

**Liga de Ensino do Rio Grande do Norte
Faculdade Natalense para o Desenvolvimento do RN**

JADSON JOSÉ DOS SANTOS

Análise da Utilização de Padrões no Desenvolvimento de Softwares em Camadas

Análise da Utilização de Padrões no Desenvolvimento de Softwares em Camadas

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Software da Faculdade Natalense para o Desenvolvimento do RN, como requisito obrigatório à obtenção do título de Especialista em Desenvolvimento de Sistemas Corporativos.

Orientador: David Ricardo do Vale Pereira, M.

JADSON JOSÉ DOS SANTOS

Análise da Utilização de Padrões no Desenvolvimento de Softwares em Camadas

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da FARN – Faculdade Natalense para o Desenvolvimento do RN, como requisito obrigatório à obtenção do título de Especialista em Desenvolvimento de Sistemas Corporativos.

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

Natal/RN
2007

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Santos, Jadson José dos

Análise da utilização de padrões no desenvolvimento de softwares em camadas / Jadson José dos Santos. – Natal: FARN, 2007.
98 f.

Orientador: David Pereira (M. Sc.)

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

1. Padrões de Projeto – Monografia. 2. Sistemas Corporativos – Monografia.
3. Desenvolvimento de Softwares em Camadas – Monografia. I. Título.

RN/FARN/BC

CDU 004.4

RB. 30711
FARN

BIBLIOTECA

Nº Reg

16, 04, 08

004.4
62850
2007
ex01

JADSON JOSÉ DOS SANTOS

Esta monografia tem como objetivo realizar um estudo sobre padrões e boas práticas de desenvolvimento de software, com ênfase em sistemas corporativos. Serão descritos diversos tipos de padrões e padrões das principais camadas de um sistema corporativo: arquitetura, aplicação, modelo e infra-estrutura, além de padrões que não são específicos e que são utilizados também, aqui chamados de padrões básicos. É apresentado também, um exemplo sobre a escolha fundamental em que todos os padrões estão fundamentados. Por fim, que possível, serão utilizadas duas abordagens para resolver o mesmo problema utilizando os padrões descritos neste texto e outra será utilizá-los, para que se possa observar a maior das características e vantagens de cada um.

Análise da Utilização de Padrões no Desenvolvimento de Softwares em Camadas

Monografia apresentada ao Curso de Especialização em Desenvolvimento de Sistemas Corporativos da FARN – Faculdade para o Desenvolvimento do RN, e 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 F. Lima
Membro Examinador

Membro Examinador

RESUMO

Este trabalho tem como objetivo realizar um estudo sobre padrões e boas práticas de desenvolvimento de software, com ênfase em sistemas corporativos. Serão descritos durante esse texto os principais padrões das principais camadas de um sistema corporativo: Apresentação, Aplicação, Modelo e Infra-Estrutura, além de padrões que não são específicos a uma determinada camada, aqui chamados de padrões básicos. É apresentado também, um capítulo sobre o conceito fundamental em que todos os padrões estão fundamentados: camadas. Sempre que possível, serão utilizadas duas abordagens para resolver o mesmo problema, uma utilizando os padrões descritos nesse texto e outra sem utilizá-los, para que se tenha uma percepção maior das características e vantagens de cada um.

Palavras Chave: Padrões de Projeto. Sistemas Corporativos. Desenvolvimento em Camadas.

LISTA ABSTRACTAÇÕES

This job has the goal to accomplish a study about patters and good practices of software developed, with emphasis in enterprise applications. It will be describe during this text Enterprise Application main patters of the main layers: Presentation, Application, Model and Infrastructure, behind of the patters that aren't specific of a particular layer, here called basic patters. It is also present, an exclusive chapter about the fundamental concept on which all patterns are substantiated: layers. It has tried, when is possible, show two different ways to encode the same problem, one way using the patters described in this text and other way without using them, to have a greater understanding of the characteristics and advantages of each one.

Key Words: Design Patterns. Enterprise Applications. Development in Layers.

Figura 1: Código fonte da classe <i>GerenciadorOrigem</i>	12
Figura 2: Código fonte da classe <i>ClienteOrigem</i>	12
Figura 3: Código fonte da classe <i>TestaOrigem</i>	13
Figura 12: O diagrama de classes representativo do padrão <i>Gateway</i>	15
Figura 14: Código fonte da classe <i>Order</i> sem usar o <i>Gateway</i>	16
Figura 12: Código fonte da classe <i>Order</i> usando o <i>Gateway</i>	16
Figura 13: Código fonte da classe <i>MessageGateway</i>	17
Figura 14: Código fonte da classe <i>MessageSender</i>	17
Figura 13: O diagrama de classes representativo do padrão <i>Layer Supertype</i>	18
Figura 16: Código fonte da classe <i>Gerenciador</i>	19
Figura 17: Código fonte da classe <i>GerenciadorCliente</i>	19
Figura 18: O diagrama de classes representativo do padrão <i>Separated Interface</i>	21
Figura 19: Código fonte da classe <i>Usuario</i>	21
Figura 20: Código fonte da interface <i>UsuarioDAO</i>	22
Figura 21: Código fonte da classe <i>UsuarioDAOImpl</i>	22
Figura 22: Código fonte da classe <i>PrepararBebidaBaseCafe</i>	24
Figura 23: Código fonte da classe <i>CafeExpresso</i>	24
Figura 24: Código fonte da classe <i>Cappuccino</i>	25
Figura 25: Código fonte da classe <i>Usuario</i> chamando o método <i>realizaPreparo</i> da classe <i>PrepararBebidaBaseCafe</i>	25
Figura 26: Saída resultante da execução do código da Figura 25	26
Figura 27: O diagrama de classes representativo do padrão <i>Strategy</i>	27
Figura 28: Código fonte da interface <i>PreparaCafeStrategy</i>	27
Figura 29: Código fonte da classe <i>PrepararCafe</i>	28
Figura 30: Código fonte da classe <i>CafeExpressoStrategy</i>	28
Figura 31: Código fonte da classe <i>CappuccinoStrategy</i>	28
Figura 32: Código fonte da classe <i>Usuario</i> executando a estratégia <i>CappuccinoStrategy</i>	29
Figura 33: A classe que implementa o padrão <i>Factory</i> para DAOs da classe <i>Usuario</i>	30

LISTA DE ILUSTRAÇÕES

Figura 1: Estrutura de um projeto em múltiplas camadas	8
Figura 2: Exemplo de uma aplicação em duas camadas	8
Figura 3: Camada de Apresentação da aplicação	8
Figura 4: Camada de Aplicação da aplicação	9
Figura 5: Camada de acesso aos dados da aplicação	9
Figura 6: Código fonte da classe <i>GerenteVendas</i>	12
Figura 7: Código fonte da classe <i>GerenteCompras</i>	12
Figura 8: Código fonte da classe <i>ChefeDelegation</i>	12
Figura 9: Código fonte da classe <i>TestaDelegation</i>	13
Figura 10: O diagrama de classes representativo do padrão <i>Gateway</i>	15
Figura 11: Código fonte da classe <i>Order</i> sem usar o <i>Gateway</i>	16
Figura 12: Código fonte da classe <i>Order</i> usando o <i>Gateway</i>	16
Figura 13: Código fonte da classe <i>MessageGateway</i>	17
Figura 14: Código fonte da classe <i>MessageSender</i>	17
Figura 15: O diagrama de classes representativo do pattern <i>Layer Supertype</i>	18
Figura 16: Código fonte da classe <i>GenericGerenciador</i>	19
Figura 17: Código fonte da classe <i>GerenciadorCliente</i>	19
Figura 18: O diagrama de classes representativo do padrão <i>Separated Interface</i>	21
Figura 19: Código fonte da classe <i>Usuario</i>	21
Figura 20: Código fonte da interface <i>UsuarioDAO</i>	22
Figura 21: Código fonte da classe <i>UsuarioDAOImpl</i>	22
Figura 22: Código fonte da classe <i>PrepararBebidaBaseCafe</i>	24
Figura 23: Código fonte da classe <i>CafeExpresso</i>	24
Figura 24: Código fonte da classe <i>Cappuccino</i>	25
Figura 25: Código fonte da classe <i>Usuario</i> chamando o método <i>realizaPreparo</i> da classe <i>PrepararBebidaBaseCafe</i>	25
Figura 26: Saída resultante da execução do código da Figura 25.	26
Figura 27: O diagrama de classes representativo do padrão <i>Strategy</i>	27
Figura 28: Código fonte da interface <i>PreparoCafeStrategy</i>	27
Figura 29: Código fonte da classe <i>PrepararCafe</i>	28
Figura 30: Código fonte da classe <i>CafeExpressoStrategy</i>	28
Figura 31: Código fonte da classe <i>CappuccinoStrategy</i>	28
Figura 32: Código fonte da classe <i>Usuario</i> executando a estratégia <i>CappuccinoStrategy</i>	29
Figura 33: A classe que implementa o padrão <i>Factory</i> para DAOs da classe <i>Usuario</i>	30

Figura 34: Dependência de classes com o padrão <i>Factory</i>	30
Figura 35: Código Fonte da classe cliente para a classe <i>DAOFactory</i>	31
Figura 36: O diagrama de classes representativo do padrão <i>Plugin</i>	32
Figura 37 Código fonte da classe <i>UsuarioDAOImplArquivo</i>	33
Figura 38: Código fonte da classe <i>PluginFactory</i>	34
Figura 39: Dependência de objetos usando instanciação direta.....	36
Figura 40: Dependência de objetos usando o padrão <i>Dependency Injection</i>	37
Figura 41: Classe para teste do padrão <i>Dependency Injection</i>	38
Figura 42: Classe <i>Usuario</i> com o Método que Usa o Padrão <i>Dependency Injection</i>	38
Figura 43: O Arquivo <i>ApplicationContext.xml</i> do <i>Spring</i> para Configurar as dependências.....	39
Figura 44: Classe Atendente para um Modelo Anêmico.....	42
Figura 45: Classe Chamada para um Modelo Anêmico.....	43
Figura 46: Classe Problema para um Modelo Anêmico.....	43
Figura 47: Classe SistemaChamadaNegocio para um Modelo Anêmico.....	43
Figura 48: Classe SistemaChamadaNegocio e para um Modelo Anêmico continuação.....	44
Figura 49: Diagrama de seqüência de um Modelo Anêmico.....	45
Figura 50: Dominio Lógico de uma aplicação de registro de chamadas.....	46
Figura 51: Classe Atendente para um <i>Domain Model</i>	47
Figura 52: Classe Chamada para um <i>Domain Model</i>	47
Figura 53: Classe Chamada para um <i>Domain Model</i> continuação.....	48
Figura 54: Classe Problema para um <i>Domain Model</i>	48
Figura 55: Classe Solicitante para um <i>Domain Model</i>	49
Figura 56: Diagrama de Seqüência de um <i>Domain Model</i>	50
Figura 57: Dependências do padrão <i>Data Mapper</i>	59
Figura 58: Arquitetura do padrão <i>Data Mapper</i>	62
Figura 59: Código Fonte da classe <i>GenericDAO</i>	63
Figura 60: Continuação do Código Fonte da classe <i>GenericDAO</i>	64
Figura 61: Código Fonte da classe <i>UsuarioDAO</i>	65
Figura 62: Código Fonte da classe <i>Usuario</i>	65
Figura 63: Arquivo <i>hibernate.cfg.xml</i>	66
Figura 64: Código para salvar um objeto da classe <i>Usuario</i> no Banco de Dados.....	66
Figura 65: Código Fonte do Repositório da classe <i>Usuario</i>	68
Figura 66: Código Fonte da classe <i>UsuarioDAO</i> usando Repositório.....	69
Figura 67: Código Fonte da classe <i>Usuario</i> usando o padrão <i>Active Record</i>	71
Figura 68: Esquema de dependências do padrão <i>Facade</i>	73

Figura 69: Código fonte da classe <i>AutomovelFacade</i>	74
Figura 70: Código fonte da interface <i>Command</i>	76
Figura 71: Código fonte da classe <i>LigaTVComando</i>	76
Figura 72: Código fonte da classe <i>Invoker</i>	77
Figura 73: Código fonte da classe <i>Cliente</i>	78
Figura 74: Código fonte da classe <i>MacroComando</i>	78
Figura 75: Esquema de dependências do padrão MVC.....	81
Figura 76: Código fonte da página de consulta de processos usando <i>Smart UI</i>	82
Figura 77: Continuação do Código Fonte da página de consulta de processos usando <i>Smart UI</i>	83
Figura 78: Continuação do Código Fonte da página de consulta de processos usando <i>Smart UI</i>	84
Figura 79: Página de Consulta de Processos usando o padrão <i>Smart UI</i>	85
Figura 80: Arquitetura de um sistema de vendas utilizando padrões.	87
Figura 81: A classe <i>AbstractManagedBean</i> da camada de Apresentação.....	88
Figura 82: A classe <i>CompraBean</i> da camada de Apresentação	89
Figura 83: A classe <i>GerenciadorCompraImpl</i> da camada de Aplicação	90
Figura 84: A classe <i>Compra</i> da camada de Modelo.....	91
Figura 85: A classe <i>Compra</i> da camada de Modelo continuação.....	92
Figura 86: A interface <i>CompraRepositorio</i> da camada de Modelo.....	92
Figura 87: A classe <i>GenericDAO</i> da camada de Infra Estrutura.....	93
Figura 88: A classe <i>CompraDAO</i> da camada de Infra Estrutura.....	94
3.2.1 Conceito	34
3.2.2 Exemplo de uso do padrão <i>Dependency Injection</i>	35
4 PADRÃO DA CAMADA DE MODELO.....	40
4.1 O PADRÃO DOMAIN MODEL	40
4.2 COMPARANDO UM MODELO NÚMERO COM O PADRÃO DOMAIN MODEL.....	42
4.3 DOMAIN DRIVEN DESIGN - DDD	50
4.3.1 Entity.....	53
4.3.2 Value Object.....	53
4.3.3 Services.....	53
4.3.4 Módulos.....	54
4.3.5 Agregações.....	55
4.3.6 Factory.....	56
4.3.7 Repositórios.....	58
5 PADRÕES DA CAMADA DE PERSISTÊNCIA.....	59
5.1 DATA MAPPER.....	59
5.1.1 Conceito.....	59
5.1.2 Exemplo de uso do Padrão <i>Data Mapper</i>	62
5.2 REPOSITORY.....	67
5.2.1 Conceito.....	67
5.2.2 Exemplo de uso do Padrão <i>Repository</i>	68

SUMÁRIO

1	INTRODUÇÃO	1
2	DESENVOLVIMENTO DE SOFTWARE EM CAMADAS	4
2.1	CONCEITOS.....	4
2.2	VANTAGENS E DESVANTAGENS.....	6
2.3	EXEMPLO DO USO DE CAMADAS	7
3	PADRÕES BÁSICOS	11
3.1	DELEGATION	11
3.1.1	Conceito.....	11
3.1.2	Exemplo do uso do padrão Delegation	11
3.2	GATEWAY	13
3.2.1	Conceito.....	13
3.2.2	Exemplo de uso do padrão Gateway	14
3.3	LAYER SUPERTYPE	18
3.3.1	Conceito.....	18
3.3.2	Exemplo de uso do padrão Layer Supertype.....	18
3.4	SEPARATED INTERFACE	19
3.4.1	Conceito.....	19
3.4.2	Exemplo de uso do padrão Separated Interface	20
3.5	TEMPLATE METHOD	23
3.5.1	Conceito.....	23
3.5.2	Exemplo de uso do padrão Template Method.....	23
3.6	STRATEGY	26
3.6.1	Conceito.....	26
3.6.2	Exemplo de uso do padrão Strategy	27
3.7	FACTORY	29
3.7.1	Conceito.....	29
3.7.2	Exemplo do uso do padrão Factory.....	30
3.8	PLUGIN.....	31
3.8.1	Conceito.....	31
3.8.2	Exemplo de uso do padrão Plugin	32
3.9	DEPENDENCY INJECTION.....	34
3.9.1	Conceito.....	34
3.9.2	Exemplo de uso do padrão Dependency Injection.....	35
4	PADRÃO DA CAMADA DE MODELO.....	40
4.1	O PADRÃO DOMAIN MODEL.....	40
4.2	COMPARANDO UM MODELO ANÊMICO COM O PADRÃO DOMAIN MODEL	42
4.3	DOMAIN DRIVEN DESIGN - DDD	50
4.3.1	Entity.....	53
4.3.2	Value Object.....	53
4.3.3	Services.....	53
4.3.4	Módulos	54
4.3.5	Agregações	55
4.3.6	Factory.....	56
4.3.7	Repositórios.....	58
5	PADRÕES DA CAMADA DE PERSISTÊNCIA.....	59
5.1	DATA MAPPER	59
5.1.1	Conceito.....	59
5.1.2	Exemplo de Uso do Padrão Data Mapper	62
5.2	REPOSITORY.....	67
5.2.1	Conceito.....	67
5.2.2	Exemplo de Uso do Padrão Repository.....	68

5.3	ACTIVE RECORD	69
5.3.1	Conceito.....	69
5.3.2	Exemplo de Uso do Padrão Active Record	70
6	PADRÕES DA CAMADA DE APLICAÇÃO.....	72
6.1	FACADE	72
6.1.1	Conceito.....	72
6.1.2	Exemplo de Uso do Padrão Facade	73
6.2	COMMAND.....	75
6.2.1	Conceito.....	75
6.2.2	Exemplo de Uso do padrão Command	75
7	PADRÕES DA CAMADA DE APRESENTAÇÃO	79
7.1	MODEL VIEW CONTROLER (MVC)	79
7.1.1	Conceito.....	79
7.2	SMART UI.....	81
7.2.1	Conceito.....	81
7.2.2	Exemplo de uso do padrão Smart UI.....	82
8	EXEMPLO DE ARQUITETURA COM PADRÕES	86
9	CONCLUSÕES.....	95
	REFERÊNCIAS	97

Um padrão também serve como uma documentação para o seu sistema, partindo do pressuposto que os padrões são de conhecimento de todos os desenvolvedores. Isso facilita entender a solução dada a um problema, se essa solução for construída em cima de um padrão. Como os padrões, pelo menos os padrões voltados para o desenvolvimento de sistemas corporativos, buscam reduzir o acoplamento e aumentar a coesão entre os componentes de um sistema, diminuindo a duplicação do código e possibilitando o reuso destes componentes, o custo de manutenção da aplicação tende a cair e qualidade do código (grau de fácil entendimento) aumentar.

A aplicação dessa ideia na área de desenvolvimento de software surgiu no final da década de 80, quando Kent Beck e Ward Cunningham (BECK, 1987), que trabalhavam com Smalltalk em projetos de interfaces gráficas, resolveram catalogar cinco "design patterns" para evitar que jovens programadores cometessem erros já conhecidos por programadores mais experientes, além de possibilitar que o projeto no qual eles trabalhavam, tivesse um ritmo de desenvolvimento mais rápido. A partir daí, a ideia de utilizar padrões para desenvolver software começou a crescer, com cada vez mais gente se interessando e ajudando sobre o assunto.

Em 1994 ocorreu um evento chamado OOPSLA'94 (Object-Oriented Programming Systems, Languages, and Applications), onde Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides apresentaram um livro que representaria um marco para o uso de patterns na área de software. O livro se chamava "Design Patterns: Elements of Reusable Object-

1 INTRODUÇÃO

A palavra “padrão” como uma forma de padronizar soluções para problemas comuns foi utilizado pela primeira vez na área de arquitetura e engenharia civil por Christopher Alexander, citado por (FRANTZ, 2007, p.1), para solucionar problemas que costumavam acontecer corriqueiramente. Assim, ao invés de se desperdiçar tempo procurando uma solução quando se deparasse com um problema, bastava procurar seu registro feito por alguém que já o tivesse solucionado anteriormente. Essa solução catalogada provavelmente já teria sido bastante discutida e, conseqüentemente, seria uma solução sólida e robusta.

Outra definição para complementar a citada anteriormente é a de Fowler (2002, p.18), no livro *Patterns of Enterprise Application Architecture*: “Padrões não são idéias originais, eles são muito mais observações que acontecem no campo de trabalho. Como resultado, os autores de padrões não dizem: nós ‘inventamos’ um padrão, mas nós ‘descobrimos’ um”.

Um padrão também serve como uma documentação para o seu sistema, partindo do pressuposto que os padrões são de conhecimento de todos os desenvolvedores, fica muito mais fácil entender a solução dada a um problema, se essa solução for construída em cima de um padrão. Como os padrões, pelo menos os padrões voltados para o desenvolvimento de sistemas corporativos, buscam reduzir o acoplamento e aumentar a coesão entre os componentes de um sistema, diminuindo a duplicação do código e possibilitando o reuso desses componentes, o custo de manutenção da aplicação tende a cair e qualidade do código (claro de fácil entendimento) aumentar.

A aplicação dessa idéia na área de desenvolvimento de software surgiu no final da década de 80, quando Kent Beck e Ward Cunningham (BECK, 1987), que trabalhavam com *Smalltalk* em projetos de interface gráfica, resolveram catalogar cinco “*design patterns*” para evitar que jovens programadores cometessem erros já conhecidos por programadores mais experientes, além de possibilitar que o projeto no qual eles trabalhavam, tivesse um ritmo de desenvolvimento mais rápido. A partir daí, a idéia de utilizar padrões para desenvolver software começou a crescer, com cada vez mais gente se interessando e estudando sobre o assunto.

Em 1994 ocorreu um evento chamado OOPSLA'94 (*Object-Oriented Programming, Systems, Languages, and Applications*), onde Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides apresentaram um livro que representaria um marco para o uso de *patterns* na área de software. O livro se chamava “*Design Patterns: Elements of Reusable Object-*

Oriented Software” e esses quatro rapazes ficam conhecidos como *Gang of Four* (GoF) (GAMMA et al, 1994). Até os dias atuais esse livro é muito referenciado e vários dos seus padrões são aplicados em projetos de desenvolvimento de softwares orientados a objetos.

Muitos autores costumam catalogar um *pattern* de acordo com quatro características principais:

- Nome: o nome do padrão;
- Problema: o problema que o padrão se propõe a resolver;
- Solução: como o padrão é implementado;
- Consequência (consequência do uso desse padrão para o sistema).

Outros autores, como o GoF, adicionam outros elementos para descrever um *pattern*, tais como Contexto em que o problema ocorre, Motivação, Aplicabilidade, “Conhecido como”, Estrutura, Colaboração, Implementação, Exemplo de Código, entre outros. Ainda não existe um consenso sobre a melhor forma de catalogação.

Existem vários tipos de *patterns*, os mais conhecidos são os *Design Patterns* que foram descritos no livro do GoF (GAMMA et al, 1994). Esses padrões estão mais relacionados com a arquitetura e o projeto de software. Mas existem várias classificações para *patterns*, dependendo do autor. Muitas dessas classificações dão nomes diferentes para os mesmos tipos de *patterns*, na maioria das vezes classificam-se os *patterns* em três categorias.

Segundo (FRANTZ, 2007, p.8), um grupo de autores conhecidos como *Gang of Five*, classifica os *patterns* em três grupos:

- *Architectural patterns*: Padrões mais genéricos que cuidam da estrutura fundamental da aplicação, por exemplo, como vai ocorrer a comunicação com sistemas externos.
- *Design patterns*: Um pouco mais restritos, tratam da estruturação de componentes de um sistema, como exemplo pode-se ter padrões que ditem as regras de como classes de uma mesma camada podem interagir.
- *Idioms*: Padrões de mais baixo nível, que estrutura aspectos particulares de um componente do sistema.

Outros dois grandes autores da área, Riehle e Zullighoven, citados por (FRANTZ, 2007, p.8), classificam os padrões em outros três grupos de acordo com a fase de desenvolvimento:

Conceptual Patterns, *Design Patterns* e *Programming Patterns*, indo da fase inicial do desenvolvimento (análise) até a fase de codificação do sistema¹.

A tendência é de que cada vez mais os padrões sejam usados para o desenvolvimento de software por trazerem uma série de vantagens atrativas para as empresas, como as citadas no parágrafo anterior.

Essa monografia está organizada da seguinte maneira: no Capítulo 2, é descrito o padrão de desenvolvimento de software em camadas, além de ser realizada uma comparação entre sistemas com os mesmos requisitos, mas com um sendo escrito em camadas e outro sem camadas. No Capítulo 3 são mostrados os padrões de uso mais genérico que não se encaixam em nenhuma camada. No Capítulo 4 são mostrados os padrões responsáveis por organizar a lógica de sua aplicação, além de um breve explicação sobre *Domain Driven Design*. Em seguida, no Capítulo 5 são descritos os padrões que realizam a persistência dos dados da aplicação. No Capítulo 6, os padrões que são intermediários entre a camada de apresentação e a lógica do sistema, os padrões da camada de aplicação. No Capítulo 7, os padrões responsáveis por fornecer serviços a sistemas externos (entenda-se por sistemas externos qualquer tipo de sistema, até mesmo usuários), os padrões da camada de apresentação. No Capítulo 8, é apresentada uma pequena aplicação que procura unir grande parte dos padrões descritos durante o todo o texto, para dar uma melhor noção de como ocorrem as interações entre os padrões quando são usados em conjunto. Por último, são apresentadas as conclusões do trabalho.

Por causa de todas essas características citadas acima, aplicações corporativas têm um alto grau de complexidade e precisam estar preparadas para sofrerem mudanças constantemente, por isso, muitos dos padrões descritos aqui estão focados em facilitar as mudanças na estruturas e regras sobre as quais o software foi construído.

Começar-se-á descrevendo o principal padrão utilizado para aplicações corporativas: camadas. A maioria dos padrões utilizados no desenvolvimento de aplicações corporativas está relacionado com a separação do código em camadas, isto é, eles procuram definir como essas camadas estão organizadas e como é feita a comunicação entre elas.

Phillip Calçado (CALÇADO, 2005, p.36) fala, em seu artigo *Arquitetura de Camadas em Java EE*, que existem duas métricas básicas de qualidade que devem ser seguidas por um

¹ Nos dias atuais, esse modelo de desenvolvimento em cascata está em desuso, por ter uma eficácia menor em relação a modelos mais modernos como o iterativo e incremental, onde em cada *release* do software são realizadas as fases de análise, desenvolvimento e teste.

