, vamos partir do início. Não sei se o chefe já te explicou, mas o sistema atual foi fornecido pela Fábrica e roda em ambiente DOS, com base de dados dBase e acessa o servidor da Fábrica por um *link* dedicado de 256 kbps (kilobits por segundo).

AS: – Bem, ele me falou apenas o básico. Pode continuar.

STI: – Certo. Prosseguindo: o sistema sincroniza com a Fábrica diariamente à noite, para baixar as atualizações de preços e modelos de veículos. Porém, quando abrimos uma ordem de venda, a solicitação é enviada imediatamente para o servidor da Fábrica, que nos retorna um número de pedido e o prazo estimado para envio do carro.

AS: – Você sabe qual o protocolo utilizado na comunicação com o servidor?

STI: – AS, da última vez que eu conversei com o administrador de redes da Fábrica, ele me falou que era via conexão direta por *socket*. Inclusive,

isso o aborrecia muito pois ele precisava deixar uma porta aberta no *firewall* para que o programa acesse o servidor, causando um furo de segurança na rede.

AS: – Entendi. Mas, você tem as especificações do protocolo, com a porta e o formato das mensagens utilizadas?

STI: – Não vai ser preciso. Na versão mais nova, o pessoal configurou tudo com *WebServices* na porta 80 padrão do http. E vão até disponibilizar um manual de integração para as concessionárias que quiserem desenvolver o próprio sistema, como é o nosso caso.

AS: – Perfeito, isso irá facilitar muito mesmo a integração. Mas, o sistema antigo continuará a funcionar?

STI: – Não. Até mesmo a porta antiga será fechada no *firewall* e apenas o sistema novo irá acessar o servidor. É por isso que a Fábrica desenvolveu uma nova versão do programa. Quem não atualizar, ou não fizer uma versão própria, não conseguirá mais fazer pedidos pela Internet.

AS: – Certo, até aqui tudo bem. Vocês me falaram sobre a questão das vendas e a arquitetura de cliente-servidor do programa atual. Mas, em relação à manutenção, como é feito o controle?

GC: – Acho que chegou o momento de eu me intrometer... A manutenção é realizada pela concessionária, sem intervenção da Fábrica, exceto na primeira em que as peças são gratuitas, mas é cobrada a mão-de-obra. O *software* de controle é bem simples, e foi encomendado a um programador autônomo.

STI: – Se você precisar, AS, temos o código-fonte do programa. Ele foi feito em linguagem Delphi com base de dados Paradox. Que, por sinal, adora se corromper nas piores horas e temos que parar todas as máquinas para fazer uma limpeza no BD.

AS: – Quanto a isso, não há motivo para se preocuparem. Nossa proposta engloba a instalação de um servidor de banco de dados com fonte de energia redundante. A chance de corromper o BD é próxima de zero. Só na hipótese de ocorrer falha em disco ou mesmo da máquina.

GC: – Isto me deixa realmente mais tranquilo. É sempre uma dor de cabeça

ver nossos empregados parados, e os clientes esperando para serem atendidos porque o sistema travou.

AS: – Voltando ao assunto da manutenção, eu tenho mais uma pergunta: o *software* de manutenção também acessa o servidor da Fábrica?

GC: – Não. Basicamente, o que ele faz é um controle de estoque (entrada e saída) das peças que recebemos da Fábrica, e gera as faturas com o custo da mão-de-obra dos serviços. Bem simples e direto, pois não precisamos de mais do que isso.

STI: – Tem mais uma observação, GC. O cadastro de clientes e tipos de veículos não é compartilhado, obrigando os vendedores a registrarem os clientes duas vezes: uma quando vendem o carro, e outra quando eles voltam para a revisão gratuita.

GC: – Obrigado, STI. Foi bom você mencionar isso. É possível unir os dois cadastros, AS?

AS: – Claro que sim, GC, e com certeza será uma de nossas primeiras providências. O projeto já irá contemplar essa junção...

– Eu acredito que já temos o suficiente por hoje. É possível agendar a próxima reunião para daqui a dois dias?

GC: – Claro. Eu liberarei o STI para lhes auxiliarem com o que for preciso.

AS: – Irei precisar do senhor também, GC. Pelo menos, nas reuniões iniciais.

GC: – Não sei se poderei ajudar muito, pois sou leigo nesses assuntos técnicos...

AS: – Mas o senhor é um especialista na sua área gerencial e conhece muito bem os processos da sua empresa. Nossa metodologia de trabalho prevê integração total com o cliente e reuniões freqüentes, para que possamos corrigir os rumos do projeto o mais breve possível, evitando divergências futuras. Sem falar que vocês terão contato com versões de teste do programa logo nas primeiras semanas.

GC: – Tudo bem, marcarei com a secretária. Já que é tão necessário assim, irei participar também.

AS: – Vejo os senhores em dois dias. Boa tarde e até logo!

3.3 ELABORAÇÃO

Decorrida a fase de Iniciação, na qual ocorre o primeiro contato entre o cliente e o desenvolvedor, a próxima fase é a Elaboração. Nesta etapa, o processo de modelagem – iniciado na fase anterior – torna-se acelerado, visando construir um modelo para o sistema, que será implementado na próxima fase. No próximo subitem, será discutida a elaboração da Linguagem Ubíqua.

3.3.1 Criação da Linguagem Ubíqua

- **Local:** sala de reunião da empresa terceirizada;
- **Participantes:** gerente da concessionária, supervisor de TI, analista de sistemas e técnico em programação;

AS: – Boa tarde! Quero lhes apresentar o Técnico em Programação (TP), nosso programador-sênior. Ele irá coordenar diretamente os demais programadores no desenvolvimento do software encomendado pelos senhores.

TP: – Boa tarde a todos. O AS me repassou as principais informações acerca da Concessionária e das funções solicitadas. Nós chegamos a um primeiro esboço, que está ilustrado na Figura 15.

GC: – Humm, muito interessante... Pensei que vocês iriam me mostrar um monte de códigos indecifráveis e fluxogramas confusos com várias páginas de extensão. É mais ou menos isso que nós queremos.

TP: – O senhor talvez esteja confundindo com o código-fonte de uma linguagem de programação. Não se preocupe, nós não iremos te incomodar com códigos ou discussões técnicas. Isto é para eu e o AS quebrarmos a cabeça.

STI: – OK, mas em que especificamente poderemos ajudar?

AS: – Nós queremos elaborar um dicionário comum de termos, que nos auxilie nas futuras discussões sobre o projeto. Costumamos chamar esse “dicionário” de Linguagem Ubíqua.

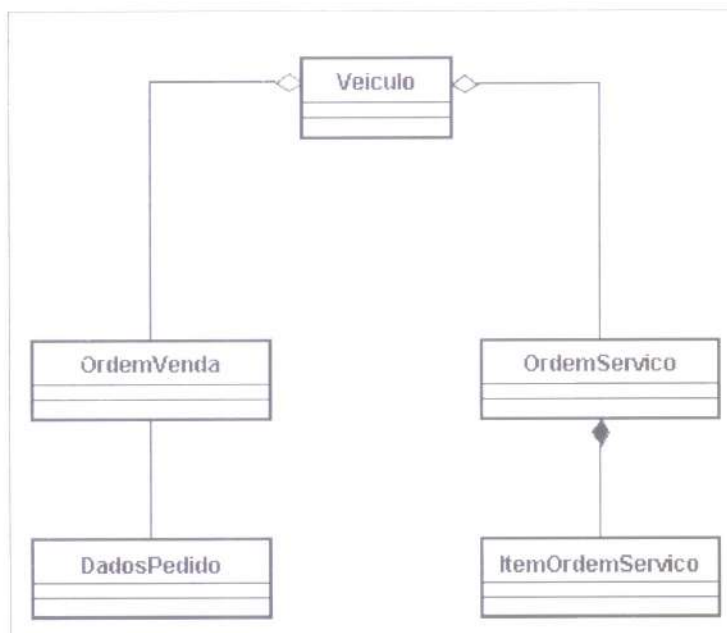


Figura 15 – Modelo da Concessionária (Revisão 1)

Fonte: Dados da Pesquisa (2008)

GC: – Que nome estranho, nunca tinha ouvido falar.

STI: – É, chefe, estranho mesmo. Mas quer dizer “Onipresente”. Descobri isso quando lia uma revista com as tendências do futuro em informática. Um dos autores comentava sobre Computação Ubíqua, que é aquela que está em todo lugar e nós nem nos damos conta, pois foi feita para ser simples e não causar impacto. Como a tecnologia embarcada em nossos automóveis: ela está lá e nem percebemos, tipo a injeção eletrônica, o controle de tração etc.