2 DESENVOLVIMENTO DE SOFTWARE EM CAMADAS

O presente capítulo procura descrever os principais conceitos envolvidos com a técnica de desenvolvimento de software em camadas, metodologia largamente aceita e utilizada no desenvolvimento de softwares corporativos. Os padrões descritos nesse trabalho são baseados nessa técnica de desenvolvimento. Em seguida, serão descritas as vantagens e desvantagens da divisão em camadas usando um exemplo prático.

2.1 CONCEITOS

Existem diferentes tipos de software para diferentes tipos de aplicações, cada uma com suas características e complexidades. Os padrões descritos durante todo esse texto são padrões que foram catalogados visando aplicações corporativas, definidas assim por (FOWLER et al, 2002):

Aplicações que manipulam uma grande quantidade de dados (que devem ser persistidos, às vezes por muitos anos), são acessados por uma grande quantidade de pessoas concorrentemente e possuem uma quantidade significativa de código que se destina a mostrar esses dados de diferentes maneiras e com diferentes propósitos, para os vários usuários que a utilizam. Geralmente precisam se integrar com outros sistemas, construídos em diferentes tempos com diferentes tecnologias e trabalham com as regras de negócio da instituição para a qual estão sendo desenvolvidos, que geralmente estão em constante mudança.

Por causa de todas essas características citadas acima, aplicações corporativas têm um alto grau de complexidade e precisam estar preparadas para sofrerem mudanças constantemente, por isso, muitos dos padrões descritos aqui estão focados em facilitar as mudanças na estruturas e regras sobre as quais o software foi construído.

Começar-se-á descrevendo o principal padrão utilizado para aplicações corporativas: camadas. A maioria dos padrões utilizados no desenvolvimento de aplicações corporativas está relacionado com a separação do código em camadas, isso é, eles procuram definir como essas camadas estão organizadas e como é feita a comunicação entre elas.

Phillip Calçado (CALÇADO, 2005, p.34) fala, em seu artigo *Arquitetura de Camadas em Java EE*, que existem duas métricas básicas de qualidade que devem ser seguidas por um módulo de software (método, classe, pacote ou qualquer divisão que se faz em um software dependendo do tamanho do mesmo): coesão e acoplamento. Módulos de software devem ter

uma alta coesão, isto é, todos os componentes desse módulo deveriam ter as mesmas responsabilidades e, idealmente, cada módulo deveria estar focado em resolver um problema específico. Além disso, devem ter um baixo acoplamento, ou seja, todo módulo deveria ser o mais independente possível de outro e uma alteração da implementação de um módulo não deveria afetar nenhum outro.

Um alto grau de acoplamento entre muitos objetos é sinal de um design pobre porque deixa o código difícil de ler e entender e, mais importante, torna o código pouco flexível a mudanças. “O uso de Camada é um padrão arquitetural que ajuda na tarefa de separar responsabilidades, promovendo baixo acoplamento e alta coesão em um sistema”. (CALÇADO, 2005, p.34). Partindo desse princípio ele conceitua camadas como “divisões lógicas de componentes agrupados por responsabilidades em comum. Idealmente, Camadas se comunicam apenas com Camadas adjacentes”. (CALÇADO, 2005, p.34-35).

A necessidade de se separar uma aplicação em camadas surgiu no começo da década de 90 com os sistemas com arquitetura cliente-servidor. Existem geralmente duas camadas nesses sistemas: a interface gráfica com o usuário e a camada de persistência (geralmente um Banco de Dados relacional).

Há duas possibilidades de implementação para esses sistemas: “cliente rico” ou “cliente pobre” (conhecidas também como: “cliente gordo” ou “cliente magro”). No cliente rico a lógica da aplicação está embutida junto com o código da interface gráfica com o usuário. Na medida em que os sistemas vão crescendo e se tornando mais complexos, essa técnica gera um grande número de problemas, como uma grande quantidade de código repetido. Uma pequena mudança pode ter um impacto enorme na aplicação (e isso é o que exatamente se quer evitar em sistemas corporativos).

A outra abordagem seria colocar a lógica da aplicação em *stored procedures*² no banco de dados relacional. Mas essa solução também tem suas desvantagens como, por exemplo, a perda de portabilidade, a dificuldade de trabalhar com transações quando está envolvido mais de um SGDB e não se poder trabalhar com orientação a objetos e suas vantagens.

Diante desses problemas, surgiu a necessidade de se evoluir para a chamada programação em três camadas, na qual se permanece com a camada de interface gráfica com o usuário (também conhecida como camada de apresentação), a camada de persistência, mas foi criada uma nova camada entre as duas anteriores, que modela o domínio³ da aplicação e onde

² Procedimentos armazenados: Funções que os SGBDs possuem para realização alguma tarefa no banco de dados. Podem encapsular a lógica de negócio da aplicação.

³ Domínio: As informações e regras do “mundo real” que seu sistema precisa manipular.

ficam as regras de negócio⁴ da mesma; chamada por muitos autores de camada de negócio, modelo⁵, domínio ou domínio lógico. Com a introdução dessa camada, resolveu-se grande parte dos problemas citados anteriormente.

As camadas de persistência e a de negócio nunca deveriam ser dependentes da camada de apresentação, pois essa dependência geraria muita duplicação de código, visto que os dados de uma aplicação podem ser mostrados em diferentes partes e de diferentes maneiras e, conseqüentemente, acarretaria muito trabalho para se alterar qualquer parte desse sistema. Martin Fowler (FOWLER et al, 2002, p.26) define assim a diferença entre as funções das camadas de apresentação e persistência: "A camada de apresentação é um serviço que seu sistema presta para alguém, pode ser um ser humano ou outro sistema, a camada de persistência é uma interface para coisas que estão provendo ao seu sistema".

A separação dessas três camadas não precisa ser fixa. Para sistemas simples podem ser três métodos diferentes, para sistemas mais complexos pode-se separar as três camadas em três classes diferentes. Com o aumento da complexidade, faz-se necessária a divisão das três camadas em três pacotes. A forma de realizar a separação não é única e depende de cada caso, mas o importante é que se separe o sistema em três camadas.

2.2 VANTAGENS E DESVANTAGENS

Calçado (2005, p.35) enumera algumas vantagens e desvantagens em desenvolver sistemas utilizando uma arquitetura em camadas:

Vantagens:

- Reduzem complexidade: agrupam componentes e simplificam a comunicação entre eles;
- Reduzem dependência/acoplamento: a regra de comunicação evita dependências diretas entre componentes de Camadas diferentes;
- Favorecem a coesão: componentes de responsabilidades relacionadas são agrupados;
- Promovem reusabilidade: camadas podem ser reutilizadas em outros sistemas ou podem ser substituídas;
- É um padrão arquitetural conhecido: facilita a comunicação e entendimento entre desenvolvedores.

Desvantagens:

- Limitadas pela tecnologia: algumas regras precisam ser quebradas por limitações tecnológicas.
- Apenas complicam um sistema muito simples: não é qualquer sistema que exige o uso de Camadas;
- Possibilidade de overdose: muitos arquitetos acabam criando Camadas demais e tornando a aplicação extremamente complexa.

⁴ Regras de Negócio: O trabalho que um sistema precisa realizar sobre os dados para cumprir seus requisitos (tarefas que o sistema precisa realizar).

⁵ Modelo: Uma representação particular do domínio.

2.3 EXEMPLO DO USO DE CAMADAS

Nesta seção são apresentados exemplos, para efeito de comparação, da implicação do uso do padrão de desenvolvimento em camadas. É apresentada uma mesma aplicação bem simples que apenas insere dados de uma empresa digitados pelo usuário em um Banco de Dados relacional, projetada de duas formas diferentes: Na primeira, com o código de inserção dos dados junto com o código responsável pelo tratamento do evento da interface gráfica com o usuário, Figura 2. A segunda, projetada em uma arquitetura multicamadas (Figuras 3, 4 e 5). Na Figura 1, é mostrada a estrutura desse projeto.

A segunda forma desse exemplo foi projetada para ter quatro camadas: Apresentação, Aplicação, Modelo e Persistência. Na Figura 3, tem-se a classe *EmpresaUI* que representa a camada de *Apresentação* da aplicação, esta camada é responsável por construir a interface gráfica com o usuário e tratar seus eventos. Na Figura 4, mostra-se o código fonte da classe *GerenciadorEmpresa*, essa classe pertence a camada de *Aplicação*, esta é a responsável por coordenar a sequência das tarefas da aplicação. A camada de *Modelo*, que possui uma modelagem do domínio da aplicação (já explicado anteriormente nesse capítulo), está representada nesse exemplo pelo objeto *Empresa*, nesse caso este objeto não possui Regras de Negócio, apenas os dados da aplicação. A camada de persistência foi implementada usando-se o padrão *DAO*, Figura 5, que é um Java EE⁶ *design pattern* para acessar dados em um Banco de Dados (visto mais adiante no Capítulo 5). E o *Repositório de Dados* é um repositório onde os dados vão ser salvos e está fora do contexto da nossa aplicação.

À primeira vista, a aplicação da Figura 2 parece ser a melhor opção, porque no outro exemplo (das Figuras 3, 4 e 5), para uma simples inserção de dados de um objeto do tipo empresa no Banco de Dados é preciso percorrer um longo caminho:

Apresentação → *Aplicação* → *Modelo* → *DAO* → *Repositório de Dados*

Se fosse necessário mostrar para o usuário se essa inserção obteve êxito ou não, seria preciso percorrer o caminho inverso para seguir a arquitetura:

Repositório de Dados → *DAO* → *Modelo* → *Aplicação* → *Apresentação*

⁶ Java Enterprise Edition: Conjunto de tecnologias da plataforma Java para desenvolver aplicações corporativas.



Figura 1: Estrutura de um projeto em múltiplas camadas.
Fonte: O próprio autor.

```

/**
 * EmpresaUI.java
 */
private void cadastrarEmpresa(ActionEvent evt) throws SQLException,
    ClassNotFoundException{

    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

    String url = "jdbc:odbc:Something";

    Connection con = DriverManager.getConnection(url);

    PreparedStatement st = con.prepareStatement(
        "INSERT INTO EMPRESA (NOME, DESCRICAO) values (?,?)");
    st.setString(1, jTextField1.getText());
    st.setString(2, jTextArea1.getText());
    st.executeUpdate();
    st.close();
}

```

Figura 2: Exemplo de uma aplicação em duas camadas.
Fonte: O próprio autor.

```

/**
 * EmpresaUI.java
 */
public void cadastrarEmpresa(ActionEvent evt) {

    GerenciadorEmpresa gerenciador = new GerenciadorEmpresa();

    gerenciador.cadastrarEmpresa(jTextField1.getText(),
        jTextArea1.getText());
}

```

Figura 3: Camada de Apresentação da aplicação
Fonte: O próprio autor.

```

/**
 * GerenciadorEmpresa.java
 */
public void cadastrarEmpresa(String nome, String descricao){

    Empresa e = new Empresa(nome, descricao);

    EmpresaDAO dao = new EmpresaDAO();

    try {
        dao.create(e);
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(null, ex.getMessage());
    } finally{
        dao.close();
    }
}

```

Figura 4: Camada de Aplicação da aplicação
Fonte: O próprio autor.

```

/**
 * EmpresaDAO.java
 */
public void create(Empresa e) throws Exception {

    PreparedStatement st = con.prepareStatement(
        "INSERT INTO EMPRESA (NOME, DESCRICAO) values (?,?)");
    st.setString(1, e.getNome());
    st.setString(2, e.getDescricao());
    st.executeUpdate();
    st.close();
}

```

Figura 5: Camada de acesso aos dados da aplicação.
Fonte: O próprio autor.

Para aplicações simples, como o exemplo anterior, que não irão crescer e nem mudar com frequência, realmente o primeiro projeto é a melhor opção. Como citado anteriormente por (CALÇADO, 2005, p.35): “camadas apenas complicam um sistema muito simples”.

Agora, se a aplicação desse exemplo fosse uma aplicação corporativa, que tivesse de exibir muitos dados complexos para diferentes usuários, a replicação de código de acesso a Banco de Dados seria enorme. Uma pequena mudança nesses dados (por exemplo, um objeto empresa vai ter o campo “nome” substituído por “CNPJ”), causaria uma mudança no código nas centenas de telas da interface com os usuários, que o sistema provavelmente iria possuir.

Outro questionamento poderia ser a respeito da perda de desempenho, pois existem muito mais chamadas a métodos em aplicações multicamada do que se o código de acesso ao Banco de Dados estivesse localizado diretamente no código da interface gráfica. Se forem chamadas remotas de métodos, poder-se-ia pensar em retirar algumas camadas para melhorar o desempenho, mas nesse caso como se trata de chamadas locais, a perda de desempenho é praticamente imperceptível.

Em aplicações corporativas onde existem diversas tarefas a serem realizadas, a separação em camadas torna o código do sistema mais coeso, facilitando o seu entendimento.

No próximo capítulo, serão apresentados padrões que não pertencem a uma camada específica ou são padrões simples, usados para compor os demais padrões mostrados mais adiante nesse trabalho.

3 PADRÕES BÁSICOS

Neste capítulo serão apresentados alguns padrões de projeto, que podem ser usados diretamente da maneira que foram definidos ou como base para compor padrões mais complexos. Os padrões listados neste capítulo não pertencem a uma camada específica da aplicação, diferentemente dos demais padrões que serão apresentados nos próximos capítulos.

3.1 DELEGATION

3.1.1 Conceito

Delegation é um padrão no qual um objeto delega a tarefa ou parte da tarefa que ele deveria implementar a outros objetos, sem que os seus cliente tomem conhecimento, isto é, sem que o código dos objetos que acessam a *Delegation* precise mudar. A relação entre o objeto invocado pelo cliente e o objeto a que é delegada a tarefa é uma composição.

Normalmente classes muito grandes possuem códigos duplicado e são difíceis de entender. Delegar código duplicados e tarefas que não são de responsabilidade do objeto a outros objetos é uma boa prática de *design* e ajuda a reduzir o tamanho dos códigos das classes das quais objetos são instanciados, melhora a reusabilidade (reaproveitamento de um mesmo trecho de código em vários lugares de uma aplicação sem precisar replicá-lo), os processos de debug e teste e facilita o entendimento dos objetos e seus métodos (DEUGO, 2007, p.2). *Delegation* permite a objetos compartilhar implementação sem usar herança e sem duplicar código.

É preciso tomar cuidado para não exagerar e delegar todos os métodos de um objeto a outro objeto, isso pode aumentar muito o número de classes de uma aplicação e, conseqüentemente, aumenta a complexidade de entendimento do seu código. Uma boa medida é perceber a coesão dos objetos, a tarefa que for de responsabilidade do objeto é bom que ele a execute, e não seja delegada a outros.

Delegation pode ser um problema também para sistemas distribuídos, pois ele aumenta o número de chamada a métodos de objetos. Como essas chamadas serão remotas, o desempenho da aplicação pode ser prejudicado.

3.1.2 Exemplo do uso do padrão *Delegation*

O exemplo para demonstrar o padrão *Delegation* consiste em um chefe de uma loja varejista, que precisa realizar uma liquidação de estoque. Para isso, delega todas as tarefas que podem ser realizadas por outros funcionários, deixando para realizar apenas aquelas tarefas que são da sua competência exclusiva.

A classe *ChefeDelegation* que implementa o padrão *Delegation* é mostrada da Figura 8. Ela é a classe responsável por realizar uma liquidação na loja varejista. Para que o chefe possa cumprir essa tarefa “complexa” é necessário delegar duas tarefas menores aos seus gerentes de vendas (Figura 6) e compras (Figura 7). Por fim, tem-se um exemplo de uma classe cliente: *TesteDelegation* (Figura 9), que apenas instancia as classes anteriores e chama o método *realizarLiquidacao()* da classe *ChefeDelegation*.

```
public class GerenteVendas{
    public String calculaPrecoVendaMinimo(){
        return "calculando preço de venda minimo";
    }
}
```

Figura 6: Código fonte da classe *GerenteVendas*
Fonte: O próprio autor.

```
public class GerenteCompras{
    public String realizaPesquisaMercado(){
        return "realizando pesquisa no mercado";
    }
}
```

Figura 7: Código fonte da classe *GerenteCompras*
Fonte: O próprio autor.

```
public class ChefeDelegation{
    private GerenteCompras g1;
    private GerenteVendas g2;

    public Delegation(GerenteCompras g1, GerenteVendas g2){
        this.g1 = g1;
        this.g2 = g2;
    }

    public String realizaLiquidacao(){
        return g1.realizaPesquisaMercado() +
            g2.calculaPrecoVendaMinimo() + decidePrecoLiquidacao();
    }

    private string decidePrecoLiquidacao(){
        return "decidindo preço no liquidação";
    }
}
```

Figura 8: Código fonte da classe *ChefeDelegation*
Fonte: O próprio autor.

```

public class TesteDelegation{

    public static void main(String[] args){
        GerenteVenda gv = new GerenteVendas();
        GerenteCompras gc = new GerenteCompras();
        ChefeDelegation chefe = new ChefeDelegation(gv, gc);
        System.out.println(chefe.realizaLiquidacao());
    }
}

```

Figura 9: Código fonte da classe *TesteDelegation*
Fonte: O próprio autor.

3.2 GATEWAY

3.2.1 Conceito

Um *Gateway* é um objeto que encapsula o acesso a um sistema ou a um recurso externo do sistema.

Sistemas Corporativos normalmente precisam se comunicar com recursos externos que quase sempre são sistemas não orientados a objetos, como um Banco de Dados relacional ou um *Web Service*⁷. A principal função do *Gateway* é esconder a complexidade de acesso a uma API externa. Se o sistema precisar mudar o recurso externo que ele está acessando, a única classe que precisará ser reimplementada é a classe que representa o *Gateway*. Isso muito importante em sistemas corporativos, que têm como uma de suas características as constantes mudanças.

Mesmo se o recurso que o sistema estiver acessando tiver uma probabilidade de mudança muito pequena, pode-se pensar no uso do *Gateway* para facilitar os testes de partes do sistema que envolvam acessos a esses recursos externos, pois pode-se alterar a implementação de um *Gateway* para que o mesmo simule o acesso a um sistema externo, sem que o restante do sistema precise tomar conhecimento que o recurso externo não está presente, assim não é preciso que o recurso esteja disponível para o sistema ser testado.

Outra vantagem do padrão *Gateway* é deixar o código do seu sistema legível, pois toda a complexidade de acesso a uma API externa fica encapsulada na classe do *Gateway*. É uma boa escolha usar um *Gateway* sempre que se tiver uma interface um pouco complexa para algum recurso que pareça externo ao sistema.

⁷ Web Service: Solução para integração de sistemas utilizando a linguagem de marcação XML e o protocolo SOAP.

Este padrão é muito similar aos padrões *Facade*, *Adapter* e *Mediator* do GoF (GAMMA et al, 1994), (FOWLER, 2002, p.468) cita algumas diferenças entre esses quatro padrões.

- Enquanto *Facade* simplifica uma API mais complexa, o que geralmente é feito escrevendo-se um serviço de uso geral, um *Gateway* é escrito pelo cliente para o seu uso particular. Além disso, o *Facade* sempre implica em uma diferente interface para o que se está ocultando, já um *Gateway* pode copiar inteiramente a fachada envolvida, usando para propósito de teste.
- O *Adapter* altera uma implementação de uma interface para combinar com outra interface que se precisa trabalhar. Com *Gateway* geralmente não existe essa interface, em vez disso, você pode usar um *Adapter* para mapear uma implementação para uma interface do *Gateway*. Nesse caso o *Adapter* é uma parte da implementação do *Gateway*.
- O *Mediator* usualmente separa múltiplos objetos para que eles não saibam da existência um do outro, mas apenas da existência do *Mediator*. Com um *Gateway* existem geralmente apenas dois objetos envolvidos e o recurso que está sendo encapsulado não sabe da existência do *Gateway*.

3.2.2 Exemplo de uso do padrão *Gateway*

Nessa subseção é mostrado um exemplo do uso do padrão *Gateway*, através de um sistema simples que precisa acessar um serviço de mensagens (esse exemplo de sistema de mensagens foi extraído de (FOWLER, 2002, p. 468)). Na Figura 10, é mostrado um diagrama de classes que representa o esquema do *Gateway*. Têm-se três classes: *MessageSender*, que representa um sistema de envio de mensagens e possui uma interface complicada de ser entendida; a classe *Order* que representa o sistema cliente que precisa se comunicar com a classe *MessageSender* para o envio de mensagens; e a classe *MessageGateway* que é a classe que implementa o padrão *Gateway*.

Como o sistema de mensagem deve ser utilizado por um número grande de clientes, sua interface deve ser o mais genérica possível. O método *send()* da classe *MessageSender*, que implementa o envio de mensagens, recebe dois parâmetros. Uma *String* indicando o tipo da mensagem e um *array* de Objetos que são os argumentos da mensagem.

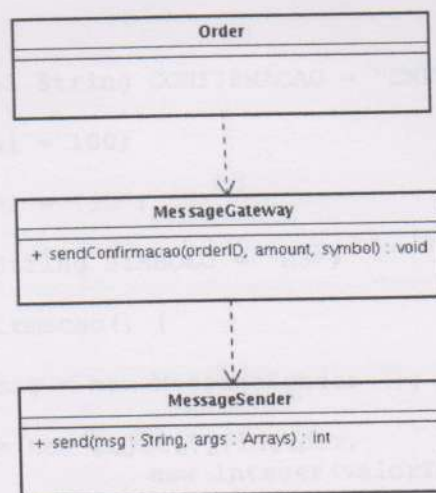


Figura 10: O diagrama de classes representativo do padrão *Gateway*
Fonte: O próprio autor.

Olhando para a assinatura do método *send()* da classe *MessageSender* é impossível dizer quais parâmetros devem ser passados a um determinado tipo de mensagem. Todo programador que esteja desenvolvendo alguma parte de um sistema que precise se comunicar com esse sistema de mensagem, deve ser conhecedor de todos os tipos de mensagem suportados e quais parâmetros esses tipos de mensagem devem receber, o que torna o desenvolvimento exaustivo e improdutivo. Nesse exemplo se aplica bem o uso de um *Gateway*, pois está se acessando um recurso externo ao sistema e se quer esconder a complexidade de acesso a essa aplicação externa.

A classe *MessageGateway* possui apenas um método *sendConfirmacao()* que envia uma mensagem do tipo confirmação e possui todos os seus parâmetros bem definidos para tal tipo de mensagem. Outra coisa que esse *Gateway* simplifica são os códigos de erros no envio de mensagens. No sistema de envio de mensagens, os erros são repassados através de códigos de erro que representam o tipo de problema que ocorreu, tal qual em linguagens mais antigas como C. O *Gateway* converte esses códigos para exceções, uma maneira mais lógica e clara de se reportar erros.

Na Figura 11 é apresentado o código do cliente (a classe *Order*) que usa o sistema de mensagens sem o uso do *Gateway*, na Figura 12 é mostrado o código do mesmo cliente agora fazendo o uso do *Gateway*.

```

public class Order{

    protected static final String CONFIRMACAO = "CNFRM";

    private int valorTotal = 100;

    private String idOrder = "10";

    public static final String SIMBOLO = "R$";

    public void sendConfirmacao() {

        MessageSender msg = new MessageSender ();

        Object[] args = new Object[]{idOrder,
                                     new Integer(valorTotal), SIMBOLO};

        int ack = msg.send(CONFIRMACAO, args);

        if(ack != MessageSender.SUCESSO)
            System.out.println(" Ocorreu erro!");

    }
}

```

Figura 11: Código fonte da classe Order sem usar o *Gateway*
 Fonte: O próprio autor.

```

public class Order{

    private int valorTotal = 100;

    public static final String SIMBOLO = "R$";

    private String idOrder = "10";

    public void sendConfirmacao() {

        MenssageGateway msg = new MenssageGateway();

        msg.sendConfirmacao( idOrder, valorTotal, SIMBOLO);

    }
}

```

Figura 12: Código fonte da classe Order usando o *Gateway*
 Fonte: O próprio autor.

Apesar de ser um exemplo simples, nota-se um aumento na complexidade da classe *Order* na Figura 11. É aconselhável encapsular essa complexidade dentro de um *Gateway*. O código do *Gateway* é mostrado na Figura 13 e o da classe *MessageSender* na Figura 14.

```

public class MenssageGateway {

    protected static final String CONFIRMACAO = "CNFRM";

    private MessageSender sender = new MessageSender();

    public void sendConfirmacao(String orderID, int amount,
        String symbol){

        Object[] args = new Object[]{orderID, new Integer(amount),
            symbol};

        send(CONFIRMACAO, args);

    }

    private void send(String msg, Object[] args){
        int returnCode = doSend(msg, args);

        if(returnCode == MessageSender.PARAMETRO_NULO)
            throw new NullPointerException("Parametro null: "+msg);

        if(returnCode != MessageSender.SUCESSO)
            throw new IllegalStateException(
                "Inesperado erro do sistema de mensagem #: "+ returnCode);
    }

    protected int doSend(String msg, Object[] args){
        assert(msg != null);
        return sender.send(msg, args);
    }

}

```

Figura 13 Código fonte da classe *MessageGateway*
 Fonte: O próprio autor.

```

public class MessageSender {

    public final static int PARAMETRO_NULO = -1;
    public final static int TIPO_MENSAGEM_DESCONHECIDA = -2;
    public final static int SUCESSO = 0;

    public int send(String msg, Object[] args){
        // lógica de envio de mensagens
        return 0;
    }

}

```

Figura 14: Código fonte da classe *MessageSender*
 Fonte: O próprio autor.

3.3 LAYER SUPERTYPE

3.3.1 Conceito

Um tipo que contém todos os atributos e métodos comuns aos objetos da camada a qual ele pertence. É muito comum para objetos de uma mesma camada terem métodos e atributos em comum. Uma boa prática, para evitar duplicação de código, é colocar esses atributos e métodos em comum em uma classe que todas as classes dessa camada possam herdar, isso é o padrão *Layer Supertype*. Se existir mais de um tipo de objeto em uma mesma camada, com atributos não comuns, é aceitável ter mais de um objeto *Layer Supertype*.

3.3.2 Exemplo de uso do padrão *Layer Supertype*

Na Figura 15 é mostrado um diagrama que representa o padrão *Layer Supertype*. O exemplo muito simples consiste apenas de uma classe *GenericGerenciador* que possui os dados comuns da camada de aplicação, nesse exemplo, o método *verificaAutorizacaoUsuario()* que executa a ação de verificar se um determinado usuário possui autorização para realizar uma tarefa mediante o *login* e a *senha* dele. Como essa operação deve ser utilizada por todas as classes dessa camada, fez-se necessário colocá-la na classe pai da camada.