AS: – Isso, STI. Foi uma boa explicação. Tenho apenas uma observação: no nosso caso, a Linguagem Ubíqua é apenas um formalismo para descrever os verbos e substantivos que vocês utilizam no cotidiano do seu negócio. Ela também tem que ser simples, e estar em todos os lugares, ou seja, no nosso modelo (ou projeto) e no seu ambiente de trabalho.

TP: – Eu gostaria de pedir aos senhores que nos listassem, com suas palavras, as principais tarefas com que vocês lidam na concessionária.

GC: – Certo. Vou começar por mim, falando mais da parte administrativa do negócio. O STI comentará sobre os aspectos técnicos em seguida.

– Bom, o primeiro passo diz respeito à venda. Nossos vendedores exibem o catálogo de produtos, no caso os nossos automóveis, aos futuros clientes. Eles, por sua vez, após escolherem o modelo, podem optar por diversos pacotes de opcionais, além da cor e se a mesma será sólida ou metálica.

AS: – Muito bem, fique à vontade para continuar.

GC: – Quando o cliente já possui todos os dados do veículo que deseja adquirir, o vendedor faz o pedido no sistema, que abre uma ordem de venda e retorna um número do pedido. É com esse número que fazemos o acompanhamento do pedido. Ao receber o veículo, entramos em contato com o cliente para que ele venha buscá-lo na loja. O próprio sistema gera uma fatura que o cliente irá pagar ou, no caso de um financiamento, a financeira é quem faz o pagamento e transfere o débito ao cliente.

TP: – Está muito bom até agora. Fale-nos da manutenção dos veículos.

GC: – Como já tinha dito ao AS, apenas a primeira revisão é gratuita, mas somente no caso de um automóvel zero km. Sim, a troca de peças que é gratuita, a mão-de-obra será cobrada de acordo com o serviço. Em qualquer outra manutenção, nós cobramos o valor das peças e do serviço. O sistema que controla isso tudo é separado do que controla as vendas, mas também emite fatura automaticamente, incluindo os impostos de peças (ICMS) e o de serviço (ISS).

AS: – STI, você tem alguma observação a acrescentar?

STI: – Quanto à explicação do GC, não, é exatamente assim que funciona nosso negócio, ele conhece até melhor do que eu. Mas, em relação à infra-estrutura, eu tenho algumas idéias a propor.

AS: – TP, por favor, anote as sugestões do STI. Os requisitos funcionais foram quase todos abordados. Vamos ver agora os não-funcionais.

TP: – Pode deixar, AS. Estou ouvindo, STI...

STI: – Vou repetir uma questão que comentei anteriormente, mas o TP não estava presente. É sobre o cadastro dos veículos e dos clientes, que não é compartilhado entre os sistemas atuais.

- TP:** – Sim, com certeza iremos integrar os dois sistemas em uma aplicação única, sem duplicação desnecessária de dados.
- STI:** – As próximas questões são sobre a permissão de acesso aos usuários e ao sistema de pagamento. Nós firmamos convênio com algumas operadoras de cartão de crédito e débito, que autorizam via rede todas as transações.
- AS:** – Foi bom você mencionar esses tópicos. No nosso vocabulário comum eles serão conhecidos como *Serviços* auxiliares. Já possuímos um módulo que realiza a autenticação do usuário no sistema e que pode ser utilizado em qualquer aplicação nossa. Ele é independente de banco de dados e arquitetura.
- STI:** – Ótimo, perfeito! Chefe, estou ficando muito satisfeito com os colegas dessa empresa terceirizada. Os caras entendem mesmo e possuem experiência de fato no mercado.
- GC:** – Não é à toa, STI, eles me foram indicados por um grande amigo e empresário do ramo imobiliário. Foram vocês que criaram aquele site que mostra as ofertas de imóveis pra alugar e vender, não foi?
- TP:** – Sim, inclusive fui eu quem projetou e codificou o portal deles na Internet. Você pode escolher receber por e-mail quaisquer novas ofertas de imóveis que se encaixem nas regras que você criar: pode ser por área construída, número de quartos e até por bairro.
- AS:** – É verdade, o portal Immobile foi muito elogiado pelos clientes e pela mídia especializada. Foi nosso primeiro grande trabalho! Mas, vamos retomar o foco e recapitular, pois a reunião já se aproxima dos quarenta minutos de duração. Nós pregamos reuniões curtas e objetivas, às vezes até mesmo em pé pra não dar tempo de ninguém divagar muito.
- GC:** – Sabe, gostei dessa idéia das reuniões em pé, acho que vou aproveitá-la por aqui na concessionária. Eu particularmente detesto perder muito tempo com conversas esparsas, o que geralmente ocorre quando se junta muita gente.

TP: – Pois é, GC, essa é uma das práticas da metodologia de desenvolvimento XP que nós adotamos na empresa. Entre outras, utilizamos com frequência a programação em pares, ou seja, dois programadores dividindo o teclado e mouse no mesmo computador.

STI: – Já ouvi alguns amigos comentarem sobre isso. Mas, falando sério, dá mesmo resultado?

TP: – Por incrível que pareça, o código escrito a quatro mãos tem menos defeitos e acaba saindo mais rápido que se apenas um estivesse trabalhando. Até porque cada programador revisa o código do colega enquanto este digita.

AS: – Sem dúvida, senhores. Vários casos de sucesso demonstram a viabilidade dessa técnica.

– Finalmente, voltando à nossa discussão, e de acordo com as minhas anotações, vou resumir agora os principais termos com que iremos trabalhar:

- **Cliente** (objeto);
- **Ordem de Venda** (objeto):
 - **Veículo** (objeto);
- **Ordem de Serviço** (objeto):
 - **Peça** (objeto);
 - **Serviço** (objeto);
- **Pagamento** (serviço).

GC: – Espere um minuto! Não entendi: como é que um Serviço pode ser um objeto e Pagamento é um serviço?

AS: – Na realidade, GC, é preciso diferenciar os conceitos para que vocês não se confundam. O Serviço que é um objeto equivale ao tipo de manutenção realizada no veículo e, portanto, dentro do programa ele será identificado como objeto (*Value Object*). Já o Pagamento que nós classificamos como serviço é, também, um objeto, mas que no contexto do sistema fornece a funcionalidade de pagamento para outros objetos, no caso a Ordem de Venda e a Ordem de Serviço.

– Vou desenhar aqui no quadro um esboço para que todos possam compreender melhor. Observem o novo modelo da Figura 16.

STI: – Gostei, é mais fácil ver do que simplesmente ficar ouvindo. Agora dá pra ter uma noção melhor do projeto.

AS: – É, ainda vamos acrescentar alguns detalhes, mas por ora já é o bastante. Alguém tem mais alguma dúvida ou sugestão?

STI: – Não, por mim, está OK.

TP: – Tranquilo, o resto podemos discutir na empresa mesmo, AS.

GC: – Aguardarei nosso próximo encontro. Por favor, AS, agende com pelo menos dois dias de antecedência, tudo bem? As próximas semanas serão corridas para mim. Haverá lançamento de novos produtos e precisarei viajar constantemente para a Matriz, em São Bernardo.

AS: – Não se preocupe, GC. Teremos ainda muito trabalho antes de retornarmos aqui. Vou discutir ainda com minha equipe a modelagem e a arquitetura do sistema. Eu avisarei quando for necessário nos reunirmos novamente. Até mais a todos!

3.3.2 Componentes do Model-Driven Design

- **Local:** sala de reunião da empresa terceirizada;
- **Participantes:** analista de sistemas e técnico em programação;

TP: – Então, AS, o que achou do esboço da Figura 16?

AS: – Ainda está meio cru, TP, mas já apresenta o jeitão final do modelo. Lembrei agora que faltou o objeto Funcionário.

TP: – Sim, tem razão. E também temos que já definir quem será Entidade e quais serão os Value Objects. Por exemplo: Veículo e Cliente, com certeza são Entidades.

AS: – Concordo! Observe que o Veículo também possui algumas peculiaridades, que podem ser relevantes. Por exemplo: numa manutenção, os opcionais que o automóvel possui não importam, mas na ordem de venda eles são necessários.