Nas Figuras 16 e 17 é mostrado o código das classes que são esquematizadas na Figura 15.

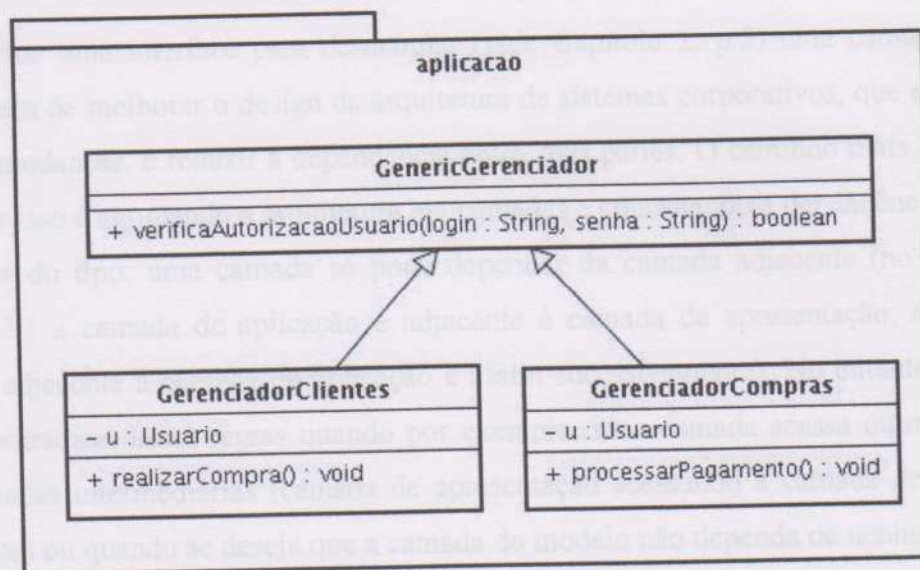


Figura 15: O diagrama de classes representativo do padrão *Layer Supertype*
Fonte: O próprio autor.

```

public class GenericGerenciador {

    public boolean verificaAutorizacaoUsuario(String login,
                                              String senha){

        //lógica para verificar autorização do usuário

    }

}

```

Figura 16: Código fonte da classe *GenericGerenciador*
Fonte: O próprio autor.

```

public class GerenciadorClientes extends GenericGerenciador{

    private Cliente c; //já instanciado

    public void realizaCompra(){
        if(this.verificaAutorizacaoUsuario(c.getLogin(), c.getSenha())){
            //lógica para realizar uma compra do usuário
        }else{
            System.out.println("Sem autorização para realizar
                               essa operação");
        }
    }

}

```

Figura 17: Código fonte da classe *GerenciadorCliente*
Fonte: O próprio autor.

3.4 SEPARATED INTERFACE

3.4.1 Conceito

Define uma interface para desacoplar (vide Capítulo 2, p.5) uma camada de outra. Uma maneira de melhorar o design da arquitetura de sistemas corporativos, que estão sempre sujeitos a mudanças, é reduzir a dependência entre suas partes. O caminho mais comum para se alcançar isso é agrupando a arquitetura em camadas e controlando a dependência entre elas, com regras do tipo: uma camada só pode depender da camada adjacente (no exemplo da subseção 2.3 a camada de aplicação é adjacente à camada de apresentação, a camada de modelo é adjacente à camada de aplicação e assim sucessivamente). No entanto, as vezes é preciso contradizer essas regras quando por exemplo, uma camada acessa outra sem passar pelas camadas intermediárias (camada de apresentação acessando a camada de persistência diretamente) ou quando se deseja que a camada do modelo não dependa de nenhuma outra (da camada de persistência por exemplo). Nesses casos o uso do padrão *Separated Interface* ajuda a diminuir essas dependências entre as camadas. No exemplo da camada de apresentação

acessando diretamente a camada de persistência, a apresentação passaria de depender da interface da camada de persistência. Mesmo que as classes da camada de persistência mudem, se a interface não mudar, a apresentação não precisará mudar. Na prática, ainda existe uma dependência indireta, se as mudanças na persistência forem tão grandes que afetarem a interface também, será preciso refazer suas classes da apresentação. É quase impossível eliminar essas dependências completamente, elas sempre vão existir.

3.4.2 Exemplo de uso do padrão *Separated Interface*

Na Figura 18 é mostrado o esquema do padrão *Separated Interface*. Existe a classe *Usuario* da camada de modelo que precisa acessar a camada de persistência para persistir os seus dados. Em vez de acessar diretamente os métodos da classe *UsuarioDAOImpl* do pacote *dao*, a classe *Usuario* só enxerga os métodos da interface *UsuarioDAO*⁸, que está na sua própria camada, então a classe *Usuario* nem toma conhecimento que existe uma camada *dao* de persistência.

Em sistema onde apenas uma camada deve acessar a outra, essa interface pode ficar dentro da camada cliente, e quem desenvolver a camada cliente será responsável por definir a interface entre esta camada e a camada servidora (camada que presta o serviço). Em sistemas onde várias camadas dependem dos serviços de uma terceira camada, a interface entre essas camadas pode ficar em outra camada separada. Nesse caso o desenvolvedor da implementação será também o responsável por definir a interface.

O código da classe *Usuario* é mostrado na Figura 19. Nas Figuras 20 e 21 são apresentados os códigos da interface *UsuarioDAO* e da classe *UsuarioDAOImpl* respectivamente.

⁸ Está se usando nesse exemplo a estrutura interface da linguagem Java, para linguagens que não suportam esse tipo de estrutura, pode-se usar uma classe abstrata. Classes abstratas podem ser uma boa escolha também se é preciso fornecer alguma implementação padrão.

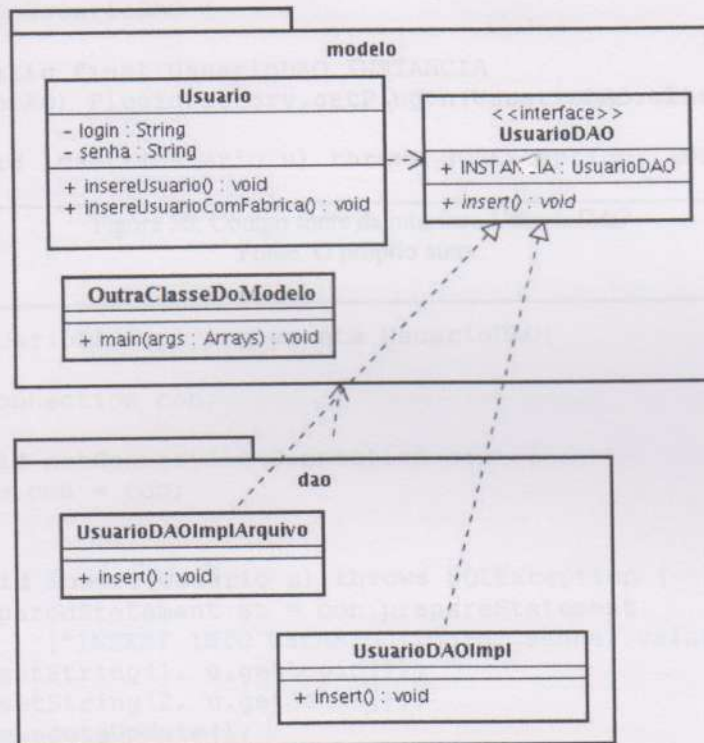


Figura 18: O diagrama de classes representativo do padrão *Separated Interface*
Fonte: O próprio autor.

```

public class Usuario{

    private String login;

    private String senha;

    public void insereUsuario() throws SQLException, IOException{
        if(login != null && senha != null){
            UsuarioDAO dao = new UsuarioDAOImpl();
            dao.insert(this);
        }
    }

    public void insereUsuarioComPlugin() throws SQLException,
                                           IOException{
        if(login != null && senha != null){
            UsuarioDAO.INSTANCIA.insert(this);
        }
    }

    // outros métodos
}
  
```

Figura 19: Código fonte da classe *Usuario*
Fonte: O próprio autor.

```

public interface UsuarioDAO {

    public static final UsuarioDAO INSTANCIA
    = (UsuarioDAO) PluginFactory.getPlugin(UsuarioDAO.class);

    public void insert(Usuario u) throws SQLException, IOException ;

}

```

Figura 20: Código fonte da interface *UsuarioDAO*
Fonte: O próprio autor.

```

public class UsuarioDAOImpl implements UsuarioDAO{

    private Connection con;

    public void setConnection(Connection con) {
        this.con = con;
    }

    public void insert(Usuario u) throws SQLException {
        PreparedStatement st = con.prepareStatement
        ("INSERT INTO USUARIO (login, senha) values (?, ?)");
        st.setString(1, u.getLogin());
        st.setString(2, u.getSenha());
        st.executeUpdate();
        st.close();
    }

}

```

Figura 21: Código fonte da classe *UsuarioDAOImpl*
Fonte: O próprio autor.

Umas das dependências que ainda existe no exemplo acima, é a do cliente com a classe que está sendo instanciada. A classe cliente ainda precisa saber da existência da classe que se está instanciando. Para diminuir essa dependência é necessário utilizar algum padrão de criação que resolva esse problema. *Factory* (GAMMA et al, 1994), *Plugin* e *Dependency Injection* são alguns padrões que diminuem a dependência na hora de se instanciar novos objetos. Esses padrões serão discutidos mais tarde nesse mesmo capítulo (seções 3.7, 3.8 e 3.9 respectivamente).

Pode-se notar que esse padrão diminui a dependência entre partes ou camadas de um sistema, mas aumenta a quantidade de código que um desenvolvedor precisa manter. Levando em consideração que será preciso usar outros padrões, como o *Factory*, para tornar mais independente o código cliente das classes que se está precisando acessar. Para aumentar a produtividade é recomendável criar *Separated Interface* apenas para partes do sistema em que é preciso diminuir o acoplamento. Separar a interface da sua implementação é um *refactoring* simples na arquitetura de um sistema que pode ser feito apenas quando surgir a necessidade.

3.5 TEMPLATE METHOD

3.5.1 Conceito

O padrão *Template Method* garante a ordem em que ações devem ser executadas para realizar alguma tarefa específica. O padrão consiste em se encapsular cada ação individual, que não se sabe como executar, como um método abstrato, e definir uma classe abstrata que executa os métodos abstratos criados na ordem correta. A classe concreta que estender a classe abstrata, referida anteriormente, será responsável por implementar os métodos abstratos para executar as ações. Ou seja, a classe abstrata sabe a ordem das ações e a classe que a implementar conhece que ações devem ser executadas.

A centralização do fluxo da lógica na superclasse abstrata é um exemplo de Inversão de Controle (IoC). Geralmente definem-se APIs e chamam-se os métodos dessa API, com *Template Method* o método da superclasse abstrata (uma API) é responsável por chamar os métodos das classes filhas. Inversão de Controle é descrito em mais detalhes mais adiante nesse mesmo capítulo, na subseção 3.9 que trata do padrão Injeção de Dependências.

3.5.2 Exemplo de uso do padrão *Template Method*

Nessa subseção é mostrado um exemplo simples, baseado no exemplo do livro *Head First Design Patterns* (FREEMAN, E. et al, [199?], p.275), de uma situação onde o padrão *Template Method* pode ser útil. Suponha que um *Usuario*, Figura 19, precise preparar um café para beber. Existem dois tipos de cafés que podem ser preparados: Café Expresso e Cappuccino. Para se preparar esses dois tipos de café, é necessário seguir alguns passos mostrados na Figura 22. Com o objetivo de diminuir a duplicação de código e garantir que todas as classes clientes sigam os passos na ordem correta, criou-se a classe *PrepararBebidaBaseCafe*, Figura 22, e colocou um método *final* que chame os métodos do preparo do café na ordem correta, o método deve ser *final* para garantir que quem herdar a classe *PrepararBebidaBaseCafe* não o sobrescreva. Os métodos que são comuns no preparo do café foram implementados nessa mesma classe, os outros métodos que são específicos para cada tipo de café foram declarados **abstratos** e **protegidos**, para ser obrigados a serem implementados nas classes filhas e só poderem ser vistos por quem herdar a classe *PrepararBebidaBaseCafe*, respectivamente⁹.

⁹ Desconsiderando que métodos protegidos podem ser acessados sem herança se as classes estiverem no mesmo pacote

```

public abstract class PrepararBebidaBaseCafe {

    public final void realizaPreparo() {
        montar();
        adicionaCodimentos();
        adicionaAguaQuente();
        preparo();
        adicionaCodimentosFinais();
        tomar();
    }

    protected void tomar(){
        System.out.println("Tomando café");
    }

    protected abstract void montar();

    protected abstract void adicionaCodimentos();

    protected abstract void preparo();

    protected abstract void adicionaAguaQuente();

    protected abstract void adicionaCodimentosFinais();
}

```

Figura 22: Código fonte da classe *PrepararBebidaBaseCafe*
Fonte: O próprio autor.

Quem herdar essa classe vai ficar obrigado a implementar os métodos abstratos, isso é, as classes filhas conhecem o que deve ser feito. Exemplo de classes que herdam de *PrepararBebidaBaseCafe* são mostrados nas Figuras 23 e 24.

```

public class CafeExpresso extends PrepararBebidaBaseCafe {

    protected void adicionaAguaQuente() {
        System.out.println("Água quente passa sobre pressão no pó de
        café");
    }

    protected void adicionaCodimentos() {}

    protected void adicionaCodimentosFinais() {}

    protected void montar() {
        System.out.println("Torra e Moe o grão de café");
    }

    protected void preparo(){
        System.out.println("O café vai direto para a xícara");
    }
}

```

Figura 23: Código fonte da classe *CafeExpresso*
Fonte: O próprio autor.

```

public class Cappuccino extends PrepararBebidaBaseCafe{

    protected void adicionaAguaQuente() {
        System.out.println("Coloque a água fervente");
    }

    protected void adicionaCodimentos() {
        System.out.println("Adiciona açúcar, chocolate em pó,
bicarbonato de sódio, canela em pó");
    }

    protected void adicionaCodimentosFinais() {
        System.out.println("Colocar por cima chantilly e cobertura de
chocolate");
    }

    protected void montar() {
        System.out.println("Misture no liquidificador o leite em pó e o
café solúvel (para triturar o café)");
    }

    protected void preparo() {
        System.out.println("Misturar até ficar cremoso");
    }
}

```

Figura 24: Código fonte da classe *Cappuccino*
Fonte: O próprio autor.

Como o método que vai ser chamado é o método *final* na classe pai (o método que contém a sequência das tarefas e não deve ser sobrescrito nas classes filhas da classe que o contém), a sequência das tarefas realizadas para se preparar os tipos de café é seguida, isso é, a classe pai não conhece como fazer, mas sabe a ordem em que deve ser feito. Um exemplo de um cliente para o padrão *Template Method* é mostrado na Figura 24.

```

public class Usuario{

    // outros metodos e atributos

    public void horaIntervalo(){
        Cappuccino cafel = new Cappuccino();
        cafel.realizaPreparo();

        System.out.println("\n");

        CafeExpresso cafe2 = new CafeExpresso();
        cafe2.realizaPreparo();
    }
}

```

Figura 25: Código fonte da classe *Usuario* chamando o método *realizaPreparo* da classe *PrepararBebidaBaseCafe*
Fonte: O próprio autor.

```

>Misture no liquidificador o leite em pó e o café solúvel (para triturar o
café)
>Adiciona açúcar, chocolate em pó, bicarbonato de sódio, canela em pó
>Coloque a água fervente
>Misturar até ficar cremoso
>Colocar por cima chantilly e cobertura de chocolate
>Tomando café
>
>Torra e Moe o grão de café
>Água quente passa sobre pressão no pó de café
>O café vai direto para a xícara
>Tomando café

```

Figura 26: Saída resultante da execução do código da Figura 25.

3.6 STRATEGY

3.6.1 Conceito

O padrão *Strategy* é uma alternativa ao padrão *Template Method*, onde além de se poder garantir a seqüência das ações pode-se ter seqüências diferentes e essas serem escolhidas em tempo de execução.

O padrão *Strategy* é mais flexível do que o padrão *Template Method*, porém essa flexibilidade tem um preço, ele é um pouco mais complexo e exige um maior trabalho para ser implementado. Outro ponto fraco do padrão *Strategy* é que existe uma dependência do cliente em relação às classes que implementam a estratégia, como pode-se perceber no diagrama da Figura 27.

Nesse diagrama tem-se a interface *Strategy* que representa uma estratégia genérica com o método *execute()* que os clientes devem chamar. *Strategy1* e *Strategy2* são duas classes que implementam a interface *Strategy* e constroem as suas respectivas tarefas na seqüência correta sobrescrevendo o método *execute()* da mesma interface. *Context* é uma classe auxiliar que contém uma estratégia que irá ser executada e é a classe por onde o cliente executa a estratégia desejada.

Esse padrão é mais flexível, pois ao contrário do *Template Method*, quem vai usar o padrão não precisa invocar uma classe concreta e uma classe que implemente uma ação individual não precisa herdar de uma superclasse abstrata que contém o *template* da seqüência das ações que se está querendo garantir.

O padrão *Strategy* define uma família de algoritmos, encapsulando cada um como um objeto, e torna esses objetos acessíveis. O padrão *Strategy* deixa os algoritmos muito independentes dos clientes que os usam.

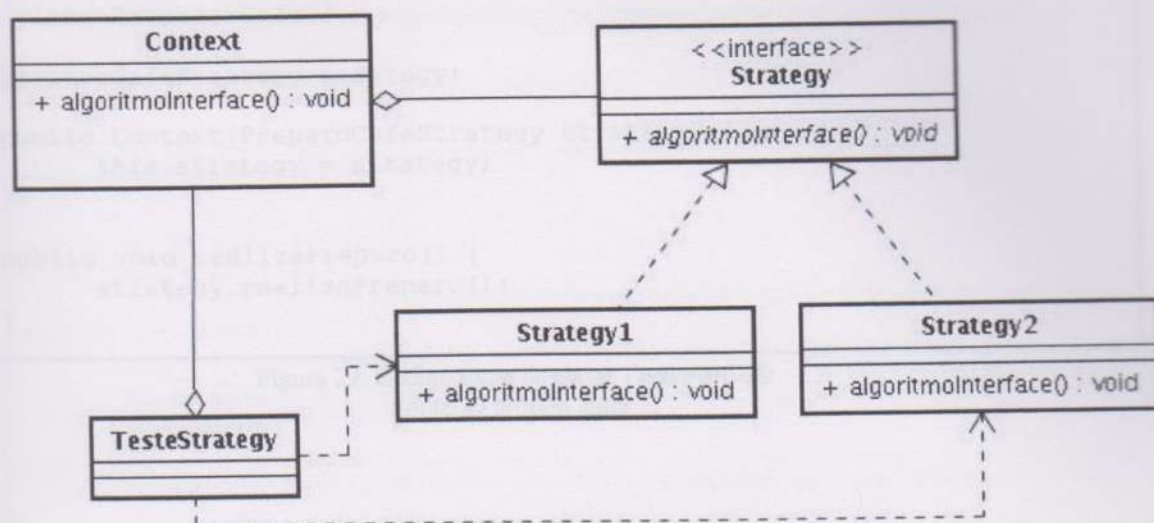


Figura 27: O diagrama de classes representativo do padrão *Strategy*
 Fonte: O próprio autor.

3.6.2 Exemplo de uso do padrão *Strategy*

Nessa subseção é mostrada a resolução do mesmo problema da subseção 3.5, o problema de se executar algoritmos diferentes para preparar tipos de cafés diferentes, mas dessa vez implementada utilizando o padrão *Strategy*.

Na Figura 28 é mostrado o código fonte da interface *PreparoCafeStrategy*, implementada por todas as classes que desejarem implementar alguma estratégia.

```

public interface PreparoCafeStrategy {

    public void realizaPreparo();

}
  
```

Figura 28: Código fonte da interface *PreparoCafeStrategy*
 Fonte: O próprio autor.

Na Figura 29 é mostrado a classe *PrepararCafe*, classe que nesse padrão é responsável por conter a estratégia que se deseja usar.

```

public class PrepararCafe {

    PreparoCafeStrategy strategy;

    public Context(PreparoCafeStrategy strategy) {
        this.strategy = strategy;
    }

    public void realizaPreparo() {
        strategy.realizaPreparo();
    }
}

```

Figura 29: Código fonte da classe *PrepararCafe*
Fonte: O próprio autor.

Nas Figuras 30 e 31 tem-se as duas classes que implementam as tarefas para se preparar um Café Expresso e um Cappuccino, respectivamente.

```

public class CafeExpressoStrategy implements PreparoCafeStrategy {

    public void realizaPreparo() {
        montar();
        adicionaAguaQuente();
        preparo();
        tomar();
    }

    // definicao dos metodos
}

```

Figura 30: Código fonte da classe *CafeExpressoStrategy*
Fonte: O próprio autor.

```

public class CappuccinoStrategy implements PreparoCafeStrategy {

    public void realizaPreparo() {
        montar();
        adicionaCodimentos();
        adicionaAguaQuente();
        preparo();
        adicionaCodimentosFinais();
        tomar();
    }

    // definicao dos metodos
}

```

Figura 31: Código fonte da classe *CappuccinoStrategy*
Fonte: O próprio autor.

Por último na Figura 32, tem-se o código da classe *Usuario* executando a estratégia desejada.

```
public class Usuario{  
    // outros metodos e atributos  
    PrepararCafe context;  
  
    public void horaIntervalo(){  
        context = new PrepararCafe(new CappuccinoStrategy());  
        context.realizaPreparo();  
    }  
}
```

Figura 32: Código fonte da classe *Usuario* executando a estratégia *CappuccinoStrategy*

3.7 FACTORY

3.7.1 Conceito

Uns dos sinais que um código está bem construído, possui qualidade e que boa parte dos padrões descritos nesse texto buscam é o baixo acoplamento. Idealmente objetos não deveriam conhecer a implementação de outros objetos só a interface¹⁰, esses também é um dos princípios básicos da programação orientada a objetos, o encapsulamento.

Toda vida que for preciso instanciar um objeto, vai-se está instanciando um objeto concreto (objeto que possui além da interface detalhes internos), logo o principio citado acima de somente se conhecer a interface de um objeto está sendo quebrado, pois geralmente é necessário saber detalhes internos de um objeto para se instanciá-lo, como por exemplo, que seu construtor recebe um inteiro como argumento e que um valor válido para esse parâmetro é ser maior que zero.

Outro caso que indica a má qualidade de um código é quando é preciso instanciar uma grande quantidade de objetos de tipos diferentes e os critérios da instanciação só são conhecidos em tempo de execução. Ou seja, o código vai ficar repleto de seleções (*ifs*) e isso implica em código difícil de entender e difícil de mudar, aumentando o custo com manutenção do sistema.

Factory é um padrão que encapsula a criação de objetos (GAMMA et al, 1994). Os cliente dos objetos vão ter uma dependência apenas da classe que implementa o padrão *Factory* e a fábrica (ou *Factory* em inglês) vai depender dos objetos instanciados. Essa

¹⁰ Não confundir com *interface* de Java. Interface nesse contexto é o conjunto de métodos que uma classe declara como públicos, isso é, o que ela deseja que as outras classes tenham acesso.

dependência é menos grave porque vai está concentrada em um único ponto do sistema. *Factory* é um meio termo entre uma instanciação direta de um objeto e usar um padrão que remova completamente essa dependência, como o *Dependency Injection*.

3.7.2 Exemplo do uso do padrão *Factory*

Na Figura 33, é apresentado o código de uma fábrica de DAOs, objetos para acesso a dados que seguem um padrão de projeto cujo detalhamento será realizado no Capítulo 5, para a classe *Usuario*, essa fábrica possui um método estático *getUsuarioDAO()* que retorna uma classe que implemente a interface *UsuarioDAO* de acordo com o tipo passado.

```
public class DAOFactory {

    public static UsuarioDAO getUsuarioDAO(String tipo) {
        if ("arquivo".equals(tipo)) {
            return new UsuarioDAOImplArquivo();
        } else {
            if ("banco".equals(tipo))
                return new UsuarioDAOImpl();
            return null;
        }
    }
}
```

Figura 33: A classe que implementa o padrão *Factory* para DAOs da classe *Usuario*.
Fonte: O próprio autor.

Na Figura 34 é mostrado o esquema de dependências criado entres as classes com o padrão *Factory*.

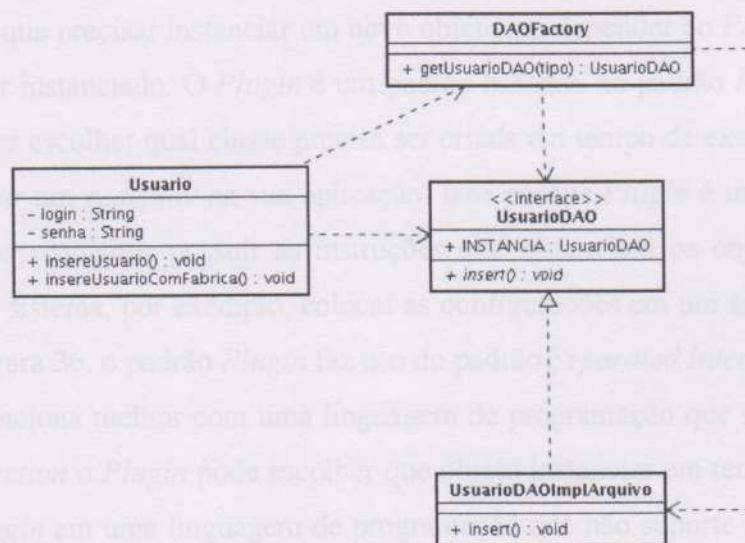


Figura 34: Dependência de classes com o padrão *Factory*
Fonte: O próprio autor.

Utilizar o padrão *Factory* para instanciar objetos faz com que as classes dependam apenas da interface, não importando se a classe que irá implementar *UsuarioDAO* irá salvar em arquivo, ou em um Banco de Dados, para a classe que está utilizando a fábrica, nesse caso a classe *Usuario*, só interessa saber que os suas informações foram salvas. Um exemplo de cliente para a classe *DAOFactory* é mostrado na Figura 35.

```
public class Usuario{
    private String login;
    private String senha;

    public void insereUsuarioComFabrica() throws SQLException,
                                                IOException{
        if(login != null && senha != null){
            UsuarioDAO dao = DAOFactory.getUsuarioDAO("arquivo");
            dao.insert(this);
        }
    }

    // outros métodos
}
```

Figura 35: Código Fonte da classe cliente para a classe *DAOFactory*

3.8 *PLUGIN*

3.8.1 Conceito

Plugin é um padrão que tem por finalidade diminuir a dependência da criação de objetos. A classe que precisar instanciar um novo objeto vai depender do *Plugin* e não mais do objeto que vai ser instanciado. O *Plugin* é um padrão baseado no padrão *Factory*, mas com a diferença de poder escolher qual classe precisa ser criada em tempo de execução sem precisar re-compilar ou dar um *redploy* na sua aplicação. Isso porque *Plugin* é uma fábrica que tem como um de seus requisitos possuir as instruções que instanciam os objetos em um único ponto externo ao sistema, por exemplo, colocar as configurações em um arquivo texto. Como se pode vê na Figura 36, o padrão *Plugin* faz uso do padrão *Separated Interface*.

Plugin funciona melhor com uma linguagem de programação que suporte *reflection*¹¹, porque com *reflection* o *Plugin* pode escolher que objeto instanciar em tempo de execução. É possível usar *Plugin* em uma linguagem de programação que não suporte *reflection*, mas será

¹¹ Reflection permite um programa Java obter informações sobre suas propriedades e estruturas em tempo de execução. Por exemplo, os nomes dos métodos de uma determinada classe.

preciso criar, dentro do *Plugin*, uma lógica condicional para que ele possa escolher qual classe instanciar em determinado momento.

3.8.2 Exemplo de uso do padrão *Plugin*

Neste exemplo um objeto *Usuario* vai persistir seu estado, para isso ele precisa instanciar um objeto da camada de persistência responsável por essa tarefa. Como um objeto do tipo *Usuario* pode ser salvo de diferentes formas (nesse exemplo são mostradas duas formas, em um arquivo e em um Banco de Dados) e a escolha dessa maneira de salvar muda constantemente, seria improdutivo ter que re-compilar o código da aplicação toda vida que se desejar alterar a implementação dessa maneira de salvar.

Para resolver esse problema foi criada a estrutura mostrada na Figura 36, que implementa o padrão *Plugin*.

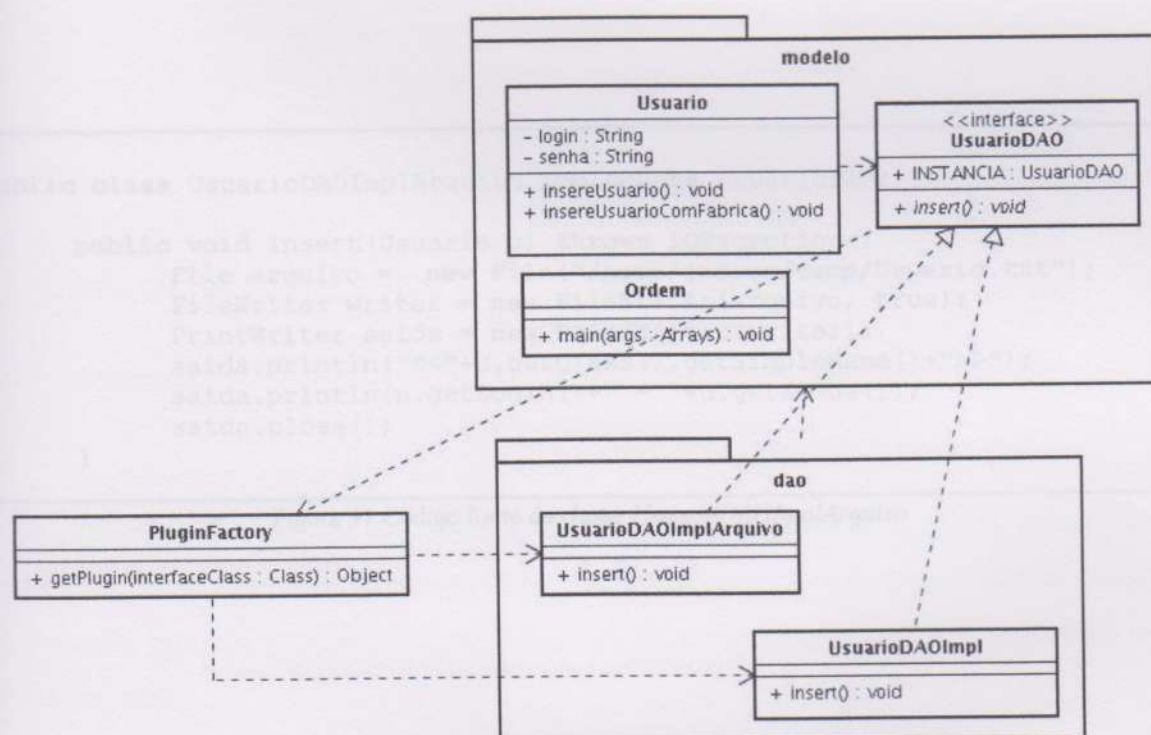


Figura 36: O diagrama de classes representativo do padrão *Plugin*
Fonte: O próprio autor.

O objeto da classe *Usuario* chama o método da classe *UsuarioDAO* através de uma classe *PluginFactory* que esta interface possui (Figura 20). A classe *PluginFactory* ler um arquivo de propriedade onde está configurado qual implementação de interface *UsuarioDAO*

precisa ser instanciada. O *PluginFactory* realiza essa instanciação através de *reflection* e retorna para o objeto *Usuario* uma implementação da interface *UsuarioDAO*. A classe usuário chama o método *insert()* da interface *UsuarioDAO* que, utilizando polimorfismo, chama o método *insert()* do objeto que implementa a interface *UsuarioDAO* e que foi instanciado pelo *Plugin*. Que tipo de objeto foi retornado não importa para a classe *Usuario*, desde que implemente a interface *UsuarioDAO* e possua um método *insert()* para persistir as informações desejadas. Na Figura 37 é mostrado um exemplo de uma classe que pode ser instanciada pelo *Plugin* e que guarda as informações de um objeto *Usuario* em um arquivo.

Repare no código da classe *Usuario*, Figura 19, que esta não toma conhecimento das classes *UsuarioDAOImpl* (Figura 21) ou *UsuarioDAOImplArquivo* (Figura 37) e nem de qual classe está sendo chamada. O código da interface *UsuarioDAO* é mostrado na Figura 20 e o código do *PluginFactory* na Figura 38.

```
public class UsuarioDAOImplArquivo implements UsuarioDAO{

    public void insert(Usuario u) throws IOException {
        File arquivo = new File("/home/jadson/temp/Usuario.txt");
        FileWriter writer = new FileWriter(arquivo, true);
        PrintWriter saida = new PrintWriter(writer);
        saida.println("<<<" + u.getClass().getSimpleName() + ">>>");
        saida.println(u.getLogin() + " - " + u.getSenha());
        saida.close();
    }
}
```

Figura 37 Código fonte da classe *UsuarioDAOImplArquivo*

```

public class PluginFactory {

    private static Properties props = new Properties();

    static{
        try{
            props.load(PluginFactory.class.getResourceAsStream
                ("/home/jadson/temp/plugins.properties"));
        }catch(Exception ex){
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static Object getPlugin(Class interfaceClass){
        String implName= props.getProperty(interfaceClass.getName());

        if(implName == null){
            throw new RuntimeException("Implemenatacao nao existe: "
                +interfaceClass.getName());
        }
        try{
            return Class.forName(implName).newInstance();
        }catch(Exception ex){
            throw new RuntimeException("Nao conseguiu criar uma "
                +"nova instancia: "
                +interfaceClass.getName());
        }
    }
}

```

Figura 38: Código fonte da classe *PluginFactory*
 Fonte: O próprio autor.

3.9 DEPENDENCY INJECTION

3.9.1 Conceito

Dependency Injection é um padrão que permite manter baixo nível de acoplamento entre diferentes módulos de um sistema. Com o uso desse padrão, o acesso a recursos de outros módulos é realizado por um container, o programador precisa apenas dizer ao container os recursos que ele precisa e ele se encarrega de criar esses recursos e colocá-los onde deveriam estar.

Ao invés do programador se encarregar de criar uma instância de uma classe para ele usar (o caminho mais óbvio), alguém faz esse trabalho e entrega a instância para ele (inversão da lógica de programação). Esse mecanismo é chamado de Inversão de Controle (IoC).

Com injeção, não existe pedido explícito, o serviço simplesmente está pronto quando precisa ser usado - daí vem a inversão de controle (IoC). Ou seja, *Dependency Injection* é um

tipo específico de Inversão de Controle. Inversão de controle é um conceito mais antigo e é uma característica comum em diversos *frameworks*.

Um dos fundamentos da orientação a objetos diz que idealmente os objetos não devem conhecer a implementação de outros objetos, apenas a interface, isso é o que se chama de encapsulamento. Como pode-se vê na Figura 39, *Dependency Injection*, no que diz respeito ao encapsulamento, é um padrão completo, pois o cliente enxerga apenas a interface do componente ou serviço que ele está tentando usar. Mas *Dependency Injection* tem seu preço. Fowler em seu artigo *Inversion of Control Containers and the Dependency Injection pattern*, diz que *Dependency Injection* “tende a ser difícil de entender e leva a problemas quando tentamos fazer o *debug*”.

Em geral, *Dependency Injection* deve ser usada para realizar comunicação entre camadas diferentes do seu sistema, por exemplo, um objeto do modelo instanciando um objeto da camada de infra-estrutura. Se for um objeto da camada do modelo chamando outro objeto do modelo, deve-se usar instanciação direta ou no máximo o padrão *Fábrica* quando se tratar de algo mais complexo. Outra situação em que *Dependency Injection* pode ser usada é quando se criam classes para serem usadas em múltiplas aplicações.

Existem três tipos de *Dependency Injection*: injeção pelo construtor, injeção por métodos *sets* e injeção por interfaces. Os dois mais usados são injeção pelo construtor (que utiliza o construtor da classe para injetar tudo o que ela precisa) e injeção por métodos *sets* (que de maneira semelhante utiliza os métodos *set* para injetar as dependências). Isso porque a Injeção por Interfaces é mais invasiva, já que se tem que escrever uma interface para cada dado que se deseja injetar.

3.9.2 Exemplo de uso do padrão *Dependency Injection*

Nesta subseção mostra-se um exemplo simples do uso da Injeção de Dependência usando-se o *framework* Spring (SPRING, 2007).

Na seção 3.4 onde é descrito o padrão *Separated Interface*, foi falado que esse padrão é usado para diminuir a dependência entre camadas de uma aplicação, visto que um objeto cliente só vai conhecer e tratar como a interface do objeto que vai prestar o serviço. Foi mostrado também que esse padrão funciona muito bem quando se está chamando um

¹² Componente é sempre aquele que foi criado para ser usado por terceiros sem que os mesmos tenham acesso à sua implementação. Um Serviço é um componente que é chamado remotamente e o cliente não vai se usar apenas o nome Serviço para identificar a entidade a interface, mas vai ter que saber o nome do Serviço e vai também para Componentes.

componente ou serviço¹², mas que ele ainda possui uma dependência na hora de se instanciar o serviço, como se pode perceber na Figura 39.

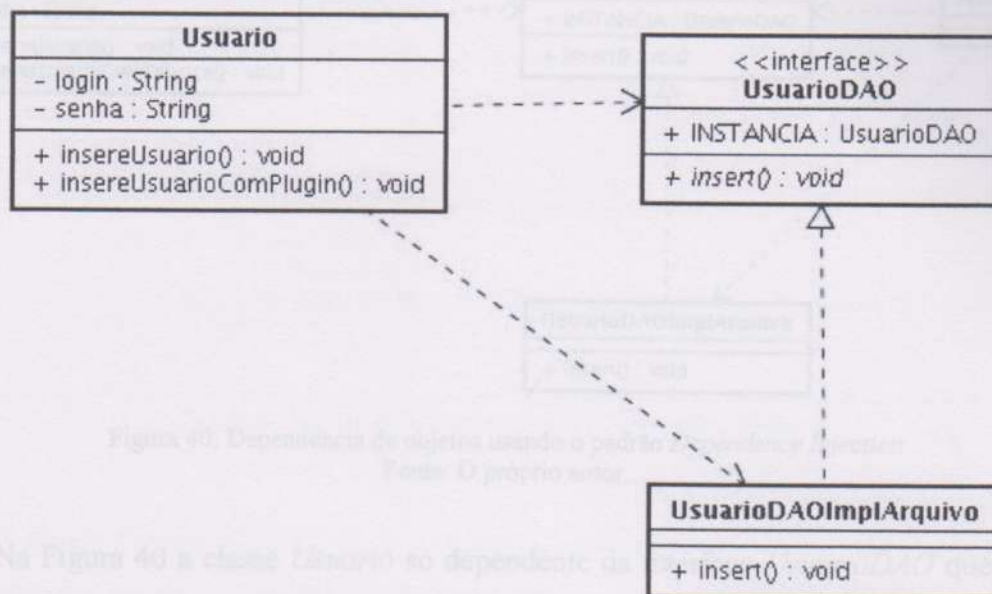


Figura 39: Dependência de objetos usando instanciação direta.

Fonte: O próprio autor.

Um objeto *Usuario* depende da interface *UsuarioDAO* e da classe *UsuarioDAOImplArquivo*. O padrão *Dependency Injection* pode ser usado para resolver completamente essa dependência, como se pode vê na Figura 40.

¹² Componente é algum módulo que foi criado para ser usado por terceiros sem que se precise alterá-lo ou conhecer a sua implementação. Um Serviço é um componente que é chamado remotamente. No restante do texto vai ser usado apenas o termo Serviço para otimizar a escrita e a leitura, mas o que foi falado a respeito de Serviços vale também para Componentes.

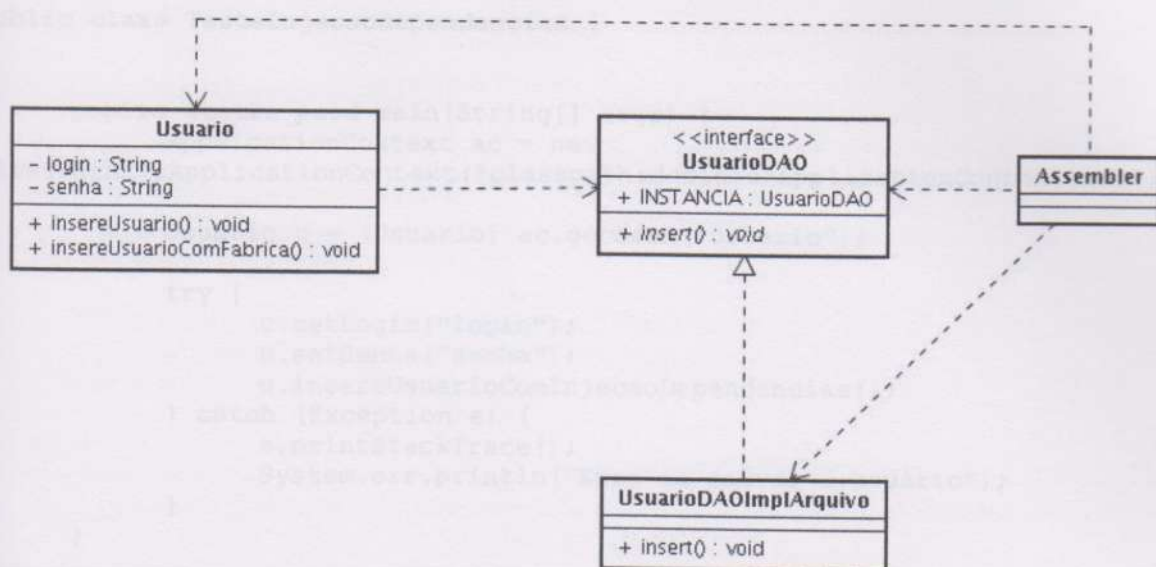


Figura 40: Dependência de objetos usando o padrão *Dependency Injection*
Fonte: O próprio autor.

Na Figura 40 a classe *Usuario* só depende da interface *UsuarioDAO* que é o mais correto, a instanciação de um objeto da classe *UsuarioDAOImplArquivo* é realizada por um objeto da classe *Assembler* (objeto criado pelo Framework de injeção de dependências que se estiver usando, nesse caso em particular o Spring). Esse objeto *Assembler* é que vai depender (conhecer a implementação) das demais classes envolvidas, mas isso não é problema, o *framework* pode depender da sua aplicação, a sua aplicação é que jamais deveria depender do *framework* usado.

Na Figura 41 é mostrado o código fonte de uma classe de teste que cria um objeto *Usuario* usando os recursos do *framework* Spring. Para executar esse exemplo será preciso adicionar dois arquivos *JAR* ao *classpath* da sua aplicação: *spring.jar* e *commons-logging.jar*. Esses arquivos podem ser baixados do site do Spring (SPRING, 2007).

No arquivo *applicationContext.xml*¹³, Figura 43, estão configuradas todas as dependências da classe *Usuario*, nesse exemplo a classe *UsuarioDAOImplArquivo*. Quando é executado o comando `ac.getBean("usuario")`, já é instanciado um objeto da classe *Usuario* com todas as suas dependências injetadas pelo Spring.

¹³ Nesse exemplo o arquivo *applicationContext.xml* foi colocado no mesmo pacote das classes onde se deseja usar o padrão de injeção de dependências.

```

public class TesteInjecaoDependencias {

    public static void main(String[] args) {
        ApplicationContext ac = new
        ClassPathXmlApplicationContext("classpath:dominio/applicationContext.xml");

        Usuario u = (Usuario) ac.getBean("usuario");

        try {
            u.setLogin("login");
            u.setSenha("senha");
            u.inserirUsuarioComInjecaoDependencias();
        } catch (Exception e) {
            e.printStackTrace();
            System.err.println("Erro ao salvar o usuário");
        }
    }
}

```

Figura 41: Classe para teste do padrão *Dependency Injection*
 Fonte: O próprio autor.

A classe *Usuario*, Figura 42, é a mesma classe que vem sendo usados nos exemplos desse trabalho, apenas acrescida do método *insereUsuarioComInjecaoDependencias()* (os outros métodos e atributos foram omitidos, mas podem ser vistos na Figura 19). Percebe-se a simplicidade do método, apenas usa a referência da interface *UsuarioDAO* que possui.

O código da interface *UsuarioDAO* e a classe *UsuarioDAOImplArquivo* foram mostradas nas Figuras 20 e 21 respectivamente.

```

public class Usuario{

    //vai ser injetado
    private UsuarioDAO dao;

    public Usuario(){

    }

    public Usuario(String login, String senha){
        this.login = login;
        this.senha = senha;
    }

    public void inserirUsuarioComInjecaoDependencias(){
        if(login != null && senha != null){
            dao.insert(this);
        }
    }
}

```

Figura 42: Classe *Usuario* com o Método que Usa o Padrão *Dependency Injection*
 Fonte: O próprio autor.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="daoArquivo" class="dao.UsuarioDAOImplArquivo"/>

    <bean id="usuario" class="modelo.Usuario">
        <property name="dao" ref="daoArquivo"/>
    </bean>

</beans>
```

Figura 43: O Arquivo *ApplicationContext.xml* do *Spring* para Configurar as dependências.
Fonte: O próprio autor.

Neste capítulo foram descritos os padrões mais gerais que não estão associados a uma camada específica ou que formam a base para padrões mais complexos. Nos próximos capítulos desse trabalho vai-se começar a descrever os padrões que pertencem a uma das quanto principais camadas que estar-se tomando como base nesse trabalho (Apresentação, Aplicação, Modelo e Persistência). A começar pela camada que recebe o maior foco no desenvolvimento de sistemas corporativos, a camada de modelo.

4 PADRÃO DA CAMADA DE MODELO

Neste capítulo descreve-se a parte mais importante da modelagem de um software, como estruturar a lógica de negócio e os dados de uma aplicação.

4.1 O PADRÃO *DOMAIN MODEL*

Domínio é um conjunto de “entidades” ou “elementos” que descrevem a estrutura e comportamento de um problema no “mundo real”. Por exemplo: o domínio de um sistema financeiro são os cálculos de impostos, tarifas, juros, entre outros; o domínio de um programa jurídico são as varas, tramitações, processos, entre outros. O Modelo é uma abstração do domínio para resolver um problema. Como um problema pode ter várias soluções, um domínio pode ser modelado de maneiras diferentes, mas isso não significa que cada problema tem que ser solucionado por um modelo exclusivo. Um modelo pode resolver mais de um problema, o que muda neste caso será o significado as entidades do modelo, chamado de contexto (contexto é melhor explicado mais adiante nesse trabalho, na subseção 4.3.4).

O padrão *Domain Model* consiste em modelar o domínio da sua aplicação através de uma rede de objetos interconectados que possuem os dados e as regras de negócio do sistema. O *Domain Model* orientado a objetos parece muito com o modelo Entidade-Relacionamento de um Banco de Dados, mas com algumas diferenças: o *Domain Model* mistura dados e regras de negócio, tem atributos multivalorados, associações complexas e herança.

Pode-se optar por se projetar um *Domain Model* mais simples ou mais complexo. Quando mais simples for projetado o *Domain Model* mais parecido com um modelo Entidade-Relacionamento de um Banco de Dados ele será.

Como as regras de negócio de uma corporação podem mudar, e normalmente mudam com uma frequência considerável, é importante estar preparado para modificar o modelo que foi criado. Aconselha-se separar o modelo criado do restante do código da aplicação. Isso é feito colocando-se o modelo de uma aplicação em uma camada exclusiva para ele. A maioria dos padrões de projeto visa minimizar o acoplamento entre essa camada e as demais camadas de um sistema. Conceitos ortogonais¹⁴ como autenticação, transações, *logs*, entre outros não

¹⁴ Conceitos que divergem entre seus objetivos. Conceitos que não se misturam, não possuem pontos em comum. Em geometria, retas ortogonais são retas que formam um ângulo reto entre si e não possuem um ponto em comum.

devem, quando a tecnologia permite, estar implementados junto com regras de negócio, isto é, o modelo deve ser completamente independentes deles.

Segundo vários autores, entre eles (CALÇADO, 2007) em seu artigo *Fantoches*, e Martin Fowler, em seu livro *Patterns of Enterprise Application Architecture* (FOWLER, et al, 2002), deve-se modelar o *Domain Model* o mais próximo possível de como as coisas acontecem no mundo real. Isso facilita a comunicação entre os desenvolvedores e os usuários do sistema, pois vão ser usados na modelagem elementos que o usuário está acostumado a utilizar no seu dia-a-dia.

Também ajuda a medir a flexibilidade do seu sistema, dando ao sistema uma flexibilidade próxima da real. Um grande problema encontrado na modelagem de um sistema é determinar o quanto flexível o seu modelo deve ser. Pode-se gastar muito tempo no desenvolvimento de um sistema tentando-se criar uma solução flexível, altamente genérica, que qualquer mudança no domínio da aplicação resulte em uma mínima mudança no modelo projetado. Mas as vezes, a aplicação não necessita de tal flexibilidade, construindo o seu modelo o mais próximo possível dos problemas encontrados no mundo real o sistema ficará com uma flexibilidade próxima da flexibilidade do negócio que se está modelando. Uma grande mudança no código da aplicação será necessária somente se existirem grandes mudanças nas regras de negócio, e isso é compreensível e aceito com mais facilidade pelos clientes e patrocinadores do sistema (um outro problema encontrado na área de desenvolvimento de software é que os clientes e patrocinadores geralmente não estão satisfeitos com os prazos e custos de implementar novas funcionalidades ou mudar outras já existentes, isso porque eles não conseguem entender o quanto pode ser complicado implementar aquela idéia simples que eles tiveram sobre o sistema, como um *Domain Model* bem construído deixa o sistema com uma flexibilidade próxima do domínio da aplicação e tende utiliza termos que o cliente e o patrocinador estão acostumados, é mais fácil para os desenvolvedores do sistema explicar porque aqueles custos e prazos são necessários e o outro lado tende a aceitar estas previsões tendo uma certeza maior que não estão sendo ludibriados pelos desenvolvedores).

Outra vantagem de se usar o padrão *Domain Model* é de realmente utilizar as vantagens do paradigma da orientação a objetos. Uma das bases desse paradigma é o encapsulamento, objetos devem possuir dados privados e ser os responsáveis por manipular seus próprios dados além de garantir a consistência dos mesmos. Um objeto em princípio não deve conhecer a implementação de outro objeto. Colocar a implementação das regras de

negócio nas classes que realmente possuem os dados que serão manipulados por essas regras evita duplicidade e reduz o acoplamento entre os diferentes objetos.

Ainda é bastante utilizada a modelagem onde se separa os dados do sistema da sua lógica de negócio. Críticos dessa modelagem alegam que quando se cria objetos apenas com dados e outros objetos apenas com a lógica de negócio está se programando “proceduralmente”, apenas utilizando objetos. Martin Fowler é um dos que mais combate essa técnica conforme descrito em seu artigo *AnemicDomainModel*. Phillip Calçado mostra em seu artigo *Fantoches* um exemplo onde nota-se com bastante clareza a diferença entre Programação Procedural e Programação Orientada a Objetos. Calçado ([200?], p.5) ainda cita algumas características dos sistemas que são modelados separando a lógica de negócio dos dados da aplicação:

É o tipo de sistema onde reusabilidade é mantida com copiar-e-colar, já que seus métodos são “gordos” demais e são tão especializados que simplesmente não servem para nada além do que foram concebidos inicialmente. Outro sintoma deste tipo de design são métodos com muitas linhas de código.

Em um domínio lógico bem elaborado muitas classes interagem para realizarem até mesmo uma simples tarefa. Não existe o que Phillip Calçado chama em seu artigo *Fantoches* de “mestres dos fantoches”, um objeto que contém todas as regras de negócio e manipula objetos do domínio que possuem apenas dados.

4.2 COMPARANDO UM MODELO ANÊMICO COM O PADRÃO *DOMAIN MODEL*

Nesta seção é mostrado um exemplo de uma aplicação de registro de chamada modelada de duas maneiras distintas, usando um “modelo anêmico” e usando o padrão *Domain Model*. É um exemplo simples onde um solicitante relata algum problema e o sistema armazena esse problema até que alguém o solucione.

A implementação utilizando um modelo anêmico é mostrado nas Figuras 44, 45, 46, 47 e 48.