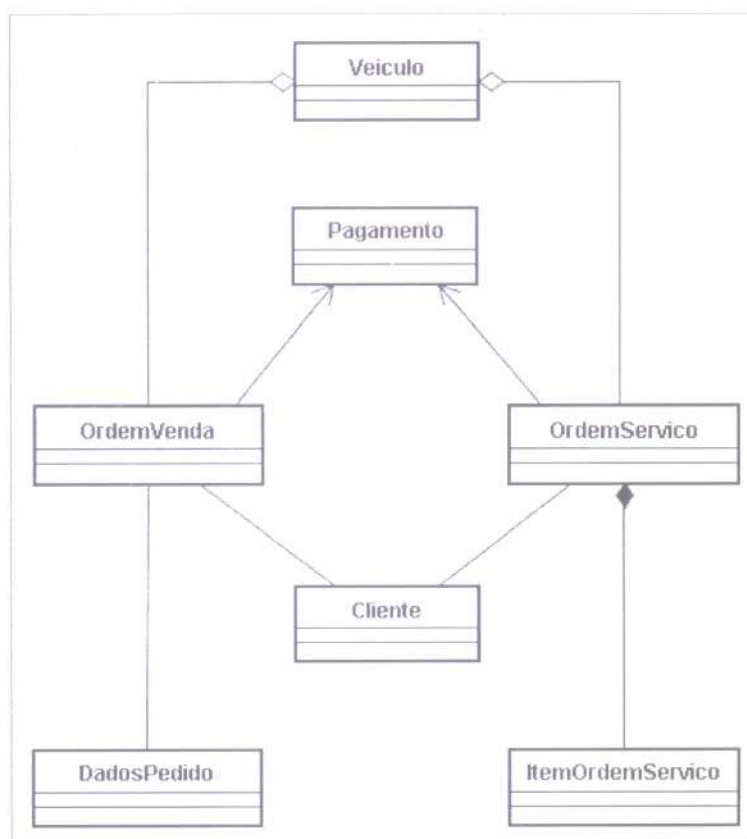


Figura 16 – Modelo da Concessionária (Revisão 2)

Fonte: Dados da Pesquisa (2008)

TP: – Discordo, em parte. Na manutenção, os opcionais também são necessários. E se, de repente, o cliente traz um carro com problema no vidro elétrico, que foi instalado pela própria concessionária?

AS: – É mesmo, não havia percebido esse detalhe. Sendo assim, vamos então colocar essa propriedade como um VO, porque ela não possui identidade própria.

TP: – Com certeza. Nem os opcionais, nem a cor e o modelo. E, já que a concessionária também instala equipamentos, devemos incluir na Ordem de Serviço um campo para os opcionais.

AS: – Certo. Mas isso não geraria confusão? Dois objetos *Opcionais* na mesma aplicação, em locais distintos?

TP: – Pode ser. Vamos ver o que o pessoal da concessionária acha.

AS: – É, mas, por enquanto, ficamos com o modelo da Figura 17. A meu ver, o resultado final está muito bom; o sistema está ganhando forma.

– Vou chamar o STI aqui para revermos essa questão dos opcionais. Amanhã, pode ser? Vai estar muito ocupado?

TP: – Pra mim, está ótimo. Dá tempo de terminar as alterações na interface web do Immobile.

AS: – Muito bem, estamos agendados para amanhã pela manhã. Até mais.

- **Local:** sala de reunião da empresa terceirizada;
- **Participantes:** analista de sistemas, supervisor de TI e técnico em programação;

TP: – Bom dia, STI, e obrigado por vir. Eu sei que seu tempo é curto, mas precisamos de sua opinião.

STI: – Bom dia, TP, AS. Realmente, o ritmo lá na concessionária é forte, mas esse programa é prioridade. O chefe viajou e me deixou focado nisso. Vamos adiantar o que for possível, pois na próxima semana ele volta e, com certeza, também dará sua opinião.

AS: – Claro. Queremos montar um protótipo o quanto antes. Fizemos algumas inclusões no modelo que você tinha visto anteriormente. Eu imprimi aqui a Figura 17. Alguma dúvida ou sugestão sobre os elementos do sistema?

STI: – Olhando bem, ainda faltam alguns detalhes. No Veículo, vocês precisam acrescentar a Categoria: se é Sedan, Hatch (compacto), Van ou Pick-up. Essa informação é importante na hora de calcular o valor do frete. Quanto maior o carro, mais espaço ele ocupa no caminhão-cegonha, que faz o transporte da fábrica até nós e, portanto, mais caro fica.

AS: – Obrigado, STI. Vamos incluir isso sim. O que mais você percebeu?

STI: – Aqui, na Ordem de Serviço, estou vendo que foi previsto um item para Peça (material) e outro para o Serviço (mão-de-obra). Mas, essa Peça é uma unidade ou pode ser um kit, como a fábrica costuma nos enviar? Por exemplo, um sensor de estacionamento.

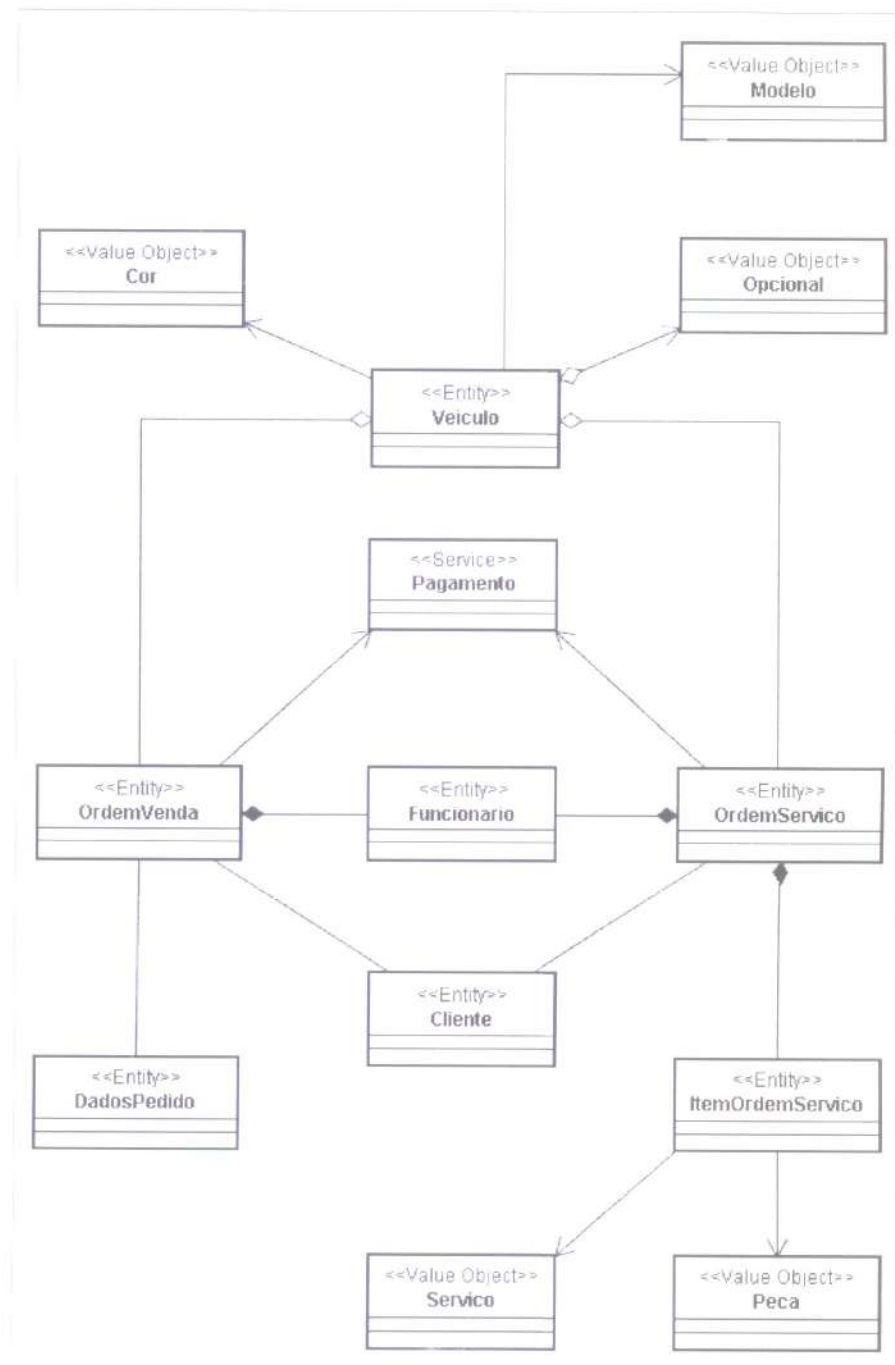


Figura 17 – Modelo da Concessionária (Revisão 3)

Fonte: Dados da Pesquisa (2008)

TP: – Era exatamente isso que estávamos comentando antes de você chegar. No caso, era sobre os opcionais originais de fábrica, ou os instalados na concessionária. Sabemos que existe essa possibilidade, mas não chegamos a um consenso em como implementar isso.

- STI:** – É melhor vocês fazerem assim: deixem o objeto Peça, e incluam um Opcional na Ordem de Serviço. Praticamente todos os opcionais podem ser instalados na Concessionária, e eles só vêm em kits completos. No caso das Peças, como pastilha de freio ou maçaneta de porta, elas vem em unidades.
- AS:** – OK, mas não vai confundir os vendedores quando eles forem cadastrar o veículo ou o serviço? Dois Opcionais em lugares distintos no sistema...
- Como irão saber se foi mesmo instalado na Concessionária?
- TP:** – AS, podemos criar um campo booleano para indicar se o Opcional foi instalado ou se é original da Fábrica. O sistema até conseguiria saber quais acessórios foram instalados pela Concessionária, apenas fazendo uma busca no banco de dados e já preencheria quando o vendedor desse entrada no veículo para manutenção. Seria útil até para verificar se o serviço ainda está na garantia.
- AS:** – Excelente, TP, ótima idéia. Estou anotando aqui. Por favor, STI, prossiga.
- TP:** – Bem, AS, pelo que consegui ver, era só isso. Talvez o GC possa acrescentar algo mais, quando ele voltar da viagem.
- AS:** – Sua ajuda foi de grande valia, STI. Vou esperar o GC retornar para concluirmos a modelagem e já passar para a programação de fato. Se tiver alguma outra sugestão, entre em contato por e-mail ou mesmo o telefone. Vou te enviar uma cópia do modelo para o seu e-mail.
- STI:** – Valeu, eu agradeço. Preciso ir agora, pois ainda tenho que fazer atualização do servidor.
- AS:** – OK, vá lá. Abraço e bom trabalho!
- TP:** – AS, eu estava pensando aqui, enquanto vocês dois conversavam.
- AS:** – Sim, estou ouvindo...
- TP:** – Nós criamos dois objetos (Entidades) diferentes para Ordem de Venda e Ordem de Serviço. Mas, se você pensar bem, elas possuem vários atributos em comum. Por exemplo, o Cliente, data, número da nota fiscal...

- AS:** – Entendi aonde você quer chegar: o projeto ficaria mais correto se nós fizéssemos uma generalização, com uma super-classe *Ordem* e as outras duas sendo derivadas dela?
- TP:** – Exato! *Ordem* seria a classe pai, e as demais estenderiam essa classe definindo seus atributos respectivos e próprios de cada uma.
- AS:** – Eu vou mais além: podemos definir, inclusive, que *Ordem de Serviço* é a raiz de um *Aggregate*.
- TP:** – Sim, e o mesmo se aplica a *Veículo*. O que acha de gerenciar com uma Fábrica a criação do objeto *Veículo*?
- AS:** – Acho que vai simplificar a codificação, com certeza, principalmente quando for necessário restaurar o objeto do banco de dados. Mas é melhor usar o *Factory Method* no próprio *Veículo*, pois só precisaremos criar este tipo de objeto.
- TP:** – É verdade, pra que complicar demais?
- AS:** – Ainda bem que aprendeu, aspira. Eu já tinha te chamado a atenção para esse seu hábito. Eu particularmente não gosto de super-projetar tudo; perde-se muito tempo tentando generalizar e prever todas as situações que possam ocorrer. É mais produtivo refatorar conforme a necessidade.
- TP:** – Você é um chefe legal, AS, aprendi muito mesmo contigo.
- AS:** – A recíproca é verdadeira, meu amigo. Afinal, quem me apresentou o Ruby, hein?
- TP:** – É mesmo, a danada da linguagem faz quase tudo sozinha... daqui a pouco, seremos descartáveis.
- AS:** – A produtividade é de fato muito alta. Mas, lembre-se da Regra de Ouro, TP: não existe *Silver Bullet* (Bala de Prata)! Não há uma solução universal para todos os problemas da computação. A melhor linguagem é aquela que resolve o seu problema. Não adianta tentar quebrar uma parede usando uma chave de fenda, ou usando dinamite; no primeiro caso, você iria passar um dia inteiro tentando e, no segundo, seria desperdício de força. Melhor usar a ferramenta adequada: uma simples broca furadeira!

TP: – É, um antigo professor me dizia isso, mas com outras palavras.

AS: – Esse teu professor era muito sábio, mas isso vem com a experiência.

– Quanto ao nosso projeto atual, agora penso que já temos uma versão melhor do sistema, não é? Veja a Figura 18 e me diga o que achou.

TP: – Acho que é mesmo nossa primeira versão funcional!

AS: – É assim que se fala! Reúna a equipe de programação, discuta o modelo com eles e aguarde minha reunião com o GC. Se ele aprovar, vocês já podem começar a escrever o primeiro protótipo.

TP: – Tudo bem, farei isso.

AS: – Beleza. Hoje à noite a pizza é por minha conta!

- **Local:** sala de reunião da concessionária;
- **Participantes:** gerente da concessionária, analista de sistemas e supervisor de TI;

GC: – Boa tarde, senhores, é bom revê-los! Estou ansioso pra saber as novidades sobre o projeto. Então, AS, em que pé estamos?

AS: – Boa tarde, GC! O que posso te dizer é que não podíamos estar melhor. Eu trouxe para o senhor praticamente a versão final do nosso modelo. Faltam apenas suas sugestões e comentários.

STI: – Pois é, chefe. Avançamos muito na sua ausência, mas eu não posso dar o aval definitivo sem antes te consultar. Além do mais, existe a possibilidade de eu esquecer alguma coisa.

GC: – Eu confio plenamente em você, STI, nem se preocupe quanto a isso. Mas eu queria mesmo ver antes, é sempre bom uma segunda opinião.

AS: – GC, eu trouxe aqui uma cópia da modelagem do sistema (Figura 18) para você analisar e nos dar sua opinião.

GC: – Senhores... eu estou convencido que, nunca antes na história desta concessionária, alguém me mostrou um projeto tão enxuto e, ao mesmo tempo, bem elaborado.

– Vejo que vocês se preocuparam até em diferenciar o automóvel por categoria. Muito bom!