```
public class Atendente {  
  
    private String nome;  
    // sets e gets  
}
```

Figura 44: Classe Atendente para um Modelo Anêmico.

Fonte: O próprio autor.

```

public class Chamada {

    private Date data;

    private List<Problema> problemas = new ArrayList<Problema>();

    private Atendente atendente;

    // sets e gets

}

```

Figura 45: Classe Chamada para um Modelo Anêmico.
Fonte: O próprio autor.

```

public class Problema {

    private String descricao;

    private String solucao;

    //sets e gets

}

```

Figura 46: Classe Problema para um Modelo Anêmico.
Fonte: O próprio autor.

```

public class SistemaChamadaNegocio {

    static List<Chamada> listaChamadas = new ArrayList<Chamada>();

    public static void main(String[] args) {
        List<Problema> listaProblemas = new ArrayList<Problema>();

        Problema p1 = new Problema();
        p1.setDescricao("computador nao liga");
        Problema p2 = new Problema();
        p2.setDescricao("rede fora do ar");

        listaProblemas.add(p1);
        listaProblemas.add(p2);

        Chamada c = new Chamada();
        c.setData(new Date());
        c.setProblemas(listaProblemas);

        listaChamadas.add(c);
        System.out.println("chamada fechada: "+isChamadaFechada(c));

        imprimirChamada(c);
        resolverProblemas(c);

        System.out.println("chamada fechada: "+isChamadaFechada(c));

        imprimirChamada(c);
    }
}

```

Figura 47: Classe SistemaChamadaNegocio para um Modelo Anêmico.
Fonte: O próprio autor.

```

public static void resolverProblemas(Chamada c){
    Atendente est = new Atendente();

    est.setNome("Estagiário");

    for (Problema p : c.getProblemas()) {
        p.setSolucao("Problema solucionado");
    }

    c.setAtendente(est);
}

public static void imprimirChamada(Chamada c){
    System.out.println("data da chamada: "+c.getData());
    for (Problema p : c.getProblemas()) {
        System.out.println(p.getDescricao()
            +" - "+p.getSolucao());
    }

    if(isChamadaFechada(c)){
        System.out.println(" >> solucionado por: ");
        System.out.println(c.getAtendente().getNome());
    }
}

public static boolean isChamadaFechada(Chamada c){
    for (Problema problema : c.getProblemas()) {
        if(!(problema.getSolucao() != null))
            return false;
    }
    if(c.getAtendente() == null)
        return false;

    return true;
}

```

Figura 48: Classe SistemaChamadaNegocio e para um Modelo Anêmico continuação.
Fonte: O próprio autor.

Na Figura 49 é mostrado o Diagrama de Seqüência dos objetos do sistema de chamadas utilizando um modelo de domínio anêmico. Nota-se que todas as chamadas de métodos são realizadas por um único objeto, nesse caso *SistemaChamadaNegocio*, e que ou são chamadas de instanciação de novos objetos ou são chamadas para acessar os dados das outras classes. Isso é, os outros objetos são objetos “burros” que funcionam como as antigas estruturas de C e C++;

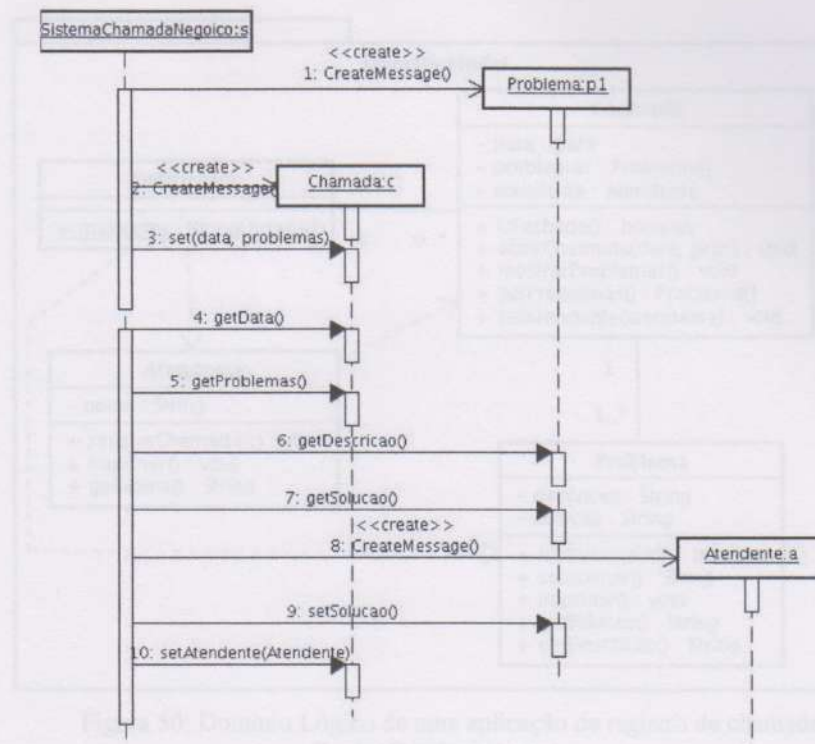


Figura 49: Diagrama de sequência de um Modelo Anêmico.
Fonte: O próprio autor.

A Figura 50 mostra o diagrama de classes da aplicação quando ela é modelada utilizando o *Domain Model*. Um modelo muito simples constituído apenas de quatro classes. Um solicitante pode abrir de zero a várias chamadas, uma chamada pode ter de um a vários problemas.

Um objeto da classe *Chamada* sabe quais informações sobre uma chamada devem ser impressas, mas não deveria saber quais informações de um objeto da classe *Atendente* precisam ser mostradas, logo em vez de obter os dados do atendente da chamada e imprimi-los, quebrando o encapsulamento, simplesmente o objeto da classe *Chamada* vai chamar o método *imprimir* da classe *Atendente* e deixar que esse método, que possui a competência para tal, decida o que deve ser impresso ou não.

Como resultado, em um *Domain Model* bem projetado, como já foi dito anteriormente, muitas classes interagem para realizarem até mesmo uma simples tarefa de imprimir algumas linhas de texto. Podemos notar isso no Diagrama de Sequência da Figura 46, todos os objetos interagem durante uma requisição para realizar a tarefa designada. Não existe um objeto que está presente durante toda a requisição, ou seja, pois possui o controle as demais as demais os objetos e o componente que possui toda a lógica de negócio.

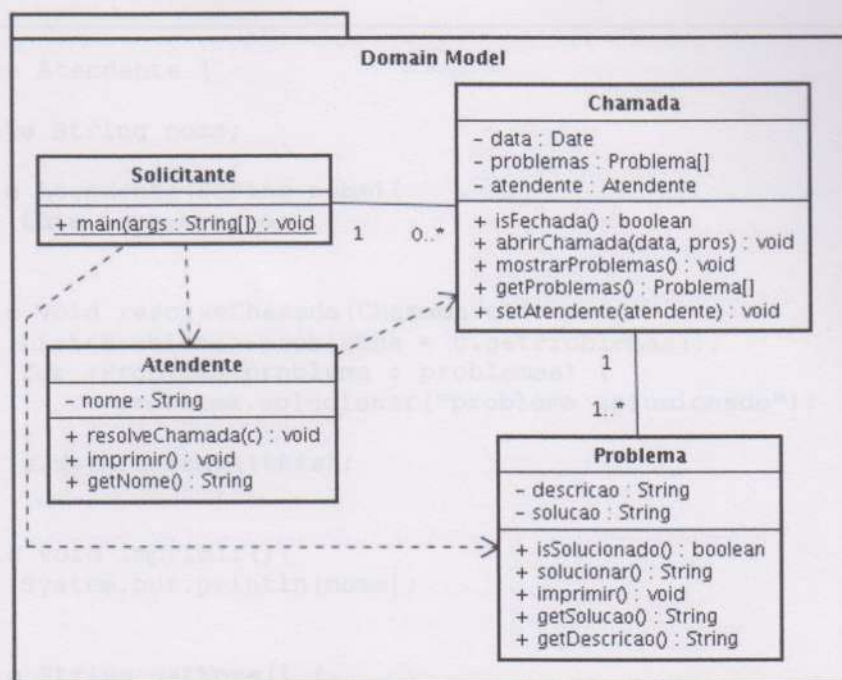


Figura 50: Domínio Lógico de uma aplicação de registro de chamadas
Fonte: O próprio autor.

Nas Figuras 51, 52, 53, 54 e 55 pode-se ver a implementação das classes mostradas na Figura 50. Nota-se que em um *Domain Model*, os métodos com as regras de negócio estão nas classes que deveriam possuir responsabilidade para executá-los. Por exemplo, no modelo projetado nesse exemplo, quem deve ser responsável por resolver uma chamada é um atendente, logo o método `resolveChamada` está na classe *Atendente*, Figura 51. Outro exemplo desses conceitos pode-se vê no método `mostraProblemas` da classe *Chamada*, Figura 52 e 53. Um objeto da classe *Chamada* sabe quais informações sobre uma chamada devem ser impressas, mas não deveria saber quais informações de um objeto da classe *Atendente* precisam ser mostradas, logo em vez de obter os dados do atendente da chamada e imprimi-las, quebrando o encapsulamento, simplesmente o objeto da classe *Chamada* vai chamar o método `imprimir` da classe *Atendente* e deixar que esse método, que possui a competência para tal, decida o que deve ser impresso ou não.

Como resultado, em um *Domain Model* bem projetado, como já foi dito anteriormente, muitas classes interagem para realizarem até mesmo uma simples tarefa de imprimir algumas linhas de texto. Pode-se notar isso no Diagrama de Seqüência da Figura 46, todos os objetos interagem durante uma requisição para realizar a tarefa designada. Não existe um objeto que está onipresente durante toda a requisição, onisciente pois acessa e conhece os dados de todos os objetos e onipotente que possui toda a lógica de negócio.

```

public class Atendente {

    private String nome;

    public Atendente(String nome) {
        this.nome = nome;
    }

    public void resolveChamada(Chamada c) {
        List<Problema> problemas = c.getProblemas();
        for (Problema problema : problemas) {
            problema.solucionar("problema solucionado");
        }
        c.setAtendente(this);
    }

    public void imprimir() {
        System.out.println(nome);
    }

    public String getNome() {
        return nome;
    }
}

```

Figura 51: Classe Atendente para um *Domain Model*.

Fonte: O próprio autor.

```

public class Chamada {

    private Date data;
    private List<Problema> problemas = new ArrayList<Problema>();
    private Atendente atendente;

    public void abriChamada(Date data, List<Problema> pro) {
        this.data = data;
        problemas.addAll(pro);
    }

    public boolean isFechada() {
        for (Problema problema : problemas) {
            if(!problema.isSolucionado())
                return false;
        }
        if(atendente == null)
            return false;

        return true;
    }
}

```

Figura 52: Classe Chamada para um *Domain Model*.

Fonte: O próprio autor.

```

    public void mostraProblemas() {
        System.out.println("data da chamada: "+data);
        for (Problema p : problemas) {
            p.imprimir();
        }
        if(isFechada()){
            System.out.println(" >> solucionado por: ");
            atendente.imprimir();
        }
    }

    public List<Problema> getProblemas() {
        return problemas;
    }

    public void setAtendente(Atendente atendente) {
        this.atendente = atendente;
    }
}

```

Figura 53: Classe Chamada para um *Domain Model* continuação.
Fonte: O próprio autor.

```

public class Problema {

    private String descricao;

    private String solucao;

    public Problema(String descricao){
        this.descricao = descricao;
    }

    public void solucionar(String solucao){
        if(descricao != null)
            this.solucao = solucao;
    }

    public void imprimir(){
        String temp = "problema: "+descricao;
        if(solucacao != null)
            temp += " \n>> solucao dada: "+solucao;
        System.out.println(temp);
    }

    public boolean isSolucionado(){
        return solucao != null;
    }

    public String getSolucao() {
        return solucao;
    }

    public String getDescricao() {
        return descricao;
    }
}

```

Figura 54: Classe Problema para um *Domain Model*.
Fonte: O próprio autor.

```

public class Solicitante {

    private static List<Chamada> listaChamadas = new
    ArrayList<Chamada>();

    public static void main(String[] args) {

        List<Problema> listaProblemas = new ArrayList<Problema>();

        listaProblemas.add(new Problema("computador nao liga"));
        listaProblemas.add(new Problema("rede fora do ar"));

        Chamada c = new Chamada();

        c.abriChamada(new Date(), listaProblemas);

        listaChamadas.add(c);

        System.out.println("chamada fechada: "+c.isFechada());

        c.mostraProblemas();

        System.out.println("\n");

        Atendente est = new Atendente("Estagiário");

        est.resolveChamada(c);

        System.out.println("chamada fechada: "+c.isFechada());

        c.mostraProblemas();