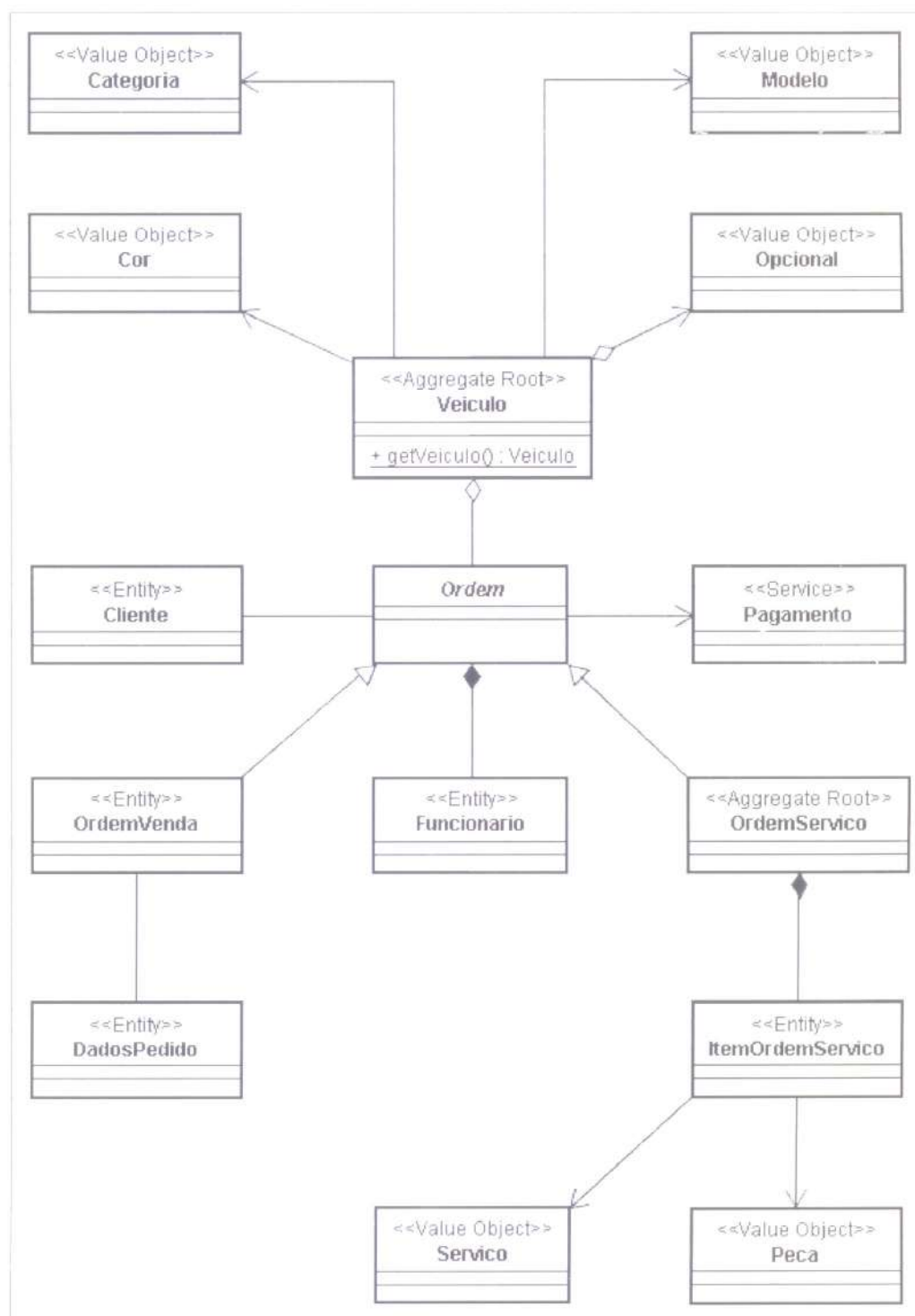


Figura 18 – Modelo da Concessionária (Revisão 4)

Fonte: Dados da Pesquisa (2008)

STI: – Eu dei essa sugestão, GC, justamente porque a empresa que nos fornece o serviço de frete calcula o seu custo baseado no tamanho do veículo.

GC: – Está mais do que certo, STI. Mas, existe um ponto que não nos atentamos até agora: toda Ordem, seja de Venda ou de Serviço, possui um Responsável, que pode ser um funcionário ou um gerente.

AS: – No caso, GC, quais seriam as diferenças entre o funcionário e o gerente? Quais os direitos que um tem, e o outro não?

GC: – A princípio, o gerente tem mais poderes, pois pode dar descontos maiores; o vendedor só dá desconto sobre vendas à vista, mas em compensação é o único que ganha comissão sobre cada venda sua que é faturada.

– No caso do gerente, ele não recebe comissão sobre uma venda unitária. O bônus dele é sobre um percentual do lucro durante o mês. Quanto mais os funcionários venderem, mais eles ganham por venda e, ao fim do mês, mais o gerente ganha também.

AS: – OK. Ambos recebem comissão, mas ela é calculada de maneiras diferentes. Vou incluir também um Serviço de Gerenciamento de Usuário, para definir os níveis de acesso e prioridades de cada operador do sistema.

– O gerente terá condições de oferecer descontos maiores no preço dos automóveis, os funcionários em geral poderão oferecer descontos apenas sobre vendas à vista e o administrador do sistema poderá cadastrar e excluir usuários e configurar os parâmetros da aplicação.

GC: – Perfeito, é isso mesmo. Está aprovado então, podem começar já!

STI: – AS, você tem alguma estimativa para quando teremos uma primeira versão do programa para testarmos?

AS: – Bem, STI, vou repassar a versão aprovada por vocês aos meus colegas e estimo em duas semanas para apresentarmos algo.

GC: – Então, ao trabalho. Quero logo pôr as mãos neste sistema e começar a trabalhar de forma mais racional aqui na Concessionária.

AS: – Então, senhores, até a próxima reunião!

3.4 CONSTRUÇÃO

Nesta etapa do projeto, inicia-se a fase de codificação do sistema. O modelo do negócio já se encontra no estágio em que os principais requisitos foram definidos e o escopo está delimitado. O foco deixa de ser a modelagem e passa a ser a implementação diminuindo, mas não eliminando, o contato entre os desenvolvedores e os clientes. Os diálogos terão continuidade a seguir.

- **Local:** sala dos programadores da empresa terceirizada;
- **Participantes:** analista de sistemas e técnico em programação;

AS: – TP, estive ontem na Concessionária em reunião com o GC e o STI. Ambos gostaram bastante do modelo que nós projetamos.

TP: – Beleza, e quando começamos?

AS: – Agora, se você quiser! Olhe só, aqui estão as alterações que nós fizemos na reunião, mostradas na Figura 19.

TP: – OK. Vocês incluíram um Responsável pelas Ordens, e aqui tem um Controle de Usuário. Isso foi você quem colocou, não foi?

AS: – Exato. Como cada tipo de Funcionário será portador de diferentes permissões e direitos sobre o sistema, decidi incluir esse Serviço.

TP: – Tudo bem, nós até temos uma API que usamos no Immobile e daria para adaptar aqui.

AS: – TP, você tem carta branca. Reúna-se com sua equipe, explique o modelo a eles e pode ligar os motores, usando um trocadilho infame.

TP: – Os meus já estão até aquecidos e roncando!

AS: – Sim, só mais uma pergunta: qual a arquitetura que você pretende utilizar no projeto?

TP: – Bom, como já possuímos uma boa base de código genérico, pensei em escrever a camada de infra-estrutura e domínio em Java SE 6 mesmo. Mas estou em dúvida quanto à interface gráfica: pela natureza do projeto, deve ser voltada para a web; mas, continuamos com Java mesmo, usando JSF, ou faço em Python com o Jython¹?

¹ Implementação da linguagem de script Python que executa na própria Java VM

- AS:** – Eu sei que você gosta do Python, mas por que não dá uma lida no *framework* Mentawai? Comecei a estudá-lo recentemente e gostei bastante. Além de ser um projeto nacional, com extensa documentação em português, praticamente não possui arquivos XML de configuração.
- Eles trabalham com configuração programática: as propriedades são definidas via chamada a métodos Java. Você fica menos propenso a erros, pois a própria IDE é capaz de validar sua configuração.
- TP:** – Que interessante, vou olhar mesmo. Pode ser uma boa alternativa pra escrevermos protótipos alfa...
- AS:** – Era exatamente essa a idéia. Como o sistema é simples, em poucos dias já poderemos apresentar uma versão demo para o GC.
- TP:** – OK, AS, considere feito.

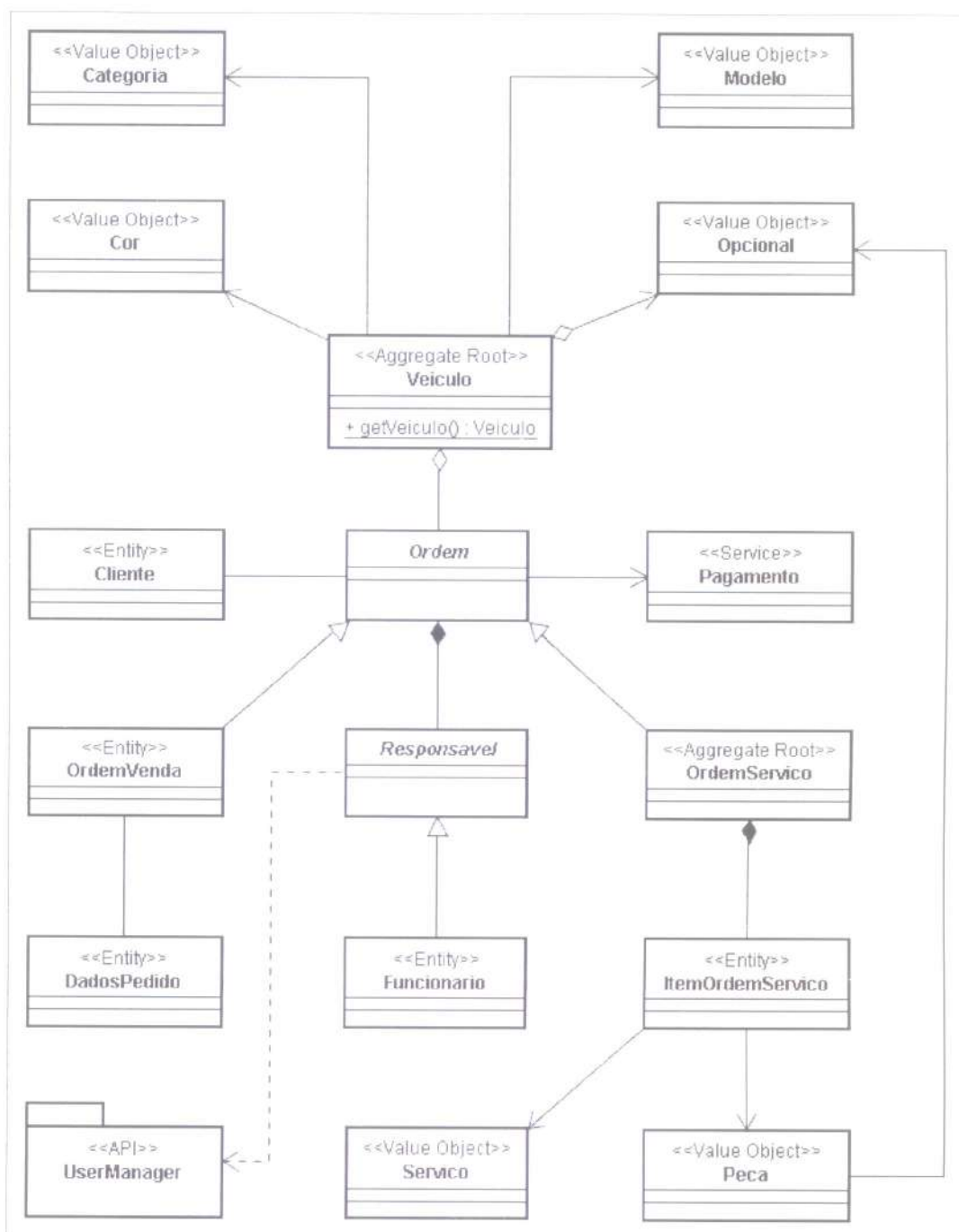


Figura 19 – Modelo da Concessionária (Versão Final)

Fonte: Dados da Pesquisa (2008)

3.5 TRANSIÇÃO

Esta é a última fase do ciclo de desenvolvimento do produto, no qual se concentram as atividades de implantação do *software* criado e treinamento dos futuros operadores. Em geral, toda transição deve ser bem planejada, considerando não somente os aspectos técnicos mas, principalmente, os psicológicos.

Usuários que já estão acostumados a lidar com um sistema consolidado tendem a oferecer muita resistência ao que é novo e diferente, mesmo que este seja melhor. Nestas situações, uma transição suave é mais indicada, em contraponto a uma abordagem radical. O esclarecimento sobre as novas vantagens e o ganho de produtividade reduzem a intolerância dos usuários. Uma solução imposta, implantada à revelia, e sem maiores explicações os deixam muito mais insatisfeitos e rebeldes, ocasionando às vezes a própria sabotagem do novo sistema.

Em relação às questões técnicas, os desenvolvedores e analistas do novo *software* devem garantir que, em seu futuro local de instalação, todos os requisitos foram satisfeitos, a saber:

- qualidade e redundância no fornecimento de energia elétrica;
- quando for o caso, se houver CPUs poderosas e em grande quantidade, refrigeração contínua e dedicada;
- cabeamento de rede certificado;
- *switches* com porta GigaBit Ethernet e, se houver cascadeamento, com conector para fibra ótica;
- solução automática para backup de dados, dentre outros...

Essa lista poderia crescer bastante, a depender apenas do porte e do orçamento destinado ao projeto. Soluções de segurança biométrica para acesso a uma sala-cofre, que abrigaria os servidores, além de sistema detector e eliminador de incêndios são alguns dos possíveis incrementos; sua aplicação se encaixaria melhor em aplicações de missão-crítica² (*critical mission*), os quais necessitam estar ativos vinte e quatro horas por dia, sete dias na semana, o que se conhece no meio técnico por 24/7. E, concluindo o Estudo de Caso, seguem os diálogos finais.

2 São assim chamados porque uma falha pode causar enormes prejuízos financeiros, e até mesmo humanos, no caso de indústrias nucleares ou hidroelétricas.

- **Local:** sala de reunião da empresa terceirizada;
- **Participantes:** gerente da concessionária, supervisor de TI, analista de sistemas e técnico em programação;

AS: – Bom dia! Estou muito feliz, pois hoje é um marco positivo no nosso projeto. Estamos entregando a primeira versão completamente funcional do AutoMobile, o *software* de automação comercial encomendado por vocês.

GC: – AS, eu compartilho imensamente de sua alegria! Estou tão contente quanto você. Este é um passo à frente que damos em relação à nossa concorrência pois, a partir de agora, poderemos ser mais eficientes e produtivos. Ganharemos nós e nossos clientes, sem dúvida.

STI: – E quando o AutoMobile será instalado? Por sinal, gostei do nome!

TP: – É, nosso chefe tem preferência pessoal por coisas *Mobiles*...

AS: – Mais adiante, eu explico aos senhores o porquê desse nome. Antes de instalar o *software* em definitivo, precisamos cuidar da parte técnica e do treinamento dos seus vendedores.

GC: – Do que precisarão?

AS: – Bem, GC, no início eu havia discutido com o TP para instalarmos tudo numa máquina dedicada aqui na empresa de vocês. Mas, depois surgiu a possibilidade de alugarmos o espaço em um provedor de hospedagem e serviços de Internet.

GC: – Por favor, AS, explique-me melhor.

AS: – OK. No primeiro caso, a vantagem é que vocês teriam um maior controle sobre o ambiente em que a aplicação será executada. A resolução de problemas se daria mais rapidamente. A desvantagem é que seria necessário investir em CPUs mais potentes e dedicadas para o AutoMobile, além de um sistema de refrigeração constante.

– Na segunda opção, a principal vantagem é o custo, pelo menos 50% menor, e vocês não se preocupariam em ter que atualizar servidor e dar manutenção no mesmo. E economizariam bastante em energia elétrica. Mas, como seríamos nós os responsáveis pela manutenção da

aplicação, cobraríamos uma mensalidade a título de resolução imediata dos problemas que venham a ocorrer. No primeiro caso, vocês nos pagariam por um contrato de manutenção anual, com direito a atualizações e pequenas modificações no sistema. Ao final, a segunda opção te traria uma economia de 30% aproximadamente.

GC: – Então, STI, qual é a sua opinião?

STI: – GC, eu opto pela locação dos serviços de hospedagem. É bastante confiável, e o provedor se encarrega de tudo. Sem contar que o imóvel em que a Concessionária está instalada é alugado. Qualquer modificação que fizermos, será investimento perdido caso nos mudemos para um local diferente.

TP: – Quero acrescentar, GC, que dispomos de uma equipe para suporte técnico das 7 às 21 hrs. Qualquer problema ou dúvida poderá ser sanado em poucos minutos, independente da opção escolhida.

GC: – Muito bem, concordarei com você, STI. De fato, a segunda solução é a que possui melhor custo-benefício. Ficaremos com ela.

AS: – GC, vou providenciar a instalação do AutoMobile no banco de dados que usamos como teste, para realizar o treinamento com seus vendedores. Quando eles estiverem familiarizados com o sistema, fechamos o contrato de locação com a empresa de hospedagem.

GC: – Tudo bem. Podemos marcar o treinamento para a próxima segunda?

TP: – Sim, até lá estarei com o *software* configurado e testado, pronto para usarmos.

GC: – Até segunda, então!

- **Local:** sala de reunião da concessionária;
- **Participantes:** gerente da concessionária, supervisor de TI, analista de sistemas e técnico em programação;

AS: – E então, GC, o que achou do treinamento e do sistema em si?

GC: – AS, estou muito surpreso com a facilidade e a simplicidade do Auto-

Mobile. No treinamento, quando você nos pediu para abrir o navegador de Internet, pensei que iríamos baixar algum programa ou ver alguma animação, ou vídeo. Não imaginava que era possível fazer um programa para executar no navegador.

STI: – Vocês usaram AJAX, não foi?

TP: – Na realidade, uma mistura de várias tecnologias, dentre elas o AJAX para a interação com o usuário. Os menus *pop-up*, a inclusão e exclusão de itens nas opções de venda e manutenção, tudo isso foi implementado usando HTML dinâmico e JavaScript...

STI: – Ficou excelente, bem Web 2.0, como o pessoal chama, não é?