    }
}

```

Figura 55: Classe Solicitante para um *Domain Model*.

Fonte: O próprio autor.

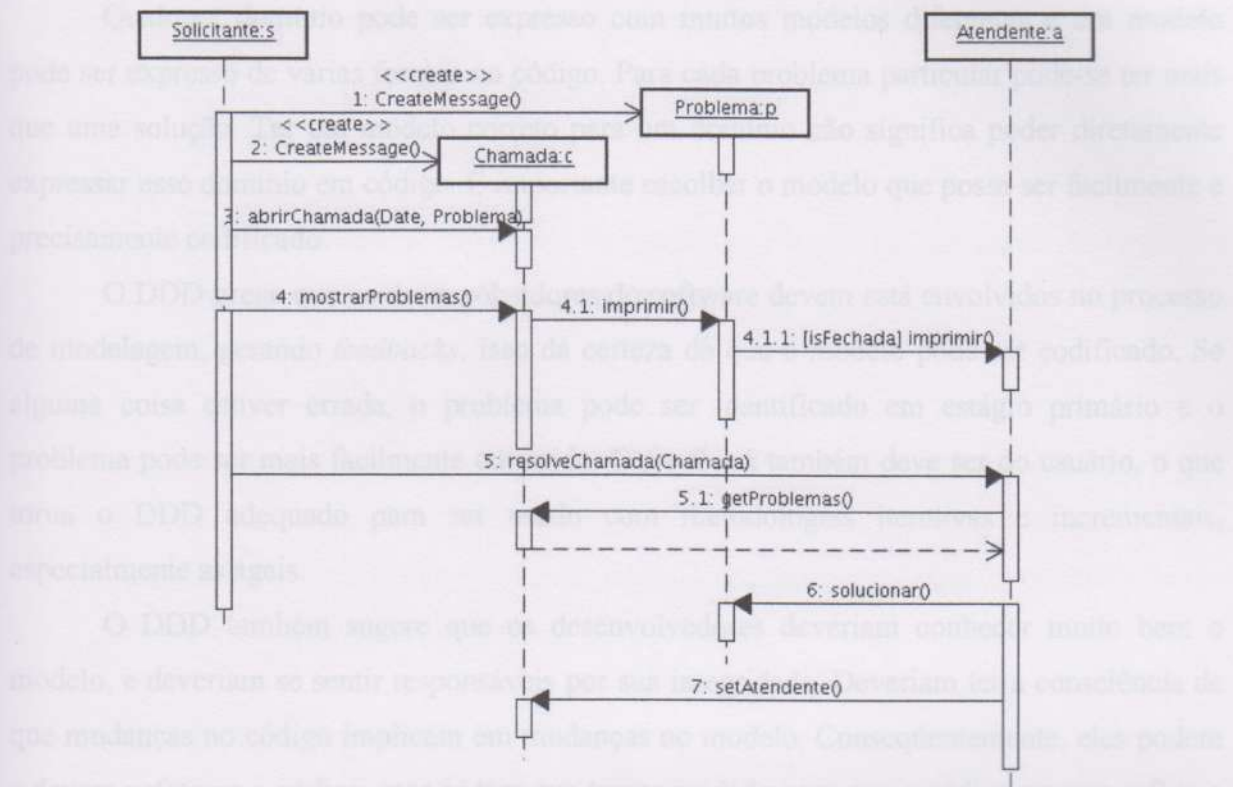


Figura 56: Diagrama de Sequência de um Domain Model

Fonte: O próprio autor.

4.3 DOMAIN DRIVEN DESIGN - DDD

Nesta seção serão apresentados resumidamente alguns conceitos envolvendo uma filosofia de desenvolvimento chamada *Domain Driven Design* (DDD). DDD é uma forma disciplinada de criar um *Domain Model*. O foco dessa técnica é criar um modelo do domínio o mais próximo do modelo do mundo real, para que o desenvolvedor possa compreender mais facilmente o domínio da aplicação, para isso, ele deve desenvolver uma “linguagem universal” com os usuários e outros desenvolvedores.

Domain Driven Design combina práticas de desenvolvimento e design, e mostra como design e desenvolvimento podem trabalhar juntos para criar uma solução melhor. Um bom design vai acelerar o desenvolvimento, enquanto um *feedback* vindo do processo de desenvolvimento irá melhorar o design.

É preciso um processo longo de análise do domínio de negócio do software, descobrir os elementos fundamentais do domínio, enfatizar os relacionamentos entre eles, e criar um modelo correto, que capture com precisão o domínio.

Qualquer domínio pode ser expresso com muitos modelos diferentes e um modelo pode ser expresso de varias formas no código. Para cada problema particular pode-se ter mais que uma solução. Ter um modelo correto para um domínio não significa poder diretamente expressar esse domínio em código. É importante escolher o modelo que possa ser facilmente e precisamente codificado.

O DDD prega que os desenvolvedores do software devem está envolvidos no processo de modelagem, gerando *feedbacks*. Isso dá certeza de que o modelo pode ser codificado. Se alguma coisa estiver errada, o problema pode ser identificado em estágio primário e o problema pode ser mais facilmente corrigido. O *feedback* também deve ser do usuário, o que torna o DDD adequado para ser usado com metodologias iterativas e incrementais, especialmente as ágeis.

O DDD também sugere que os desenvolvedores deveriam conhecer muito bem o modelo, e deveriam se sentir responsáveis por sua integridade. Deveriam ter a consciência de que mudanças no código implicam em mudanças no modelo. Conseqüentemente, eles podem e devem *refatorar* o código, mas se tem que tomar cuidado para que o código sempre reflita o modelo originalmente projetado. Se os processos de análise e implementação forem separados logo o código vai perder essa relação com o modelo.

O DDD sugere dividir programas complexos em camadas. Essa divisão deve seguir algumas regras: os elementos internos de cada camada devem ter o máximo de coesão possível, uma camada deve ter baixo acoplamento com as camadas acima dela (podendo manter dependência apenas com as camadas mais abaixo), concentrar todo código relacionado com o *Domain Model* em uma camada e isolá-la do código das demais camadas (o modelo é a exceção, não podendo depender nem das camadas abaixo dela). Os objetos de domínio livre de fazerem as tarefas das demais camadas, podem se focar em expressar precisamente o modelo. Isso permite o modelo ser rico o bastante e claro o bastante para capturar o conhecimento essencial do negócio e fazê-lo funcionar.

Abaixo mostra-se uma descrição das quatro camadas principais que o DDD recomenda que se use:

Interface com o Usuário (Camada de Apresentação)

- Responsável por apresentar informações para o usuário e interpretar os comandos vindos do usuário (usuário nesse contexto é qualquer entidade externa ao sistema que

interage com o sistema solicitando algum serviço, podendo ser uma pessoa ou outro sistema).

Camada de Aplicação

- Camada que coordena as atividades da aplicação. Ela não contém lógica de negócio e não guarda o estado dos objetos de negócio. Pode apenas guardar o estado das tarefas da aplicação.

Camada de Modelo

- Esta camada contém informações a respeito do domínio. É a principal camada de uma aplicação. Os estados dos objetos de negócio são guardados aqui. A persistência dos objetos é delegada para a camada de infra-estrutura.

Camada de Infra-Estrutura

- Esta camada serve como suporte para todas as outras camadas. Ela provê comunicação entre camadas. Implementa persistência para objetos de negócio, contém bibliotecas de suporte para a camada de interface com o usuário, entre outros.

A camada de modelo deveria ser focada no domínio. Ela não deveria estar envolvida em atividade de infra-estrutura. A camada de interface com o usuário não deveria nem estar fracamente conectada com a camada de lógica de negócio, nem com as tarefas que normalmente pertencem a camada de infra-estrutura. Uma camada de aplicação é necessária em muitos casos. Ela tem que ser gerenciada sobre a camada de lógica de negócio e supervisionar e coordenar toda a atividade da aplicação.

Existem alguns conceitos que estão envolvidos com o projeto da camada de domínio no *Domain Driven Design*. Nesta seção descrevem-se esses conceitos levando-se em conta apenas o que for relevante do ponto de vista do DDD, alguns dos conceitos citados nessa seção, como Fábricas, Capítulo 3, e Repositórios, Capítulo 5, são padrões de projeto e são mostrados com mais detalhes em outros capítulos desse trabalho.

4.3.1 *Entity*

Entidades são objetos que possuem um identificador. Esse identificador deve ser único para todo o modelo. Sua principal função é distinguir um objeto de todos os outros do modelo que possuem identidade.

Entidades são objetos importantes no *Domain Model* e eles devem ser considerados desde o começo da modelagem do sistema. Também é importante determinar se um objeto precisar ser uma entidade ou não, pois isso implica em certos custos para o sistema discutidos mais adiante nessa mesma seção.

4.3.2 *Value Object*

Existem implicações no desempenho do sistema fazer todos os objetos do domínio entidades. O processo para se criar um identificador em um objeto não é tão simples. Em muitos casos estamos interessados apenas nos valores dos atributos de um objeto, independentemente de qual objeto seja. Um objeto que é usado para descrever certos aspectos do domínio, mas que não possui um identificador é chamado de *Value Object*.

É recomendável que *Values Objects* sejam imutáveis, ou seja, que eles sejam criados com os valores passados no construtor e esses valores nunca mais sejam modificados durante sua existência. Quando se precisar de um valor diferente para os atributos, simplesmente se cria outro *Value Object*. Isso simplifica bastante o design da aplicação.

Como *Values Objects* são objetos imutáveis e não possuem identificador eles podem ser compartilhados entre objetos, até mesmos objetos de outras camadas, pois não se corre o risco que outros objetos alterem o seu valor e gerem uma inconsistência.

Values Objects podem conter outros *Values Objects*, bem como referência para Entidades.

4.3.3 *Services*

Um dos princípios da programação orientada a objetos é o encapsulamento, no qual objetos devem ser responsáveis por manipular seus próprios dados e manter a consistência deles. Mas, às vezes, existem algumas ações no domínio que não pertencem a nenhum objeto do domínio em particular, essas ações são importantes e não podem ser ignoradas.

É necessário criar alguma coisa para conter essas operações. Usando uma linguagem orientada a objetos, como não se pode deixar ações soltas, é necessário criar objetos que

possuam essas operações. Esses objetos que possuem operações que não dizem respeito a um único objeto no domínio são chamados em DDD de *Services*.

Um exemplo do que foi descrito no parágrafo anterior, citado em (AVRAM, A., MARINESCU, F. 2007), é transferência de um valor de uma conta de banco para outra conta. Onde essa operação deveria estar? Na conta de origem? Na conta de destino? Esse conceito não pertence a uma conta em particular, logo deveria estar em um *Service*.

Se a incorporação de uma funcionalidade em um *Entity* ou *Value Object* criar uma confusão se essa funcionalidade pertence mesmo a o objeto em questão, é uma boa indicação que provavelmente essa funcionalidade deveria estar incorporada a um *Service*.

Há três características básicas de um *Service*.

- A operação contida em um *Service* faz referência a um conceito do domínio que naturalmente não pertence a uma *Entity* ou a um *Value Object*.
- A operação necessariamente refere-se a outro objeto no domínio.
- A operação não guarda o estado dela.

4.3.4 Módulos

Para aplicações complexas, o modelo tende a ficar muito grande, tornando-se difícil de ser compreendido como um todo. Por esse motivo, é preciso reorganizar o modelo em módulos.

Contexto diz respeito ao significado de uma entidade do modelo. Por exemplo, uma classe *Pedido* pode ter mais de um significado dependendo do Contexto (pedidos de entrada ou pedidos de baixa no estoque). Um Contexto pode ser dividido em Módulos, que agrupam entidades com conceitos em comum. A comunicação entre Contextos e entre Módulos é dada através de diversos padrões e técnicas para manter uma alta coesão e um baixo acoplamento do modelo.

Existem várias formas de coesão entre elementos de um módulo. As duas formas principais são a coesão na comunicação e a coesão funcional. A coesão na comunicação é atingida quando partes do módulo operam sobre os mesmos dados, por exemplo, um módulo que realize operações completamente distintas para o sistema, mas sempre em cima dos dados o objeto *Usuario*. Já a coesão funcional é alcançada quando todas as partes de um módulo trabalham juntas para realizar uma tarefa bem definida, como exemplo cita-se um sistema com um módulo responsável por gerar relatórios, todas as classes desse módulo realizam exclusivamente alguma tarefa ligada a geração dos relatórios. Essa última é considerada o

melhor tipo de coesão, pois separa melhor as funcionalidades de cada módulo. Módulos, como camadas, deveriam ter interfaces bem definidas que seriam acessadas por outros módulos. Em vez de chamar três objetos de um módulo, é melhor acessar uma interface, porque isso reduz o acoplamento.

4.3.5 Agregações

Objetos de domínio atravessam um conjunto de estados durante suas vidas. Eles são criados, carregados na memória, usados e destruídos. Gerenciar o ciclo de vida de um objeto do domínio constitui um desafio para ele próprio e, se isso não é feito apropriadamente, pode ter um impacto negativo no *Domain Model*.

Um modelo pode conter um grande número de objetos de domínio. Não importa como foi feito o design, acontece que muitos objetos possuem associações, criando uma complexa rede de relacionamentos. Existem associações muitos para muitos e um grande número dessas associações são bidirecionais. Isso aumenta em muito a complexidade, fazendo o gerenciamento do ciclo de vida de cada objeto envolvido na associação difícil. O número de associações deveria ser reduzido o máximo possível.

Depois de reduzir e simplificar associações entre objetos ainda há muitos relacionamentos. Se muitos objetos contêm tais referências, é difícil assegurar que elas estejam todas removidas. Isso geralmente é deixado para ser verificado no nível do Banco de Dados com suas integridades referências entre tabelas. Transações são usadas para reforçar a integridade dos dados.

Também é necessário está hábil para resolver o problema das invariantes. Invariantes são as condições que um objeto tem que manter durante todo o seu ciclo de vida. É dever da implementação do objeto zelar que esteja sempre em estado válido. Por exemplo, um objeto avião precisa ter duas asas enquanto existir, essa condição nunca pode ser quebrada (não varia durante o ciclo de vida do objeto). Phillip Calçado fala muito bem desse problema das invariantes em seu artigo *Contratos Nulos*. Isso é difícil para realizar quando muitos objetos possuem referência para objetos cujos dados estão sendo mudados. É difícil garantir a consistência dos dados em um modelo com associações complexas.

Muitos desses problemas podem ser resolvidos ou minimizados com o uso de Agregações. Uma Agregação é um grupo de objetos associados que são considerados como único objeto no que diz respeito às mudanças dos dados. A Agregação é limitada por uma fronteira que separa os objetos de dentro dos objetos de fora dela. Toda Agregação possui um

objeto raiz, esse objeto tem que ser uma Entidade e ele é o único objeto acessível de fora da Agregação. Este ainda pode conter referências para qualquer um dos objetos da Agregação e os outros objetos dentro desta podem referenciar uns aos outros, mas um objeto fora da Agregação só pode referenciar o objeto raiz. Se existem outras Entidades dentro das fronteiras da Agregação, a identidade dessas entidades é local, fazem sentido apenas dentro da agregação.

Como outros objetos podem referenciar somente o objeto raiz, isso significa que eles não podem diretamente mudar os outros objetos da Agregação. Tudo que eles podem fazer é mudar o objeto raiz, ou pedir ao mesmo para realizar alguma ação. Ele poderá mudar os outros objetos, mas isso é uma operação contida dentro da Agregação e esta pode ser melhor controlada.

Se o objeto raiz é apagado ou removido da memória, todos os outros objetos da Agregação serão apagados também, porque não há outros objetos mantendo referências para algum objeto da Agregação. Quando uma mudança é feita no objeto raiz que indiretamente afeta ou outros objetos da Agregação, é mais simples determinar as invariantes porque o objeto raiz será o responsável por fazer isso. É mais difícil fazer quando objetos externos têm acesso direto para objetos internos e os mudam. Forçar as invariantes nessa circunstância envolve colocar alguma lógica em um objeto externo para tratá-las, que não é desejável, pois o próprio objeto deveria ser o responsável por manter a consistências dos seus dados.

É possível para o objeto raiz passar referências transientes de objetos internos para um objeto externo, com a condição que o objeto externo não mantenha a referência depois que a operação está finalizada. Uma maneira simples de fazer isso é passar cópias de *Value Objects* para os objetos externos. Não importa o que acontecer com esses objetos, porque a integridade da Agregação não será afetada de nenhuma forma.

Se objetos de uma Agregação são persistidos em um Banco de Dados, somente o objeto raiz da Agregação deveria ser obtível através de consultas. Os outros objetos deveriam ser obtidos através de associações com o objeto raiz. Para um objeto dentro da uma Agregação é permitido guardar referência para objetos raízes de outras Agregações.

4.3.6 *Factory*

Quando um objeto quer criar outro objeto, ele chama seu construtor e, possivelmente, passa algum parâmetro para ele. Mas quando a construção de um objeto é um processo trabalhoso, criá-lo envolve muito conhecimento sobre a estrutura interna do mesmo. Isso

significa que cada cliente do objeto irá obter conhecimento específico sobre a construção dele. Isso quebra encapsulamento dos objetos de domínio e da Agregação. Se o cliente pertence à camada de aplicação, parte da camada de domínio está integrada com essa camada, desorganizando todo o projeto da sua aplicação. É como se fosse preciso fornecer a matéria prima para um fabricante para poder comprar um produto. Isso não é impossível, mas não é assim que as coisas acontecem, não é preciso ser um engenheiro em telecomunicações para comprar um celular.

É necessário ser introduzir um novo conceito que ajude a encapsular o processo de criação de objetos complexos, esse conceito é chamado Fábrica. Elas são usadas para encapsular o conhecimento necessário para criação de objetos e são especialmente úteis para criar Agregações. Quando o objeto raiz da Agregação é criado, todos os objetos contidos na Agregação são criados juntos com o este e todas as invariantes (subseção 4.3.5, p.55) são verificadas.

Uma Agregação contém uma série de objetos que possuem uma ligação próxima. A construção do objeto raiz é ligada com a criação dos outros objetos da Agregação. A lógica para a criação de uma Agregação naturalmente não pertence a qualquer objeto, porque isso é sobre a criação de outros objetos.

Parece apropriado usar uma classe Fábrica especial que é dada a tarefa de criar as Agregações, e que irá conter as regras, as restrições e as invariantes que têm que ser consideradas para a Agregação ser válida. Os objetos irão permanecer simples e servir aos seus propósitos sem o custo dessa lógica de construção complexa.

É importante o processo de criação ser atômico, ou o objeto é criado com seus dados em um estado válido ou não é criado. Se aquele não for atômico, há chances dele ser feito pela metade para alguns objetos, levando-os para um estado inconsistente. Isso é mais crucial para Agregações. Quando o objeto raiz é criado, é necessário que todos os objetos subordinados às invariantes sejam criados também. Para *Value Objects*, que são imutáveis, isso significa que todos os atributos são inicializando em seus estados válidos. Se um objeto não pode ser criado propriamente, uma exceção deve ser lançada, para ser ter certeza que um valor inválido não é retornado.

4.3.7 Repositórios

O propósito do uso de um Repositório é encapsular toda a lógica necessária para obter uma referência de objeto. Os objetos de domínio não têm que se preocupar com a infra-estrutura necessária para obter outro objeto do domínio. Eles iram apenas pegá-los do Repositório, como consequência o modelo manterá sua claridade e foco, ficando desacoplado da infra-estrutura necessária para persistir os objetos.

Um Repositório age como um lugar de armazenamento para objetos acessíveis globalmente, através de uma interface global bem conhecida. Um Repositório pode também incluir um padrão *Strategy*. Ele pode acessar um armazenamento persistente baseado em uma estratégia específica, permitindo a ele usar um local diferente de armazenamento para diferentes tipos de objetos.

É aconselhado que para cada tipo de objeto que precise ser acessado globalmente, se crie um Repositório que providencie a ilusão de acessar objetos em coleções, como se tivessem em memória. Outra boa prática é criar Repositórios apenas para os objetos raiz das Agregações. Um Repositório deve conter métodos com todas as operações que devem ser realizadas com objetos persistidos, por exemplo, incluir, apagar, buscar pelo identificador, entre outras. Um Repositório pode ter detalhes sobre o acesso à infra-estrutura, mas sua interface deve ser simples. Há um relacionamento entre Fábricas e Repositórios. Eles são ambos padrões de design do *Domain Model*, e ambos ajudam o gerenciamento do ciclo de vida dos objetos do domínio. Enquanto aquelas estão concentradas com a criação de objetos, estes cuidam de objetos já existentes. Um Repositório pode manter objetos em *cache* localmente, mas o mais freqüente é precisar obter esses objetos de algum meio persistente.

Neste capítulo foi descrito o padrão *Domain Model*, discutiu-se as vantagens de um modelo rico em relação a um modelo anêmico e foi descrito um pouco da técnica de programação *Domain Driven Design*. No próximo capítulo serão descritos os padrões da camada de persistência.

5 PADRÕES DA CAMADA DE PERSISTÊNCIA

Este capítulo é focado nos padrões que pertencem a uma camada que pode ser considerada uma subcamada da Camada de Infra-Estrutura, os padrões da Camada de Persistência. Essa camada faz parte da Camada de Infra-Estrutura porque fornece a Infra-Estrutura necessária para a persistência dos objetos.

A Camada de Infra-Estrutura contém os códigos de acesso aos serviços que estão sendo oferecidos ao sistema. No caso da Camada de Persistência, esse serviço é o armazenamento das informações do modelo, geralmente em um Banco de Dados relacional.

5.1 DATA MAPPER

5.1.1 Conceito

O padrão *Data Mapper* consiste em uma camada de “objetos mapeadores”, que movem dados entre os objetos do modelo e o Banco de Dados, mantendo um independente do outro; além de manter tanto os objetos do modelo, quando o Banco independentes do próprio objeto que está fazendo o mapeamento, como se pode ver na Figura 57.

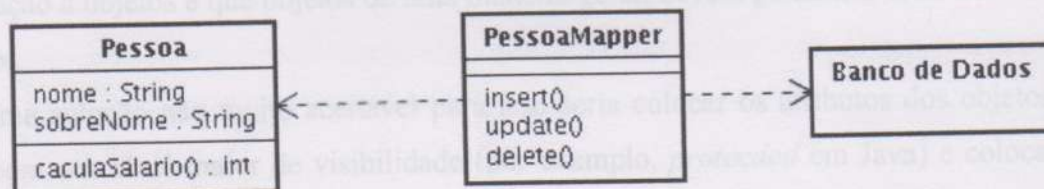


Figura 57: Dependências do padrão *Data Mapper*
Fonte: O próprio autor.

Objetos e Banco de Dados relacionais têm diferentes princípios de agrupamento de dados. Se for preciso transferir dados entre essas duas estruturas, essa transferência se torna complexa e, se os objetos em memória conhecem a estrutura do Banco de Dados relacional, realizar alguma mudança em uma das estruturas afetará a outra.

Um *Data Mapper* simples apenas mapeia uma tabela do banco de dados para uma classe equivalente em memória, campo por campo. Mas em exemplos práticos dificilmente isso ocorre. O modelo de objetos, na maioria das vezes, possui dados multivalorados, herança e outros aspectos não suportados pelo esquema relacional de um Banco de Dados. E um *Data Mapper* deve suportá-los. A camada de *Data Mapper* pode ser substituída para propósitos de testes ou para permitir um modelo único trabalhar com diferentes Bancos de Dados.

Usa-se *Data Mapper* principalmente quando se deseja tornar o esquema de um Banco de Dados relacional e o modelo de uma aplicação independentes entre si. O caso mais comum é quando o modelo de uma aplicação está usando o padrão *Domain Model* (Capítulo 4), já que esse precisa ser auto contido e independente das outras camadas da sua aplicação. O uso do padrão *Data Mapper* torna o código mais limpo e fácil de entender uma vez o modelo fica livre de código para persistência dos objetos.

Contudo, o trabalho gerado por se precisar construir uma camada a mais no sistema só se justifica se a lógica da aplicação for complexa, se a lógica for simples, não haverá a necessidade de se usar uma camada extra de mapeamento dos dados; o acesso ao Banco de Dados pode ser feito, por exemplo, diretamente através do padrão *Active Record*, visto mais adiante nesse mesmo capítulo na seção 5.3. À medida que a aplicação for ficando complexa, ela pode ser “refatorada” para substituir o *Active Record* pelo *Data Mapper*.

Um problema de se utilizar o padrão *Data Mapper* é que os “objetos mapeadores” precisam acessar os campos dos objetos do modelo. Conseqüentemente será preciso criar métodos públicos para todos os campos dos objetos do modelo, mesmo que sejam campos imutáveis que não precisassem ser modificados durante o ciclo de vida do objeto. Isso pode gerar um problema de inconsistência dos dados, e como dito anteriormente, um dos princípios da orientação a objetos é que objetos de uma maneira geral, devem garantir a consistência dos seus dados.

Uma solução não muito aceitável para isso seria colocar os atributos dos objetos do modelo com um nível maior de visibilidade (por exemplo, *protected* em Java) e colocar as classes que realizam o mapeamento no mesmo pacote do modelo, mas com isso, outras partes do sistema que conhecem o modelo vão conhecer seus mapeamentos também. Pode-se usar *reflection*, que passa por cima das regras de visibilidade, é mais lento, mas pode ser um atraso desprezível se comparado com o tempo de acesso ao Banco de Dados. Ou pode-se usar métodos públicos, implementando alguma lógica que dispare uma exceção, quando o método for usado fora do contexto do preenchimento dos objetos com dados provenientes do Banco de Dados.

A camada de persistência precisa saber quando objetos foram criados, destruídos ou sofreram mudanças. Para isso, o padrão *Data Mapper* trabalha com o padrão *Unit of Work* (FOWLER et al, 2002). Como objetos do modelo geralmente estão muito interconectados uns com os outros, é necessário implementar um padrão para determinar até que nível nesses relacionamentos se deseja alcançar, senão é possível que sejam trazidos mais dados do que era

desejado do Banco de Dados. Para isso, o *Data Mapper* trabalha com o padrão *Lazy Load* (FOWLER et al, 2002). Para otimizar as consultas, seria interessante que o *Data Mapper* não fosse buscar no Banco de Dados objetos que já tivessem em memória, trazidos em requisições anteriores, para resolver esse problema o *Data Mapper* deve trabalhar com mais um padrão conhecido como *Identity Map* (FOWLER et al, 2002).

Para a plataforma Java a Sun¹⁵ catalogou uma sub-implementação desse padrão com o nome de *Data Access Object* (DAO) (SUN, 2007). Pois DAOs apenas encapsulam o acesso aos dados, eles não realizam o mapeamento em si. Nos exemplos dessa seção serão utilizados para implementar o padrão *Data Mapper* o padrão DAO junto como o *Framework* de mapeamento objeto relacional Hibernate¹⁶. O Hibernate além de realizar o mapeamento dos campos de um objeto para os campos de uma tabela, já implementa todos os padrões anteriormente citados: *Unit of Work*, *Lazy Load* e *Identity Map*, necessários para um funcionamento eficaz do *Data Mapper*. O Hibernate também suporta mapeamentos mais complexos como herança e atributos multivalorados.

Para tornar as duas camadas, modelo e mapeamento, mais independentes, pode-se usar o padrão *Separated Interface* (Capítulo 3) para separar a interface das classes “mapeadoras” das suas implementações e utilizar o padrão *Dependency Injection* para instanciar e injetar essas implementações. A maioria dos métodos de busca são comuns para todos os objetos do modelo, nesse caso, é uma prática corriqueira utilizar o padrão *Layer Supertype* (Capítulo 3) na camada de mapeamento. O exemplo desta da arquitetura é mostrado na Figura 58.

É necessário baixar em seu site os arquivos JARs mostrados abaixo e adicioná-los ao classpath de aplicação, junto com um arquivo JAR que contenha o *Driver* do Banco de Dados que se pretende usar.

- | | |
|-------------------------------|-----------------------------|
| • asm.jar | • chetane-1.1.jar |
| • asm-all.jar | • ejb3-persistence.jar |
| • cglib-2.1.jar | • hibernate.jar |
| • commons-collections-3.1.jar | • hibernate-annotations.jar |
| • commons-logging-1.0.4.jar | • jta.jar |
| • dom4j-1.6.jar | • log4j-1.2.9.jar |

¹⁵ <http://www.sun.com/>

¹⁶ <http://www.hibernate.org/>

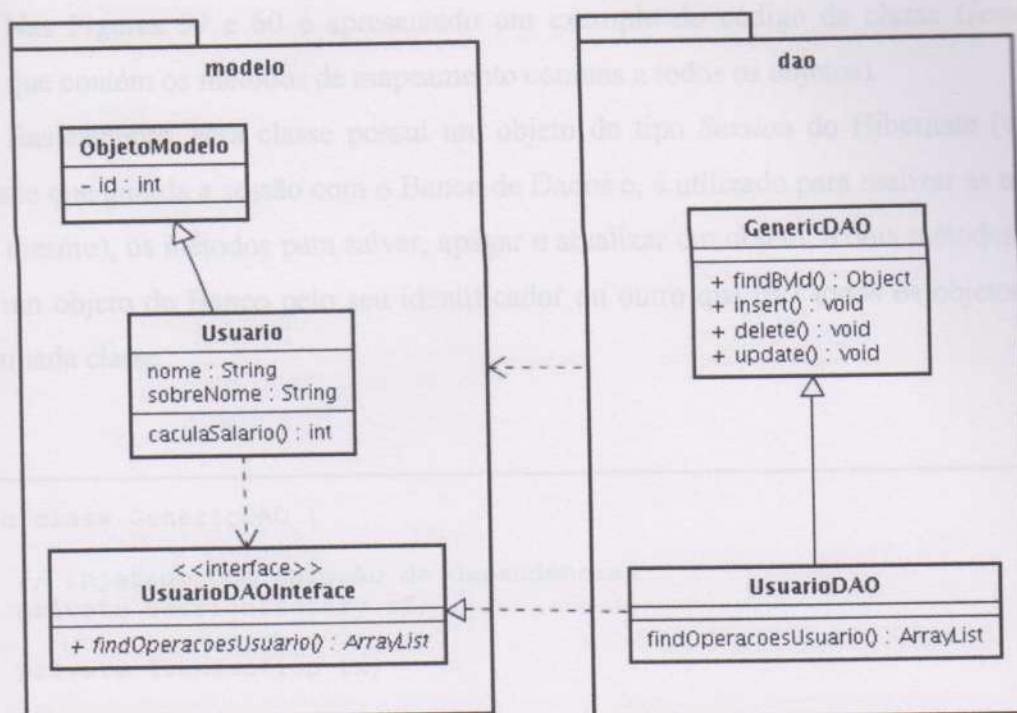


Figura 58: Arquitetura do padrão *Data Mapper*
Fonte: O próprio autor.

5.1.2 Exemplo de Uso do Padrão *Data Mapper*

Como foi dito na subseção anterior, nesta subseção utilizar-se-á o padrão DAO mais o *Framework* Hibernate para implementar o padrão *Data Mapper*. Para utilizar o Hibernate é necessário baixar em seu site os arquivos JARs mostrados abaixo e adicioná-los ao *classpath* da aplicação, junto com um arquivo JAR que contenha o *Driver* do Banco de Dados que se pretenda usar.

- asm.jar
- asm-attrs.jar
- cglib-2.1.jar
- commons-collections-2.1.1.jar
- commons-logging-1.0.4.jar
- dom4j-1.6.jar
- ehchace-1.1.jar
- ejb3-persistence.jar
- hibernate3.jar
- hibernate-annotations.jar
- jta.jar
- log4j-1.2.9.jar

Nas Figuras 59 e 60 é apresentado um exemplo do código da classe *GenericDAO* (classe que contém os métodos de mapeamento comuns a todos os objetos).

Basicamente, essa classe possui um objeto do tipo *Session* do Hibernate (objeto do Hibernate que guarda a sessão com o Banco de Dados e, é utilizado para realizar as operações com o mesmo), os métodos para salvar, apagar e atualizar um objeto, e dois métodos, um que busca um objeto do Banco pelo seu identificador ou outro que traz todos os objetos de uma determinada classe.

```
public class GenericDAO {  
  
    // injetado com injeção de dependências  
    private SessionFactory sf;  
  
    private Transaction tx;  
  
    private Session session;  
  
    public GenericDAO() { session = getSession();}  
  
    protected Session getSession() {  
        if (session == null)  
            session = sf.openSession();  
        return session;  
    }  
  
    public void startTrasacao() {tx = getSession().beginTransaction();}  
    public void commitTrasacao() { tx.commit();}  
    public void rollbackTrasacao() { tx.rollback();}  
  
    public void close() {if ( session != null) session.close();}  
  
    public void save(Object o) throws Exception {  
        try {  
            startTrasacao();  
  
            session.save(o);  
  
            commitTrasacao();  
  
        } catch (Exception e) {  
            e.printStackTrace();  
            rollbackTrasacao();  
            throw e;  
        }  
    }  
}
```

Figura 59: Código Fonte da classe *GenericDAO*
Fonte: O próprio autor.

```

public void delete(Object o) throws Exception {
    try {
        startTrasacao();

        session.delete(o);

        commitTrasacao();

    } catch (Exception e) {
        e.printStackTrace();
        rollbackTrasacao();
        throw e;
    }
}

public void update(Object o) throws Exception {
    try {
        startTrasacao();

        session.update(o);

        commitTrasacao();

    } catch (Exception e) {
        e.printStackTrace();
        rollbackTrasacao();
        throw e;
    }
}

public Object findByPrimaryKey(Class c, Serializable id) {
    return session.get(c, id);
}

public Collection findAll(Class c) {
    return session.createCriteria(c).list();
}

```

Figura 60: Continuação do Código Fonte da classe *GenericDAO*
Fonte: O próprio autor.

A classe *UsuarioDAO*, Figura 61, implementa um DAO que contém os métodos específicos para objetos da classe *Usuario*, nesse caso o método que encontra todas as operações que um usuário realizou no sistema.

```

public class UsuarioDAO extends GenericDAO{

    public Collection findOperacoesUsuario(Usuario u){
        Query q = this.getSession()
            .createQuery("from OperacoesSistema "
                + "where OperacoesSistema.loginUsuario = "
                + u.getLogin());
        return q.list();
    }
}

```

Figura 61: Código Fonte da classe UsuarioDAO
Fonte: O próprio autor.

Na Figura 62 é mostrado novamente o código da classe *Usuario*, esse código já foi mostrado na Figura 19, mas agora foram omitidos alguns métodos e são mostradas suas anotações, para o mapeamento da classe.

```

@Entity
public class Usuario{

    public static final float FATOR_HORA_TRABALHADA = 42.5f;

    @Id private int id;

    private String login;
    private String senha;

    public Usuario(String login, String senha){
        this.login = login; this.senha = senha;
    }

    public String getLogin() {return login;}

    public String getSenha() {return senha;}

    public BigDecimal calculaSalario(int qtdHorastrabalhadas){
        return new BigDecimal(qtdHorastrabalhadas *
            FATOR_HORA_TRABALHADA);
    }
}

```

Figura 62: Código Fonte da classe Usuario
Fonte: O próprio autor.

Finalizando, para que o Hibernate saiba qual Banco de Dados deve usar, é necessário configurá-lo através de um simples arquivo XML chamado *hibernate.cfg.xml*. O conteúdo desse arquivo é mostrado na Figura 63.

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>

  <!-- a SessionFactory instance listed as /jndi/name -->
  <session-factory name="java:comp/env/hibernate/SessionFactory">
    <!-- properties -->
    <property name="connection.driver_class">
      org.postgresql.Driver
    </property>
    <property name="connection.url">
      jdbc:postgresql://localhost:5432/hibernate
    </property>
    <property name="dialect">
      org.hibernate.dialect.PostgreSQLDialect
    </property>
    <property name="show_sql">true</property>
    <property name="connection.username">postgres</property>
    <property name="connection.password">postgres</property>
    <property name="connection.pool_size">10</property>

    <!-- mapping classes -->
    <mapping class="modelo.Usuario" />
  </session-factory>
</hibernate-configuration>

```

Figura 63: Arquivo *hibernate.cfg.xml*
Fonte: O próprio autor.

A partir desse momento, para persistir ou consultar um objeto do Banco de Dados é necessário chamar algum método dos DAOs. Como por exemplo, o código para salvar um objeto *Usuario* no Banco de Dados é mostrado na Figura 64.

```

GenericDAO dao = new UsuarioDAO();
dao.save(u);

```

Figura 64: Código para salvar um objeto da classe *Usuario* no Banco de Dados
Fonte: O próprio autor.

Nota-se que as classes do modelo não contêm nenhum código relacionado com a persistência dos dados. Todo o código está concentrado nos DAOs, o que torna, como dito anteriormente, o modelo independente do Banco de Dados e o Banco independente do modelo.

5.2 REPOSITORY

5.2.1 Conceito

Repositório é uma subcamada da camada do modelo que abstrai o acesso à camada de persistência, isto é, abstrai mais ainda o modelo do Banco de Dados relacional ou outra tecnologia usada para persistência em relação ao padrão *Data Mapper*, Seção 5.1. Para os objetos do modelo fica a impressão de que os mesmos estão sendo salvos em coleções na memória, quando na verdade podem estar sendo salvos em um Banco de Dados relacional, arquivo XML ou mesmo ficando em coleções na memória (caso se queira realizar testes sem a necessidade de persistir os objetos).

O principal motivo da existência de um Repositório é que o padrão *Domain Model*, visto no Capítulo 4, deve ser auto contido, isso é, não devem depender de outros módulos, pelos menos não diretamente. Quando algum objeto do modelo precisar acessar serviços de outras camadas, deve usar o princípio da *Dependency Inversion Principe* (DIP) (MARTIN, R. C., 2008) que tem dois princípios fundamentais, segundo Phillip Calçado em seu artigo: *Desenvolvendo Sistemas OO com Padrões de Negócio*. Os Repositórios são as abstrações citadas abaixo.

- “Módulo de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações;”
- “Abstrações não devem depender dos detalhes. Detalhes devem depender das abstrações.”

Se a camada do modelo acessasse diretamente a camada de persistência, os objetos do modelo estariam dependendo diretamente da infra-estrutura da aplicação, além do mais, o modelo saberá que os objetos estão sendo armazenados em outro lugar, essa informação não é importante para o ele e, para deixar o modelo enxuto e independente, ele não deveria saber disso. Com o uso do Repositório, os objetos do modelo estariam lidando diretamente com o Repositório, lugar onde se adicionam e retiram objetos e que pertence ao próprio modelo.

A camada de persistência normalmente implementa o Repositório para salvar os dados em um Banco de Dados relacional, mas os objetos do modelo não tomam conhecimento disso.

Os objetos do modelo e os objetos da camada de persistência vão depender do Repositório (“*Detalhes devem depender das abstrações*”), o Repositório não vai depender de nenhuma das duas (“*Abstrações não devem depender dos detalhes*”). As camadas de modelo e persistência não ficam dependentes uma da outra, seguiu-se o princípio do DIP.

Repositório também torna o desenvolvimento mais intuitivo, uma vez que se programa o modelo como se não existisse persistência. Todos os objetos do modelo são obtidos do

repositório através de uma interface cujos métodos possuem a mesma sintaxe dos métodos de coleções (*add()*, *remove()*, *contains()*).

Esse padrão se torna mais importante e viável quando há um grande numero de classes do modelo e consultas pesadas, ajudando a minimizar as duplicatas das lógicas das consultas. Não é necessário ter um Repositório por classe de negócio, apesar de ser uma boa prática.

5.2.2 Exemplo de Uso do Padrão *Repository*

Na prática, Repositórios podem ser implementados como uma simples interface em Java que fornece a abstração desejada ao modelo, como mostrado na Figura 65.

```
public interface UsuarioRepositorio {  
    public void addUsuario(Usuario u);  
    public void removeUsuario(Usuario u);  
    public Collection findOperacoesUsuario(Usuario u);  
    public Collection<Usuario> getAll();  
    public boolean contains(Usuario u);  
}
```

Figura 65: Código Fonte do Repositório da classe *Usuario*
Fonte: O próprio autor.