AS: – Eu não sou partidário dessa denominação. A meu ver, é apenas uma forma diferente de utilizar tecnologias que já existiam mas, se querem enfeitar e fazer o marketing em cima, por mim tudo bem.

TP: – Na verdade, GC, a maior vantagem é não precisar instalar programa algum nos seus computadores e, além disso, a certeza de estar sempre usando a versão mais atual do AutoMobile.

– E, talvez você não saiba, mas seus vendedores poderão fazer tudo o que vocês aprenderam no curso, por meio de um *notebook* ou PDA com conexão de rede sem fio!

GC: – O que vem a ser um PDA?

TP: – A sigla em inglês significa *Personal Digital Assistant* (Assistente Pessoal Digital). Simplificando, é uma agenda eletrônica com recursos modernos de edição de texto e acesso à Internet. Os modelos mais avançados podem até acessar a rede de celular GSM.

STI: – É, um colega da faculdade comprou um com GPS e câmera digital de 2 megapixels.

GC: – Nossa, a tecnologia voa mesmo. Quem diria, tanta inovação em aparelhinhos tão pequenos?

AS: – Pois é, GC, por isso resolvemos chamar o *software* de AutoMobile: o foco do desenvolvimento foi a independência de Sistema Operacional e a total mobilidade; é possível utilizá-lo tanto de um computador comum, quanto de um celular ou PDA com acesso à Internet.

- GC:** – Só falta agora a substituição do sistema antigo...
- AS:** – Sim, iniciaremos a instalação gradual do sistema novo, e a remoção definitiva do antigo nos próximos dias. Até o momento, obtivemos bastante sucesso e estamos satisfeitos em encontrar clientes que entendam os desafios e obstáculos encontrados em um projeto de desenvolvimento de *software*.
- GC:** – Mas é claro que eu entendo! Vocês não têm idéia do que passamos para receber o sistema antigo...
- STI:** – Lhh, o programador que nos entregou o *software* atual não possuía qualquer método de trabalho, quase não vinha aqui e sempre atrasava os prazos. Sem falar que recebemos o programa sem o código-fonte, quando o combinado era que ele nos daria acesso a tudo. No final, ainda nos cobrou à parte pelos fontes, pois não havíamos assinado nenhum contrato. Ficamos mesmo à mercê dele, foi terrível! Sem falar dos travamentos no banco que nós comentamos logo no início...
- AS:** – É, nós também tivemos que aprender com nossos erros. Hoje em dia, combinamos tudo verbalmente e assinamos contratos especificando o que foi acordado. E preferimos trabalhar com contratos de escopo aberto, cobrando por homem-hora. Assim, se o cliente tiver sobra de verbas, pode solicitar mais funcionalidades no programa.
- GC:** – No final, eu dou nota 9,5 para o trabalho de vocês. Para receber um 10, só faltou cobrar um pouco mais barato...
- AS:** – Ah, GC, nós cobramos mais caro mas, em contrapartida, entregamos um produto com um padrão de qualidade superior. Não queremos ficar disputando clientes baixando o preço e canibalizando o mercado. Mas, se você nos indicar para outro cliente, podemos negociar um generoso desconto ou abater algumas mensalidades do AutoMobile.
- GC:** – Eu sei, era só uma provocação mesmo. Deus me livre passar por outra experiência daquelas! E pode deixar, meus amigos, que eu só darei ótimas referências sobre o seu trabalho!
- AS:** – Obrigado, e espero voltarmos a trabalhar juntos no futuro em outras empreitadas.

3.6 CONSIDERAÇÕES FINAIS

Neste Estudo de Caso, procurou-se simular as principais etapas que ocorrem em um processo de desenvolvimento de software, enfatizando os tópicos relativos à modelagem do domínio da aplicação. Embora a metodologia utilizada tenha sido principalmente o RUP, não é difícil perceber que foram introduzidos conceitos e práticas das metodologias ágeis, notadamente o XP. E, o que se observa na prática é uma liberdade no que diz respeito à adoção desses processos de desenvolvimento: algumas empresas oficialmente escolhem uma metodologia, mas incluem algumas práticas de outras que consideram mais produtivas.

As Metodologias Ágeis têm conquistado bastante espaço nas empresas que desenvolvem *softwares* sob encomenda, pois permitem alcançar uma maior eficiência na medida em que abole alguns artefatos das metodologias tradicionais. No XP, por exemplo, existe pouca ou nenhuma documentação. Os programadores utilizam os próprios comentários da linguagem de programação para documentar o software. A programação em par tende a gerar código com menos erros (TELES, 2007), pois cada programador revisa o código do parceiro, e assim gasta-se menos tempo com correção de *bugs*. O contato frequente com os clientes facilita a correção de rumos ao projeto logo nos momentos iniciais, diminuindo os custos das mudanças.

Ainda assim, é necessário salientar que a metodologia RUP também oferece alta produtividade e que os passos indicados nos itens anteriores não ocorrem uma única vez durante o processo de criação do software. Eles realmente acontecem na sequência, mas se repetem de forma iterativa – em *loop* – ao fim de cada Caso de Uso, ou História (no XP). É vital também identificar a real necessidade e importância dos artefatos gerados pelo RUP. Nem todos serão relevantes, ou passíveis de serem construídos, seja por restrição de orçamento ou escassez de recursos humanos. Cabe aos gerentes de projeto tomar essas decisões, e acompanhar o andamento do projeto, sempre alertas para efetuar as mudanças necessárias, buscando maximizar o escopo e diminuir os custos.

4 CONCLUSÃO

Conforme mencionado na Introdução deste trabalho, verifica-se no mercado de desenvolvimento de software, em especial nas denominadas *Fábricas de Software*, uma carência de profissionais com sólido embasamento técnico. O que habitualmente se encontram são programadores dependentes de ambientes integrados (IDE) que geram código automatizado, tornando-os escravos da ferramenta: o programador era quem deveria usar a IDE, e não o contrário.

Um dos objetivos deste trabalho foi chamar a atenção para as boas práticas de programação, focando principalmente na fase de análise. O *Domain-Driven Design* já se tornou bastante difundido no exterior e, atualmente, vem gradativamente atingindo os desenvolvedores brasileiros, pois sua eficácia é comprovada no uso diário. No segundo capítulo, foi apresentada uma introdução sobre os principais componentes do DDD, exemplificando algumas situações nas quais eles se aplicam, enfatizando os benefícios colhidos com a sua adoção.

De modo geral, um bom *design* tende a gerar *softwares* de melhor qualidade. E, ainda, uma falha em um *software* mal projetado acarretará consequências muito mais danosas do que no primeiro caso. Afinal, se existem padrões de codificação, uso dos *Design Patterns*, a existência de uma Linguagem Comum (Ubíqua) e uma política de testes automatizados com certeza o tempo necessário para localizar um defeito no programa será menor e, por consequência, o custo também se reduzirá. Isso porque todos os envolvidos no desenvolvimento possuem os conhecimentos necessários para entender e alterar qualquer parte do código.

No terceiro capítulo, salientando que esta monografia é um trabalho teórico de pesquisa, optou-se por uma abordagem diferenciada e incomum para ilustrar o Estudo de Caso: diálogos e situações foram criadas, com o intuito de demonstrar o que de fato ocorre durante um projeto de *software*, bem como a utilização e relevância do DDD no contexto geral do processo. Apesar da simplicidade do domínio escolhido, foi possível compreender como se dá a interação entre os clientes e os desenvolvedores ao longo do projeto. Ao final, chegou-se a um modelo conciso, elegante e que possui a maioria dos elementos do DDD demonstrados nos capítulos anteriores.

Porém, apesar de sua simplicidade, a maior dificuldade que um desenvolvedor encontra ao se deparar pela primeira vez com o DDD é, justamente, aquilo que o define. Torna-se difícil compreender e utilizar o *Domain-Driven Design* porque o mesmo se trata de uma metodologia, e não uma técnica, dando margem a diversas interpretações acerca dos seus conceitos. Quando transformar um objeto em uma Entidade ou um *Value Object*? Deve-se agrupar determinadas classes em Módulos ou *Aggregates*? Em quais situações utilizar um *Factory Method* ou uma *Abstract Factory*? Essas são perguntas que desencorajam o programador pouco experiente, e que nem sempre trazem uma resposta exata. Em geral, depende da finalidade do objeto, e dos requisitos do projeto. Um objeto do tipo *Veículo* pode ser um VO dentro de uma aplicação que controle Lava-Jatos, ou pode ser a raiz de um *Aggregate* como no modelo do capítulo 4. A maior recomendação é sempre analisar antes de partir para a codificação: o custo de corrigir um modelo é bem menor do que um *software* em desenvolvimento, mesmo que seja apenas um protótipo.