Se se desejar salvar os objetos do modelo no Banco de Dados, só é preciso fazer os DAOs da seção anterior implementarem o Repositório, como mostrado da Figura 66. Caso se queira disponibilizar outro lugar para guardar aos objetos só é necessário fazer as novas classes, que realizarão essa ação, implementarem o Repositório. Como se pode perceber, e é a finalidade desse padrão, o modelo não vai tomar nenhum conhecimento que houve essa mudança.

```

public class UsuarioDAO extends GenericDAO implements UsuarioRepositorio{

    public Collection findOperacoesUsuario(Usuario u){
        Query q = this.getSession()
            .createQuery("from OperacoesSistema "
                + "where OperacoesSistema.loginUsuario = "
                + u.getLogin());
        return q.list();
    }

    public void addUsuario(Usuario u) {
        try {
            save(u);
        } catch (Exception e) {
            // código tratamento exceção
        }
    }

    public void removeUsuario(Usuario u) {
        try {
            delete(u);
        } catch (Exception e) {
            // código tratamento exceção
        }
    }

    public Collection<Usuario> getAll(){
        return findAll(Usuario.class);
    }

    public boolean contains(Usuario u){
        Usuario temp = (Usuario) findByPrimaryKey(Usuario.class,
            u.getId());

        if(temp != null)
            return true;

        return false;
    }
}

```

Figura 66: Código Fonte da classe *UsuarioDAO* usando Repositório
Fonte: O próprio autor.

5.3 ACTIVE RECORD

5.3.1 Conceito

Cada objeto vai ser salvo num modelo relacional como uma linha de uma tabela. A correspondência não é um para um, mas se pode abstrair e enxergar uma tabela como o equivalente a uma classe e cada linha da tabela um objeto.

Como consequência disso, o código fica mais leve e rápido para ser escrito para sistemas simples mas, em contrapartida, como existe uma grande dependência entre o modelo e o esquema relacional do Banco de Dados, qualquer mudança em um, implica mudança no outro. Isso torna mais difícil a manutenção do código.

A estrutura dos dados dos objetos do *Active Record* devem ser exatamente iguais com a estrutura dos dados no Banco de Dados, isso é, um campo na classe para cada coluna da respectiva tabela. Se o modelo possuir lógica complexa (muitos relacionamentos, coleções, herança) o código vai começar a ficar desorganizado e difícil de entender.

O código da classe *Usuario*, agora usando o padrão *Active Record*, pode ser visto na Figura 67.

```

public class Usuario{

    public static final float FATOR_HORA_TRABALHADA = 42.5f;
    private int id;
    private String login;
    private String senha;

    protected Connection con;

    public Usuario(String login, String senha){
        this.login = login;
        this.senha = senha;
    }

    public String getLogin() {return login;}
    public String getSenha() {return senha;}

    public BigDecimal calculaSalario(int qtdHorastrabalhadas){
        return new BigDecimal(qtdHorastrabalhadas *
                                FATOR_HORA_TRABALHADA);
    }

    public void save(){
        PreparedStatement st;
        try {
            st = con.prepareStatement("INSERT INTO USUARIO(ID, LOGIN,
SENHA) values (?, ?, ?)");
            st.setInt(1, id);
            st.setString(2, login);
            st.setString(3, senha);
            st.executeUpdate();
            st.close();
        } catch (SQLException e) {
            // código tratamento exceção
        }
    }

    public boolean delete(){
        int count = 0;
        PreparedStatement st;
        try {
            st = con
                .prepareStatement("DELETE FROM CLIENTE WHERE ID = ?");
            st.setInt(1, id);
            count = st.executeUpdate();
            st.close();
        } catch (SQLException e) {
            // código tratamento exceção
        }
        return count == 1;
    }
}

```

Figura 67: Código Fonte da classe *Usuario* usando o padrão *Active Record*
 Fonte: O próprio autor.

Neste capítulo foram mostrados os padrões da camada de persistência, na sequência é mostrado o capítulo com os padrões da camada de aplicação.

6 PADRÕES DA CAMADA DE APLICAÇÃO

Neste capítulo são apresentados os padrões com características úteis à camada de aplicação. Essa camada é a menos utilizada das quatro principais camadas apresentadas nesse texto. Geralmente, apenas aplicações mais complexas necessitam de uma camada de aplicação para coordenar o fluxo das atividades do sistema.

6.1 *FACADE*

6.1.1 Conceito

Padrão que simplifica a interface entre objetos, escondendo toda a complexidade da implementação de uma ou mais classes. Muitas ações são precisas para se ligar e andar com um carro. A bateria precisa fornecer corrente para o motor de partida, o motor de partida aciona a bomba de combustível, a bomba de combustível joga o combustível nos cilindros, as velas queimam o combustível, a queima gera uma explosão na câmara compressora, essa explosão desloca os pistões, os pistões estão ligados aos eixos, os eixos movimentam os pneus e finalmente o carro anda.

Se se precisasse acionar todos esses mecanismos manualmente, seria muito trabalhoso dirigir um automóvel. Felizmente o motorista só precisa girar uma chave e apertar o pedal do acelerador. A chave e o pedal funcionam como uma Fachada e escondem a complexidade do mecanismo de um motor de um automóvel.

É exatamente isso que o padrão *Facade* faz, ele funciona como a chave e o pedal no anterior do automóvel. Uma classe fachada fornece um método simples, que na sua implementação realiza chamadas a diversos métodos para realizarem uma determinada tarefa em um subsistema complexo de classes e relacionamentos, como é mostrado na Figura 68.

O padrão *Facade* não tira a acessibilidade do subsistema, se se desejar, ainda é possível realizar chamadas a um método específico dele diretamente. *Facade* também trás o benefício de desacoplar o cliente do subsistema que ele estiver acessando. Esse princípio é chamado por muitos autores (FREEMAN et al, [199?]) de Princípio do Menor Conhecimento, isso é, quanto menos objetos um determinado objeto conhecer, menor vai ser o acoplamento entre os componentes do sistema e menor o custo de mudanças e manutenção.

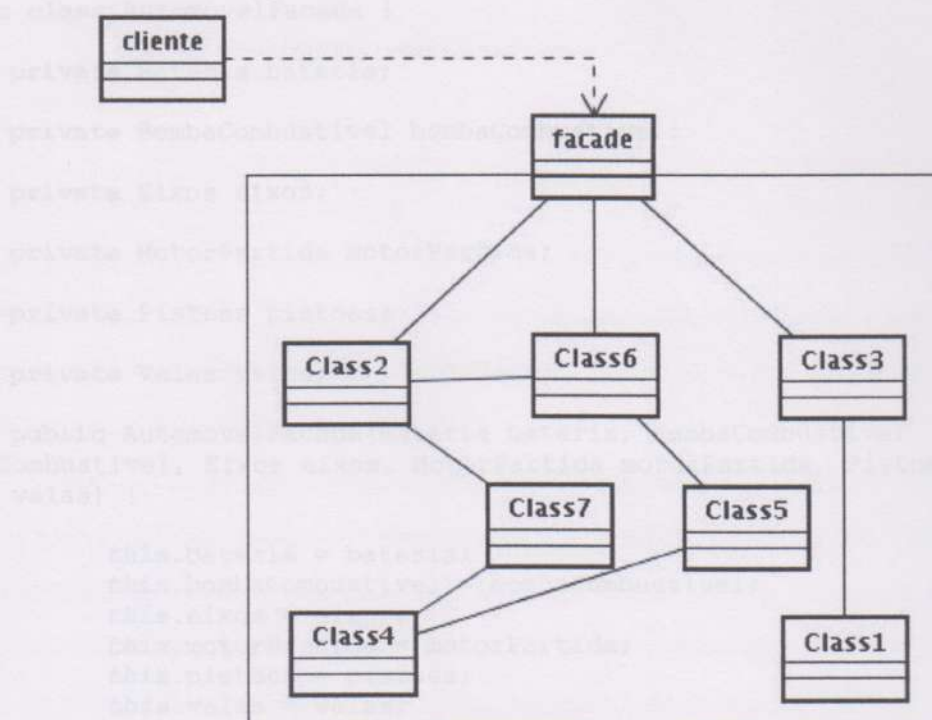


Figura 68: Esquema de dependências do padrão *Facade*.
Fonte: O próprio autor.

6.1.2 Exemplo de Uso do Padrão Facade

Nesta subseção mostra-se a implementação da classe fachada para o exemplo do automóvel da subseção anterior. Como é percebido na Figura 69, a classe *AutomovelFacade* possui o método *ligarAutomovel()* que realiza todas as ações necessárias, o cliente só precisa executar a chamada a este único método.

```

public class AutomovelFacade {

    private Bateria bateria;

    private BombaCombustivel bombaCombustivel;

    private Eixos eixos;

    private MotorPartida motorPartida;

    private Pistoes pistoes;

    private Velas velas;

    public AutomovelFacade(Bateria bateria, BombaCombustivel
bombaCombustivel, Eixos eixos, MotorPartida motorPartida, Pistoes pistoes,
Velas velas) {

        this.bateria = bateria;
        this.bombaCombustivel = bombaCombustivel;
        this.eixos = eixos;
        this.motorPartida = motorPartida;
        this.pistoes = pistoes;
        this.velas = velas;
    }

    public void ligarAutomovel() {
        bateria.forneceCorrente();
        motorPartida.acionarBomba();
        bombaCombustivel.injetaCombustivel();
        velas.disparaFaisca();
        pistoes.movimentaEixos();
        eixos.movimentaPneus();
    }
}

```

Figura 69: Código fonte da classe *AutomovelFacade*
Fonte: O próprio autor.

6.2 COMMAND

6.2.1 Conceito

A principal função do padrão *Command* é encapsular invocações a objetos. Quando invocar objetos torna-se uma tarefa complexa, como é o caso de invocar componentes remotos na especificação EJB 2.1, há a necessidade de se encapsular essas invocações para se simplificar o código. Com isso são abstraídas, para o cliente, coisas do tipo: onde, quando ou qual componente vai realizar uma determinada tarefa.

O princípio do padrão é guardar um “objeto comando” para cada objeto a ser invocado. Para invocar o objeto destino, invoca-se sempre o mesmo método do “objeto comando” (geralmente chamado de *execute*) e esse método padrão de um comando se encarrega de chamar os métodos adequados para realizar as ações requeridas. Com isso evita-se que o código do cliente fique poluído com várias estruturas de seleção (“Ifs”) testando o tipo do objeto a ser invocado para chamar o método apropriado.

O *Command* gera uma maior flexibilidade na arquitetura do sistema, uma vez que permite a adição de novas funcionalidades (comandos) sem a necessidade de mudar o design da arquitetura. Outro ponto positivo do padrão *Command* é que ele possibilita desfazer as ações realizadas por um comando. Para isso, é necessário implementar um método no comando (geralmente chamado de *undo*) que realize as operações contrárias as operações realizadas quando o comando foi invocado.

Um ponto negativo para esse padrão é que ele deixa o design da aplicação “procedural”, isto é, orientado a ações.

6.2.2 Exemplo de Uso do padrão *Command*

No exemplo dessa subseção, será mostrada a implementação do padrão *Command*, para se modelar a ação de se ligar um aparelho de Televisão.

Para se implementar o padrão *Command*, primeiro é necessário criar uma interface que contenha os métodos comuns para um comando. Esses métodos são o *execute*, para executar a ação do comando e se for necessário, o método *undo* para desfazer a ação realizada. O código dessa interface pode ser visto na Figura 70.

```

/** todo comando implementa a mesma interface
 * que consiste de apenas um metodo */
public interface Command {

    public void execute();

    public void undo();

}

```

Figura 70: Código fonte da interface *Command*
Fonte: O próprio autor.

A Figura 71 apresenta o código fonte da classe *LigaTVComando* que representa o comando de se ligar uma TV. Como pode-se perceber, todo comando deve implementar a interface *Command*, mostrada na Figura 70, e possuir o objeto de destino cuja ação vai ser realizada. Nesse exemplo ligar o aparelho de TV. Nota-se que o método *execute* deve realizar a ação do comando, representada nesse caso pelo método *ligar()*. Essa ação necessariamente não precisa ser única, poder-se-ia ter chamadas a vários métodos do objeto TV ou a métodos de outros objetos, a quantidade de chamadas necessárias para realizar o comando.

O comando desse exemplo também possui o método *undo()* que realiza as ações necessárias para desfazer as ações anteriores, nesse exemplo desligar a TV.

```

public class LigaTVComando implements Command{

    TV tv; //quem vai executar a tarefa

    public LigaTVComando(TV tv){
        this.tv = tv;
    }

    // metodo sobre escrito da interface
    public void execute(){
        tv.ligar();
    }

    public void undo(){
        tv.desligar();
    }

}

```

Figura 71: Código fonte da classe *LigaTVComando*
Fonte: O próprio autor.

É necessário também possuir um objeto auxiliar, geralmente chamada de *Invoker*, Figura 72, que encapsula as chamadas aos comandos. Esse objeto deve possuir um comando (objeto que implemente a interface *Command*, mostrada na Figura 70) e, no mínimo, um método que chama o método *execute* do objeto comando para realizar as ações necessárias.

No caso da necessidade de desfazer as ações dos comandos, esse objeto se encarregar de guardar o último comando executado e realizar a chamada ao método *undo()* do mesmo. Se houvesse a necessidade de se desfazer os últimos N comandos executados, este objeto deveria possuir uma lista dos últimos comandos executados e possuir a lógica para realizar essa operação.

```
/** classe que o cliente vai chamar quando o
 *botao for precionado
 *essa classe vai executar o comando */
public class Invoker {

    Command comando;
    Command ultimoComando; //guarda o ultimo comando executado

    public void setCommand(Command comando) {
        this.comando = comando;
    }

    public void butaoLigarFoiPrecionado() {
        comando.execute();
        ultimoComando = comando;
    }

    public void botaoDesligarFoiPrecionado() {
        ultimoComando.undo();
    }

}
```

Figura 72: Código fonte da classe *Invoker*
Fonte: O próprio autor.

Na Figura 73 tem-se o código da classe cliente que invoca a realização do comando. Nota-se que ela possui uma instância do objeto *Invoker*, do objeto cuja ação vai ser realizada (tv) e do comando que vai ser invocado (*LigaTVComando*). O cliente adiciona o objeto *TV* ao comando *LigaTVComando* de depois adiciona o comando ao *Invoker*. Em um momento oportuno posterior, o método do objeto *Invoker* é chamado, (*butaoLigarFoiPrecionado()*), e a ação do comando que o objeto *Invoker* possuía será executada.

```

public class Cliente {
    public static void main(String[] args) {
        Invoker invoker = new Invoker();

        TV tv = new TV();
        LigaTVComando liga = new LigaTVComando(tv);
        invoker.setCommand(liga);

        invoker.butoaoLigarFoiPrecionado();
        invoker.butoaoDesligarFoiPrecionado();
    }
}

```

Figura 73: Código fonte da classe *Cliente*
Fonte: O próprio autor.

Em uma situação mais complexa na qual seja necessário executar vários comandos, pode-se criar uma classe *MacroComando* que, ao ser executada, na verdade vai executar os vários comandos que a compõem. Um exemplo simples de uma classe *MacroComando* pode ser visto na Figura 74.

```

public class MacroComando implements Command{
    Command[] comandos;

    public MacroComando(Command[] comandos){
        this.comandos = comandos;
    }

    public void execute(){
        for(int i = 0; i < comandos.length ; i++){
            comandos[i].execute();
        }
    }
}

```

Figura 74: Código fonte da classe *MacroComando*
Fonte: O próprio autor.

Nesse capítulo foram apresentados os padrões da camada de aplicação, no próximo, serão apresentados os padrões da camada que ainda está faltando, a camada de apresentação.

7 PADRÕES DA CAMADA DE APRESENTAÇÃO

Serão apresentados neste capítulo padrões para a camada de apresentação. A camada de Apresentação é a parte do sistema onde ficam os objetos responsáveis por fornecer algum serviço a uma entidade externa ao mesmo, seja essa entidade uma pessoa e a resposta em páginas HTML ou Swing; ou até mesmo outro sistema e a resposta em XML ou arquivos texto.

7.1 MODEL VIEW CONTROLER (MVC)

7.1.1 Conceito

O MVC é um padrão que serve para organizar as interações entre os componentes da camada de apresentação. Esse padrão define três tipos de componentes:

- *View* são componentes que geram a parte da aplicação visível ao usuário do sistema. Pode ser uma página JSP, uma Janela Swing, um arquivo XML, entre outros. A *View* deve implementar o padrão *Observer* (FREEMAN et al, [199?]) em cima dos objetos do *Model*. Sempre que um objeto do *Model* mudar isso deve ser refletido nas *Views* que possam estar associadas ao referido objeto e conseqüentemente, nos dados que são mostrados ao usuário.
- *Model* são os componentes que contém as informações que vão ser mostradas na *View*. Pode ser o próprio objeto do modelo da aplicação (objeto da Camada de Modelo, Capítulo 4) ou apenas uma interface para os objetos do modelo.
- *Controler* são os componentes responsáveis por capturar solicitações vindas da *View* e as despacha para o *Model*. Além de atualizar a *View* apropriadamente. Para sistemas simples, onde a camada de aplicação não possui muitas funcionalidades, o *Controler* acaba as implementando.

É bom ressaltar, os três tipos de componentes acima pertencem a Camada de Apresentação e não são camadas separadas do sistema. Mas para aplicações simples, é

aceitável *Controler* seja utilizando como substituto da Camada de Aplicação e *Model* como substituto da Camada de Modelo¹⁷.

O MVC com todas as suas funcionalidades não é possível de ser implementado em aplicações Web, visto que o protocolo HTTP não permite que o servidor, onde está o *modelo*, notifique o cliente onde está a *View*. O que realmente é implementado em aplicações Web é o padrão *Front Controller* (FOWLER, et al, 2002).

Normalmente não é preciso implementar esse padrão quando vai se desenvolver um sistema. Muitos Frameworks que trabalham com a Camada de Apresentação já o implementam, basta apenas utilizá-lo, como é o caso dos componentes Swing para aplicações desktop em Java.

Existem duas discussões sobre o padrão MVC: separar o *Controler* da *View* e separar a *View* do *Model* (FOWLER, et al, 2002). A primeira é menos importante e deve ser feita apenas quando a situação realmente exigir. A separação da *View* do *Model* é umas das mais importantes indicações de bom design de software. Entre as inúmeras vantagens pode-se citar:

- Desenvolvedores se especializam em apenas em das áreas, ou no design e técnicas de usabilidade de software ou construção de lógica de negócio, acesso a Banco de Dados, desempenho, entre outros. É difícil encontrar um desenvolvedor que seja bom nas duas áreas (FOWLER, et al, 2002).
- Geralmente, softwares devem mostrar a mesma informação de maneiras diferente, dependendo do contexto que se tenha, locais e meios de acesso distintos, usuários com diferentes permissões, separar *View* do *Model* permite desenvolver múltiplas apresentações para o mesmo conteúdo.
- Objetos não visuais são mais fáceis de serem testados.

O ponto chave para essa separação é que a *View* pode depender do *Model*, mas o *Model* deve ser independente da *View*. O esquema de dependência do padrão MVC é mostrado na Figura 75.

¹⁷ Apesar do *Model* (modelo em inglês) ter o mesmo nome de uma camada eles são coisas diferentes. O *Model* de MVC não é uma camada, ele pertence à Camada de Apresentação. Apesar de pode ser usado em substituição à Camada de Modelo em sistemas pequenos.

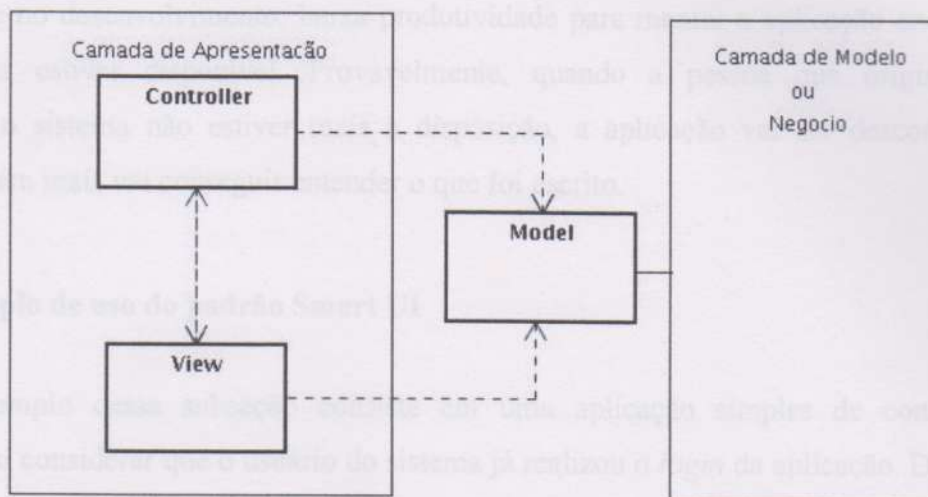


Figura 75: Esquema de dependências do padrão MVC
Fonte: O próprio autor.

7.2 SMART UI

7.2.1 Conceito

Esse padrão é a situação oposta a de se usar os demais padrões descritos durante o texto. Só se justifica para aplicações pequenas e simples que apenas recuperam dados do Banco de Dados e os mostram para o usuário do sistema, sem nenhuma modificação ou regra de negócio. Para aplicações corporativas *Smart UI* é considerado um *Anti-Pattern*¹⁸ (CALÇADO, 2005, p.38-49)

A razão da existência dessa subseção é apenas a de comparar o código de uma aplicação feita usando *Smart UI* com o código gerando usando-se os demais padrões desse texto.

Esse padrão basicamente consiste em não usar padrão algum. Nele, o código responsável por gerar a interface com o usuário é misturado com o código das regras de negócio, com o código de acesso aos dados no Banco de Dados (geralmente SQL), com o código para implementar qualquer funcionalidade que a aplicação possuir (segurança por exemplo).

Conseqüências desse padrão são: código repetido e desorganizado, alto acoplamento entre as partes do sistema, baixa coesão e quase nenhuma separação de responsabilidades, alta

¹⁸ Um *Anti-Pattern* descreve como um problema não deve ser modelado. Da mesma forma que um *Pattern*, muitas pessoas já utilizaram uma determinada solução para resolver algum problema específico que acabou gerando outros problemas ainda maiores, para impedir que outras pessoas passem pelo mesmo problema resolveu-se catalogá-la.

produtividade no desenvolvimento, baixa produtividade para manter a aplicação enquanto o desenvolvedor estiver disponível. Provavelmente, quando a pessoa que originalmente desenvolveu o sistema não estiver mais a disposição, a aplicação vai ser descontinuada, porque ninguém mais vai conseguir entender o que foi escrito.

7.2.2 Exemplo de uso do padrão Smart UI

O exemplo dessa subseção consiste em uma aplicação simples de consulta de processos. Vai considerar que o usuário do sistema já realizou o *login* da aplicação. Depois de estar autenticado, o usuário é redirecionado para a página de consulta de processos. Essa página, mostrada da Figura 79, possui um formulário no qual o usuário digita um número de processo, o sistema realiza uma consulta no banco e, se o processo existir, os dados são mostrados na mesma página.

Seguindo o padrão *Smart UI*, todo o código fonte da aplicação está na camada de apresentação. Como esse exemplo é simples e pequeno ele está todo em uma única página web como é mostrado nas Figuras 76, 77 e 78. Nessa página é misturado código para visualização (HTML), código que realiza as regras de negócio (Java) e código para consultar os dados no Banco de Dados (SQL).

Como pode-se perceber, até para uma aplicação muito pequena igual a essa, que tem caráter apenas ilustrativo e dificilmente vai ser utilizando em uma situação real, o código fica muito desorganizado e difícil de entender.

```
<%! private Connection con; %>
<%
    try{
        Class.forName("org.postgresql.Driver");
        String url = "jdbc:postgresql://localhost:5432/smartUI";
        con = DriverManager.getConnection(url, "postgres", "postgres");
    }catch (ClassNotFoundException ex) {
        out.println("Não encontrou o drive do banco");
    }catch (SQLException e){
        out.println("Erro na conexão com o banco");
    }
%>
```

Figura 76: Código fonte da página de consulta de processos usando *Smart UI*
Fonte: O próprio autor.

```

<html>
<head> <title>Smart UI Exemplo</title> </head>
<body>
    <h1>Smart UI Exemplo</h1>
    <ul>
        <li> Bem Vindo
        <%
            // busca o nome do usuario no banco
            try{
                String sql= "SELECT nome FROM usuario WHERE id = 1";
                Statement st = con.createStatement();
                ResultSet results = st.executeQuery(sql);

                while ( results.next()) {
                    <% out.println(results.getString(1));%> </B>
                    <%
                        }
                    st.close();
                }
                catch (SQLException e){
                    out.println("Erro na conexão com o banco");
                }
            <%
                <li>Data Atual: <B><%= new Date() %></B>
            </ul>
            <%
                String numeroProcesso = request.getParameter("numeroProcesso");
                if (request.getParameter ("numeroProcesso") != null
                    && request.getParameter("numeroProcesso")
                        .equals("") == false){
                    out.println("Informações do Processo:");
                }
            <%
                <br><br>

```

Figura 77: Continuação do Código Fonte da página de consulta de processos usando *Smart UI*
Fonte: O próprio autor.

```

<%
    try{
        String sql= "SELECT * FROM Processo WHERE numero = "
                    +request.getParameter("numeroProcesso");
        Statement st = con.createStatement();
        ResultSet results = st.executeQuery(sql);
    %>
<CENTER>
<%
    while (results.next()) {%>
        Número: <% out.println(results.getString(2));%>
        <BR>Réu:<%out.println(results.getString(3));%>
        <BR>Autor:<% out.println(results.getString (4));%>
        <BR>Situação:<% out.println(results.getString(5));%>
    </CENTER>
    <%
        }
        st.close();
        con.close();
    }catch(SQLException e){
        out.println("Erro na conexão com o banco");
    }
    } %>
<BR><BR><BR><BR><BR>
    <form name="formulario de busca" action="#" method="POST">
        Digite o numero do Processo:
        <input type="text" name="numeroProcesso" size="20" /><BR>
        <input type="submit" value="Procurar Processo" />
    </form>
</body>
</html>

```

Figura 78: Continuação do Código Fonte da página de consulta de processos usando *Smart UI*
 Fonte: O próprio autor.

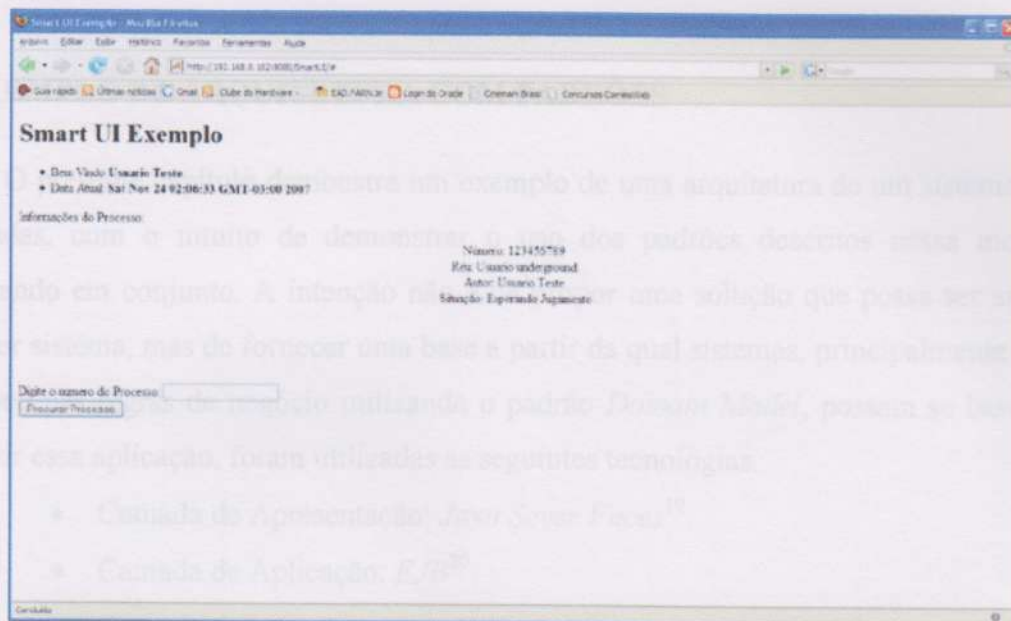


Figura 79: Página de Consulta de Processos usando o padrão *Smart UI*
Fonte: O próprio autor.

Nesse capítulo, foi finalizado o estudo dos padrões que foram propostos nesse trabalho, seguindo a divisão nas quatro camadas principais que o DDD aconselha que seja utilizada. No próximo capítulo, é apresentada uma aplicação que procura mostrar como esses padrões interarem quando estão sendo usados em conjunto.

8 EXEMPLO DE ARQUITETURA COM PADRÕES

O presente capítulo demonstra um exemplo de uma arquitetura de um sistema simples de vendas, com o intuito de demonstrar o uso dos padrões descritos nessa monografia trabalhando em conjunto. A intenção não é de propor uma solução que possa ser aplicada a qualquer sistema, mas de fornecer uma base a partir da qual sistemas, principalmente sistemas com foco nas regras de negócio utilizando o padrão *Domain Model*, possam se basear. Para construir essa aplicação, foram utilizadas as seguintes tecnologias.

- Camada de Apresentação: *Java Sever Faces*¹⁹.
- Camada de Aplicação: *EJB*²⁰.
- Camada de Infra Estrutura: *Hibernate*²¹.

A arquitetura possui quatro camadas principais: Apresentação, Aplicação, Modelo e Infra Estrutura ou sendo mais específico, Camada de Persistência, como se pode vê na Figura 80.

Na camada de Apresentação existem as páginas JSP e os *ManagedBeans*²² do JSF. A Figura 81 mostra o código da classe *AbstractManagedBean*, que implementa o padrão *Layer SuperType* (Capítulo 3) para a camada de Apresentação. Na Figura 82 tem-se o *ManagedBean CompraBean*, esse componente está associado ao Caso de Uso *Realizar Compra*. Ele recebe os dados vindos do formulário da página de realização de compras e faz uma chamada remota à classe *GerenciadorCompraImpl*.

A classe *GerenciadorCompraImpl*, que está na camada de Aplicação, implementa o padrão *Facade* (Capítulo 6) para limitar e controlar o acesso às classes da camada de Modelo; ao mesmo tempo implementa um *Service* do *Domain Driven Design* (Capítulo 4), pois é uma classe que contém informações que não pertence a uma classe específica do Modelo e controla o fluxo com que as ações dos objetos dessa camada são executadas; além de ser um componente EJB. Como todas as informações nessa arquitetura precisam passar pela *Facade* da camada de Aplicação, esse é um local bom para implementar outras características não funcionais, como transação ou segurança, essas características são implementadas utilizando a tecnologia EJB. O código da classe *GerenciadorCompraImpl* é mostrado na Figura 83.

¹⁹ <http://java.sun.com/javaee/javaxserverfaces/>

²⁰ <http://java.sun.com/products/ejb/>

²¹ <http://www.hibernate.org/>

²² Uma *ManagedBean* é uma classe cadastrada no JSF que permite, entre outras coisas, que seus métodos sejam invocados diretamente de um formulário HTML em uma página JSP.

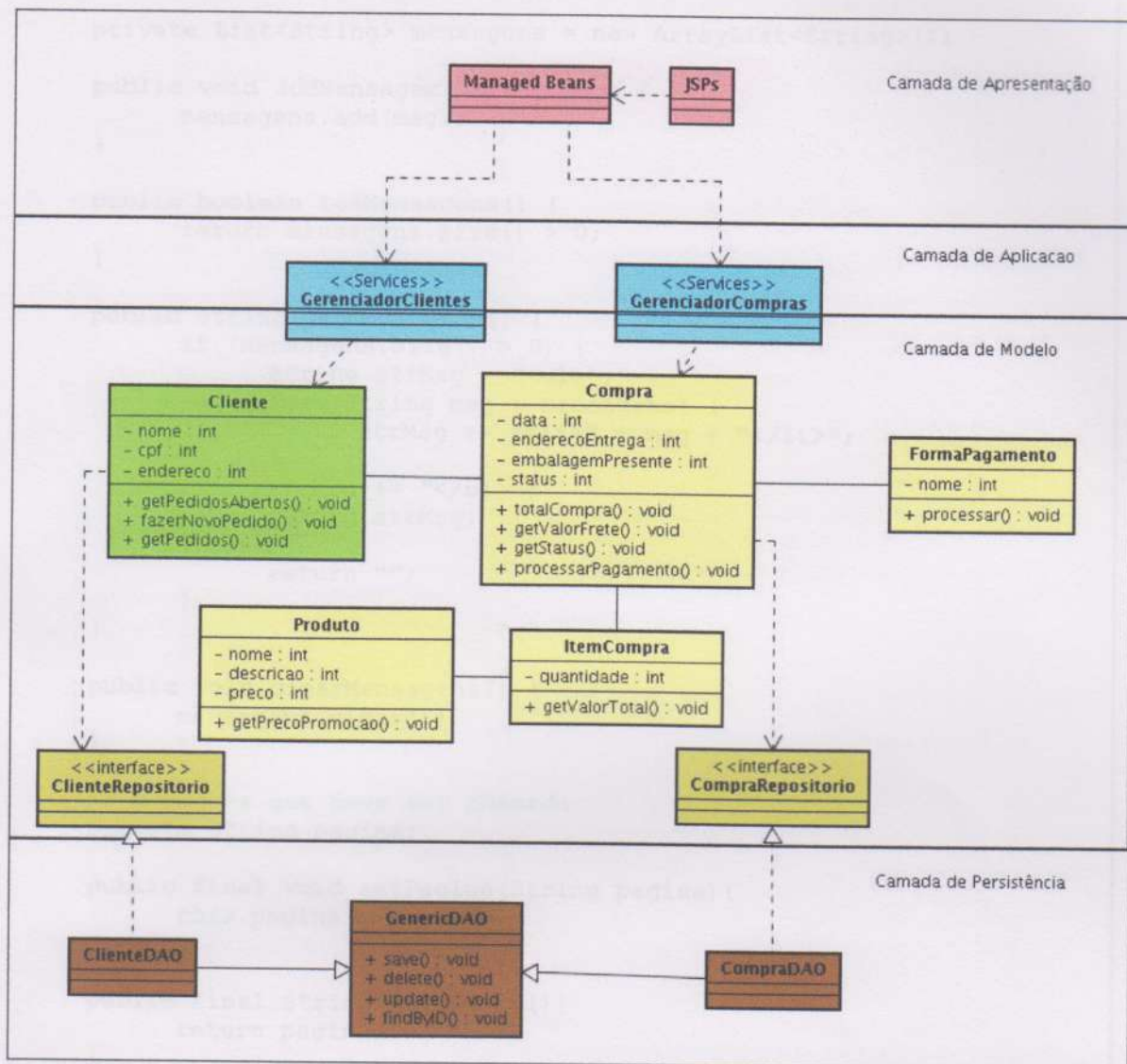


Figura 80: Arquitetura de um sistema de vendas utilizando padrões.
Fonte: O próprio autor.

A classe *GerenciadorCompraImpl* manipula os objetos do modelo e salva suas informações por meio das classes *CompraRepositorio* e *ClienteRepositorio*. Essas classes implementam o padrão Repositório (Capítulo 5), o que deixa as classes do Modelo independentes da camada de Persistência. O código da classe *CompraRepositorio* é mostrado na Figura 86. O código classe *ClienteRepositorio* é semelhante ao código da classe *CompraRepositorio* e foi omitido.

```

public abstract class AbstractManagedBean {

    private List<String> mensagens = new ArrayList<String>();

    public void addMensagem(String msg) {
        mensagens.add(msg);
    }

    public boolean temMensagens() {
        return mensagens.size() > 0;
    }

    public String getMensagens() {
        if (mensagens.size() > 0) {
            String strMsg = "<ul>";
            for (String msg : mensagens) {
                strMsg += "<li>" + msg + "</li>";
            }
            strMsg += "</ul>";
            return strMsg;
        } else {
            return "";
        }
    }

    public void clearMensagens() {
        mensagens.clear();
    }

    // a pagina que deve ser chamada
    private String pagina;

    public final void setPagina(String pagina) {
        this.pagina = pagina;
    }

    public final String getPagina() {
        return pagina;
    }

    /**
     * Indica para qual pagina redirecionar (a pagina associada ao bean)
     * @return a string da navegacao
     */
    public abstract String exibirPagina();
}

```

Figura 81: A classe *AbstractManagedBean* da camada de Apresentação.

Fonte: O próprio autor.

```

public class CompraBean extends AbstractManagedBean {

    private GerenciadorCompra gerenciador;

    private List<Produto> produtos;

    private Cliente cliente;

    private String enderecoEntrega;

    private boolean embalarPresente;

    private FormaPagamento formaPagamento;

    public CompraBean() {

        try {
            InitialContext ctx = new InitialContext();
            gerenciador = (GerenciadorCompra) ctx.lookup(
                "lojapadrees/GerenciadorCompraImpl/remote");
        } catch (Exception e) {
            // tratamento da exceção
        }

        GenericDAO dao = new GenericDAO();
        produtos = (List<Produto>) dao.findAll(Produto.class);

    }

    public List<Produto> getProdutos() {
        return produtos;
    }

    public String realizarCompra() {
        gerenciador.realizarCompra(cliente, enderecoEntrega,
            embalarPresente, formaPagamento, produtos);
        addMensagem("Compra realizada com sucesso");
        return "index"; // retorna para a pagina inicial
    }

    @Override
    public String exibirPagina() {
        return getPagina();
    }

}

```

Figura 82: A classe *CompraBean* da camada de Apresentação
Fonte: O próprio autor.

```

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class GerenciadorCompraImpl implements GerenciadorCompra {

    //instanciados por injeção de dependências
    CompraRepositorio repositorio1;
    ClienteRepositorio repositorio2;

    @Override
    @TransactionAttribute(TransactionAttributeType.MANDATORY)
    public void realizarCompra(Cliente cliente, String enderecoEntrega,
                               boolean embalarPresente, FormaPagamento formaPagamento,
                               List<Produto> produtos) {

        Compra compra = new Compra(new Date(), enderecoEntrega,
                                    embalarPresente, formaPagamento, produtos);

        cliente.fazerNovoPedido(compra);

        try {
            repositorio2.add(cliente);
            repositorio1.add(compra);
        } catch (Exception e) {
            // tratamento da exceção
        }
    }
}

```

Figura 83: A classe *GerenciadorCompraImpl* da camada de Aplicação
Fonte: O próprio autor.

Nas Figuras 84 e 85 está o código de uma classe que representa as classes da camada de Modelo, a classe *Compra*. Ela possui tanto os dados como as regras de negócio de sua competência e não possui nenhuma referência a classes de outras camadas.

Nesse exemplo, a classe *Compra* é a raiz da agregação formada pelas classes *Compra*, *ItemCompra*, *FormaPagamento* e *Produto*. Sendo assim, nenhuma classe fora da agregação pode fazer referências às demais classes da agregação, somente a classe raiz, nesse exemplo a classe *Compra*.

```

@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Compra{

    @Id private int id;

    private Date data;

    private String enderecoEntrega;

    private boolean embalarPresente;

    private Status status = Status.ABERTA;

    private FormaPagamento formaPagamento;

    @ManyToOne
    private List<ItemCompra> itensCompra;

    public Compra(Date data, String enderecoEntrega, boolean
embalarPresente, FormaPagamento formaPagamento, List<Produto> produtos){

        this.data = data;
        this.enderecoEntrega = enderecoEntrega;
        this.embalarPresente = embalarPresente;
        this.formaPagamento = formaPagamento;

        criaItensCompra(produtos);
    }

    private void criaItensCompra(List<Produto> produtos){
        itensCompra = new ArrayList<ItemCompra>();

        // para cada produto diferente cria um item compra direferente
        for (Produto produto : produtos) {
            ItemCompra item = new ItemCompra(produto);
            itensCompra.add(item);
        }
    }

    public float getTotalCompra(){
        float total = 0.0f;

        for (ItemCompra item : itensCompra) {
            total += item.getValorTotal();
        }

        total += this.getValorFrete();

        if(embalarPresente)
            total += 2.00f;

        total = total * formaPagamento.processarDesconto();

        return total;
    }
}

```

Figura 84: A classe *Compra* da camada de Modelo
Fonte: O próprio autor.

```

public float getValorFrete(){
    if(enderecoEntrega.equals("Rio Grande do Norte")){
        return 130.86f;
    }else{
        return 15.45f;
    }
}

public Status getStatus() {
    return status;
}

public Date getData(){
    return data;
}

/*
 * ABERTA = compra aberta pelo cliente
 * ESPERANDO_PAGAMENTO = cliente confirmou a compra mais nao pagou
ainda
 * ESPERANDO_ENTREGA = cliente confirmou pagamento, e a compra foi
enviada
 * CONCLUIDA = a entrega foi confirmada
 */
public enum Status {
    ABERTA, ESPERANDO_PAGAMENTO, ESPERANDO_ENTREGA, CONCLUIDA
}
}

```

Figura 85: A classe *Compra* da camada de Modelo continuação.

Fonte: O próprio autor.

```

public interface CompraRepositorio {

    public void add(Compra c) throws Exception;

}

```

Figura 86: A interface *CompraRepositorio* da camada de Modelo

Fonte: O próprio autor.

Por último, são mostradas as classes responsáveis pela persistência dos objetos. A classe *GenericDAO*, Figura 87, que também implementa o padrão *Layer SuperType*, mas agora para a camada de persistência e possui os métodos de persistência comuns a todos os objetos do modelo. E na Figura 88 tem-se a classe *CompraDAO*. Essa classe possui os métodos específicos para a classe *Compra* e implementa o repositório dela.

```

public class GenericDAO {

    protected Session session;

    public GenericDAO() {
        if (session == null) {
            session = SessionFactorySingleton.getInstance()
                .getSession();
        }
        return session;
    }

    public void close() {
        if (session != null) session.close();
    }

    private void change(Object o, char t){
        if (t == 'C') {
            session.save(o);
        } else if (t == 'D') {
            session.delete(o);
        } else if (t == 'U') {
            session.saveOrUpdate(o);
        }
    }

    public void save(Object o) {
        change(o, 'C');
    }

    public void delete(Object o) {
        change(o, 'D');
    }

    public void update(Object o) {
        change(o, 'U');
    }

    public Collection findAll(Class c) {
        Collection col = session.createCriteria(c).list();
        return col;
    }

    public Object findByPrimaryKey(Class c, Serializable id) {
        Object o = session.get(c, id);
        return o;
    }

    public List findLikeCampo(String classe, String campo, String valor)
    {
        Query q = session.createQuery( "from " + classe + " obj "
            + " where obj." + campo + " like '" + valor + "%'" );
        return q.list();
    }
}

```

Figura 87: A classe *GenericDAO* da camada de Infra Estrutura.

Fonte: O próprio autor.

```
public class CompraDAO extends GenericDAO implements CompraRepositorio{  
    @Override  
    public void add(Compra c) throws Exception {  
        save(c);  
    }  
}
```

Figura 88: A classe *CompraDAO* da camada de Infra Estrutura.

Fonte: O próprio autor.

Nesse capítulo foi mostrado um exemplo de uma aplicação que buscou apresentar alguns dos padrões descritos nesse trabalho podem interagir em conjunto em um sistema. No próximo capítulo são apresentadas as conclusões do trabalho.

Mostrou-se também vantagens no sentido de deixar o modelo o mais próximo possível do domínio da aplicação e independente das demais camadas do sistema, melhorando assim a flexibilidade desse, o entendimento do modelo pelos desenvolvedores e comunicação entre estes e os clientes. Essa melhoria na comunicação implica em diálogos mais integrados no desenvolvimento, facilitando o seu entendimento no que diz respeito aos recursos, custos e tempo despendidos durante o projeto de um sistema.

Trabalho futuro dando seguimento a pesquisa realizada nesse trabalho poderia seguir duas linhas: Primeiro, aumentar a lista de padrões descritos acrescentando padrões que se propõem a resolver determinados problemas que não foram abordados nesse trabalho. Em segundo lugar, implementar sistemas utilizando os padrões descritos nesse trabalho em algum outro e compará-los com outros construídos sem o uso desses padrões, buscando obter dados com relação a linhas de código, número de classes, tamanho das classes e métodos, desempenho da aplicação, tempo gasto para implementar determinado problema, tempo gasto

9 CONCLUSÕES

Este trabalho mostrou-se bastante proveitoso uma vez que conseguiu-se descrever detalhadamente e com exemplos os principais padrões envolvidos no desenvolvimento de um sistema corporativo, dando uma noção dos principais problemas e soluções para essa linha de desenvolvimento. Não existe uma solução única, um padrão que pode ser usado em todas as aplicações e funcione bem para todos os problemas, cada aplicação possui um conjunto de particularidades que devem ser sempre levadas em consideração na hora da escolha de quais padrões utilizar. O que se procurou com esse trabalho foi descrever padrões que fornecessem uma base para quem precisa desenvolver uma arquitetura orientada a objetos de um sistema corporativo.

Durante este trabalho deu-se destaque, pelo uso de padrões como o *Domain Model* e técnicas como o *Domain Driven Design*, a como se modelar um sistema que utilize realmente as vantagens da orientação a objetos, tais como: o real uso do encapsulamento, eliminando métodos *sets* e *gets* que apenas serviam como uma maneira mais trabalhosa de tornar os atributos públicos; possibilidade dos objetos manterem a consistência dos seus dados, diminuindo os pontos de erros no sistema; aumentar a reusabilidade do código, pois eliminam-se objetos que possuíam regras de negócio muito especializadas, que serviam apenas para atender uma situação específica, distribuindo essas regras pelos objetos que possuem a competência para executá-las.

Mostrou-se também vantagens no sentido de deixar o modelo o mais próximo possível do domínio da aplicação e independente das demais camadas do sistema, melhorando assim a flexibilidade desse, o entendimento do modelo pelos desenvolvedores e comunicação entre estes e os clientes. Essa melhora na comunicação implica em clientes mais integrados ao desenvolvimento, facilitando o seu entendimento no que diz respeito aos recursos, custos e tempo despendidos durante o projeto de um sistema.

Trabalho futuros dando seguimento a pesquisa realizada nesse trabalho poderiam seguir duas linhas: Primeiro, aumentar a lista de padrões descritos acrescentando padrões que se propõem a resolver determinados problemas que não foram abordados nesse trabalho. Em segundo lugar, implementar sistemas utilizando os padrões descrito nesse trabalho ou algum outro e compará-los com outros construídos sem o uso desses padrões, buscando obter dados com relação a linhas de código, número de classes, tamanho das classes e métodos, desempenho da aplicação, tempo gasto para implementar determinado problema, tempo gasto

para realizar mudanças nas regras de negócio ou nos requisitos funcionais e não funcionais, clareza do código, entre outros. Tal trabalho terá o intuito de provar, pelos números obtidos em uma situação real, que realmente esses padrões cumprem o que eles se propõem a fazer.

Development Series. 2006 C# Made Easy. Disponível em:
<<http://www.infoq.com/bo/easycsharp-driven-design-quickly>>. Acesso em: 03 Dez 2007.

BECK, K.; CUNNINGHAM, W. Using Pattern Languages for Object-Oriented Programs. 1987. Disponível em: <<http://www.doc.ic.ac.uk/~w/ftp/objlsp87.html>>. Acesso em: 11 Dez 2007.

CALÇADO, P. Arquitetura de Casos em Java EE. Mundo Java. Rio de Janeiro, v. 3, n. 13, p.34-43, 2005.

Desenvolvendo Sistemas OO com Padrões de Negócio. Mundo Java. Rio de Janeiro, v. 3, n. 17, p.56-66, 2005.

Fantoches. Disponível em:
<<http://fragmental.com.br/wiki/index.php/tittle=Fantoches>>. Acesso em: 16 Sep 2007.

Calidade com Domain-Driven Design. Disponível em:
<<http://blog.fragmental.com.br/2007/06/22/calidade-com-domain-driven-design/>>. Acesso em: 20 Sep 2007.

Contratos e Nulos. 23 Oct 2005. Disponível em:
<http://fragmental.com.br/wiki/index.php/tittle=Contratos_Nulos>. Acesso em: 03 Oct 2007.

Strict POJOs. Disponível em: <<http://blog.fragmental.com.br/2006/05/16/strict-pojos/>>. Acesso em: 03 Oct 2007.

Evitando YOs e BOs. Disponível em:
<http://fragmental.com.br/wiki/index.php/tittle=Evitando_YOs_e_BOs>. Acesso em: 06 Oct 2007.

DRUGO D. Fundações Patterns. School of Computer Science Carleton University. Disponível em: <<http://www.scs.carleton.ca/~design/papers/foundation.pdf>>. Acesso em: 24 Nov 2007.

REFERÊNCIAS

- AVRAM, A., MARINESCU, F. **Domain-Driven Design Quickly**. InfoQ Enterprise Software Development Series. 2006 C4Media Inc. Disponível em: <<http://www.infoq.com/books/domain-driven-design-quickly>>. Acesso em: 03 Oct 2007.
- BECK, K., CUNNINGHAM, W. **Using Pattern Languages for Object-Oriented Programs**. 1987. Disponível em: <<http://c2.com/doc/oopsla87.html>>. Acesso em: 11 Dez 2007.
- CALÇADO, P. Arquitetura de Camadas em Java EE. **Mundo Java**. Rio de Janeiro, v.3, n.15, p.34-43, 2005.
- _____. Desenvolvendo Sistemas OO com Padrões de Negócio. **Mundo Java**. Rio de Janeiro, v.3, n.17, p.56-66, 2005.
- _____. **Fantoches**. Disponível em: <<http://fragmental.com.br/wiki/index.php?title=Fantoches>>. Acesso em: 16 Sep 2007.
- _____. **Cuidado com Domain-Driven Design**. Disponível em: <<http://blog.fragmental.com.br/2007/06/22/cuidado-com-domain-driven-design/>>. Acesso em: 30 Sep 2007.
- _____. **Contratos Nulos**. 23 Oct 2005. Disponível em: <http://fragmental.com.br/wiki/index.php?title=Contratos_Nulos>. Acesso em: 03 Oct 2007.
- _____. **Strict POJOs**. Disponível em: <<http://blog.fragmental.com.br/2006/05/16/strict-pojos/>>. Acesso em 03 Oct 2007.
- _____. **Evitando VOs e BOs**. Disponível em: <http://fragmental.com.br/wiki/index.php?title=Evitando_VOs_e_BOs>. Acesso em: 06 Oct 2007.
- DEUGO D. **Foundation Patterns**. School of Computer Science Carleton University. Disponível em: <<http://www.scs.carleton.ca/~deugo/papers/foundation.pdf>>. Acesso em: 24 Nov 2007.

FOWLER, M. et al. **Patterns of Enterprise Application Architecture**. [s.l.]: Addison Wesley, 2002.

FOWLER, M. **AnemicDomainModel**. Disponível em: <<http://www.martinfowler.com/bliki/AnemicDomainModel.html>>. Acesso em 06 Oct 2007.

FOWLER, M. **Inversion of Control Containers and the Dependency Injection pattern**. Disponível em: <<http://martinfowler.com/articles/injection.html>>. Acesso em 20 Oct 2007.

FRANTZ, R. Z. **Introdução a Design Patterns para o desenvolvimento de software**. [s.l.]: 2007. Disponível em: <<http://www.jeebrasil.com.br/tags/designpatterns>>. Acesso em: 18 Jun 2007.

FREEMAN, E. et al. **Head First Design Patterns**. O' Reilly. [199?]

GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. [s.l.]: Addison Wesley, 1994.

JOHNSON R. **Expert One-on-One J2EE Design and Development**. Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

MARTIN, R. C. **The Dependency Inversion Principle**. Disponível em: <www.objectmentor.com/resources/articles/dip.pdf>. Acesso em: 13 Jan 2008.

SAUVÉ, J. P. **O Padrão Strategy**. Disponível em: <<http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/pat/strategy.htm>>. Acesso em: 26 Oct 2007.

SPRING, **Spring Framework**. Disponível em: <<http://www.springframework.org/>>. Acesso em: 26 Oct 2007.

SUN MICROSYSTEMS. **Core J2EE Patterns - Data Access Object**. Disponível em: <<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>>. Acesso em: 20 Dez 2007.