4.1 PERSPECTIVAS FUTURAS

Apenas a adoção do DDD ao projeto de um sistema de grande porte já trará bastante benefícios porém, o uso conjunto do DDD com outras tecnologias ou metodologias potencializam os ganhos. As mais difundidas estão listadas no Apêndice, mas é possível que no momento em que se esteja lendo este trabalho, algumas tenham sido atualizadas ou mesmo descontinuadas. O desenvolvedor deve sempre buscar a solidez na solução a ser implantada, evitando escolher um produto (*framework*) por modismo ou porque foi endossado por uma grande empresa.

Como projeto futuro, está sendo escrito um artigo técnico para um periódico especializado, expandindo a utilização do DDD em conjunto com as técnicas citadas no Apêndice. A proposta é atingir um público-alvo que seja formador de opinião e possa, dessa forma, retransmitir o conhecimento, esclarecendo os desenvolvedores sobre as boas práticas de programação e a necessidade de atualização constante neste meio bastante competitivo. No final, todos serão beneficiados, inclusive os clientes, que contarão com produtos de alta qualidade criados por profissionais brasileiros, gerando demanda no mercado nacional.

REFERÊNCIAS

ALUR, D.; CRUPI, J.; MALKS, D. **Core J2EE Patterns: Best Practices and Design Strategies**. Pearson Education, 2001.

AMBLER, Scott. **A Manager's Introduction to The Rational Unified Process (RUP)**. AmbySoft, 2005. Disponível em: <<http://www.ambysoft.com/unifiedprocess/rupIntroduction.html>>. Acesso em: 29 set. 2007.

AVRAM, Abel (comp.); MARINESCU, Floyd (org.). **Domain-Driven Design Quickly**. C4Media Inc, 2006. Disponível em: <<http://infoq.com/books/domain-driven-design-quickly>>. Acesso em: 12 out. 2007.

BAUER, C.; KING, G. **Java Persistence with Hibernate**. Rev. ed. Manning Publisher, 2006.

BECK, Kent. **Test-Driven Development: By Example**. Boston, USA: Addison-Wesley, 2003.

CALÇADO, Phillip. **Fantoches**. Fragmental, 2007. Disponível em: <<http://fragmental.com.br/wiki/index.php/Fantoches>>. Acesso em: 18 ago. 2007.

CALÇADO, Phillip. **Contratos Nulos**. Fragmental, 2005. Disponível em: <http://fragmental.com.br/wiki/index.php/Contratos_Nulos>. Acesso em: 20 set. 2007.

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

FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 1999.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Addison-Wesley, 2002.

GAMMA, E. et al. **Padrões de Projeto**. Porto Alegre: Bookman, 2000.

LARMAN, C. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design. Prentice Hall, 1998.

PRESSMAN, Roger. **Engenharia de Software**. 6. ed. São Paulo: McGraw-Hill, 2006.

PROJECT CARTOON. **How Projects Really Work (Version 1.0)**. Disponível em: <<http://www.projectcartoon.com/cartoon/3>>. Acesso em: 11 out. 2007.

RUMBAUGH, J. et al. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Campus, 1994.

SCHWABER, Ken. **SCRUM Development Process**. Advanced Development, 2007. Disponível em: <<http://jeffsutherland.com/oopsia/schwapub.pdf>>. Acesso em: 29 dez. 2007.

SPOLSKY, Joel. **Evidence Based Scheduling**. Fog Creek Software, 2007. Disponível em: <<http://www.joelonsoftware.com/items/2007/10/26.html>>. Acesso em: 26 nov. 2007.

SUN MICROSYSTEMS. **Java Naming and Directory Interface (JNDI)**. Disponível em: <<http://java.sun.com/products/jndi>>. Acesso em: 10 dez. 2007.

TELES, Vinícius Manhães. **Extreme Programming**. ImproveIT, 2007. Disponível em: <<http://www.improveit.com.br/xp>>. Acesso em: 15 ago. 2007.

WIKIPEDIA. **Processo Unificado**. Disponível em: <http://es.wikipedia.org/wiki/Proceso_Unificado>. Acesso em: 21 dez. 2007.

APÊNDICE

APÊNDICE A – TECNOLOGIAS AUXILIARES AO DDD

Os tópicos a seguir listam algumas metodologias de trabalho que auxiliam no desenvolvimento de software, em conjunto com o *Domain-Driven Design*. Qualquer opção pode ser utilizada individualmente, ou todas em conjunto a fim de se obter os melhores resultados.

O ambiente de programação escolhido é a plataforma Java, pois existe uma grande quantidade de *frameworks* em código-livre e farta documentação disponível para essa linguagem. Porém, é possível encontrar soluções similares, até do mesmo fabricante, para outros ambientes, tais como .Net e Ruby. Adicionalmente, todos os itens listados abaixo possuem fácil integração com a ferramenta de desenvolvimento *Eclipse*.

A.1 ASPECT-ORIENTED PROGRAMMING (AOP)

Em toda aplicação, existem requisitos que não dizem respeito exclusivamente ao domínio, tais como autenticação de usuário e acesso ao banco de dados. Estes requisitos, chamados de não-funcionais, são separados do domínio pela *Programação Orientada a Aspectos (AOP)*, a qual os configura e executa sob demanda, desacoplando o código e tornando-o mais legível.

A.1.1 Principal Ferramenta

- Nome Comercial: **AspectJ**;
- Licença: **Freeware**;
- Download: <http://www.eclipse.org/aspectj/downloads.php>;
- Documentação: <http://www.eclipse.org/aspectj/docs.php>;
- Doc. Adicional: <http://www.aspectprogramming.com>.

A.2 DOMAIN-SPECIFIC LANGUAGE (DSL)

A *Linguagem Específica do Domínio (DSL)* surgiu para permitir aos usuários que não possuem habilidades técnicas em programação criar suas próprias rotinas (subprogramas), facilitando a adição e modificação das regras de negócio. A DSL é uma linguagem de *script* de alto nível e se aproxima tanto quanto possível da linguagem natural do negócio – a *Linguagem Ubíqua*.

A.2.1 Principal Ferramenta

- Nome Comercial: **Jboss Drools**;
- Licença: **Freeware Open-Source**;
- Download: <http://labs.jboss.com/drools/downloads.html>;
- Documentação: <http://labs.jboss.com/drools/documentation.html>;
- Doc. Adicional: <http://martinfowler.com/bliki/dsl.html>.

A.3 MAPEAMENTO OBJETO-RELACIONAL (MOR)

Esta técnica também é conhecida como ORM – do original em inglês *Object-Relational Mapping* – e foi brevemente comentada ao final do Capítulo 2. Consiste em mapear os Objetos das linguagens OO para estruturas de dados tabulares dos Bancos de Dados Relacionais. Esse "mapeamento" busca abstrair as consultas ao BD via Linguagem Estruturada (Structured Query Language – **SQL**), transformando-as em objetos, deixando o programador mais à vontade para utilizar apenas a linguagem OO de alto nível.

A.3.1 Principal Ferramenta

- Nome Comercial: **Hibernate**;
- Licença: **Open-Source LGPL**;
- Download: <http://www.hibernate.org/6.html>;
- Documentação: <http://www.hibernate.org/5.html>;
- Doc. Adicional: [http://en.wikipedia.org/wiki/Hibernate_\(Java\)](http://en.wikipedia.org/wiki/Hibernate_(Java)).

A.4 TEST-DRIVEN DESIGN (TDD)

Trata-se do Desenvolvimento Baseado em Testes (**TDD**) de execução do programa, escritos especificamente para cada funcionalidade que se deseja validar. Antes de cada compilação do código-fonte, os testes são iniciados visando encontrar possíveis erros. Não se deve enviar o sistema para produção antes de solucionar todas as falhas detectadas pelos testes. Esta etapa garante um padrão mínimo de qualidade no *software* construído para o cliente, e será tão mais eficaz quantos forem os Casos de Testes (*Test Cases*).

A.4.1 Principal Ferramenta

- Nome Comercial: **JUnit**;
- Licença: **Freeware Open-Source**;
- Download: <http://www.junit.org>;
- Documentação: <http://junit.sourceforge.net/javadoc>;
- Doc. Adicional: <http://www.improveit.com.br/xp/praticas/tdd>.