Validação ágil e precisa de projetos conceituais de banco de dados

Marcos Eduardo Bolelli Broinizi

DISSERTAÇÃO APRESENTADA
AO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA
DA
UNIVERSIDADE DE SÃO PAULO
PARA OBTENÇÃO DO GRAU DE MESTRE
EM
CIÊNCIA DA COMPUTAÇÃO

Área de Concentração : **Ciência da Computação**
Orientador : **Prof. Dr. João Eduardo Ferreira**

O autor recebeu apoio financeiro da CAPES para este trabalho.

- São Paulo, dezembro de 2006 -

Validação ágil e precisa de projetos conceituais de banco de dados

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Marcos Eduardo Bolelli Broinizi
e aprovada pela Comissão Julgadora.

São Paulo, 11 de dezembro de 2006.

Banca Examinadora :

Prof. Dr. João Eduardo Ferreira (orientador) – IME-USP

Prof. Dr. Alfredo Goldman vel Lejbman – IME-USP

Prof. Dr. Alberto Henrique Frade Laender – UFMG

à minha mãe Tânia, ao meu pai Luciano
e à minha irmã Priscila

Agradecimentos

Agradeço aos meus familiares por todo o apoio recebido durante o desenvolvimento deste projeto. Aos meus colegas que auxiliaram direta ou indiretamente na concepção deste projeto. Ao meu orientador, pela paciência, auxílio e sugestões para aprimorar o desenvolvimento do projeto e a concepção do texto. Ao Instituto de Matemática e Estatística da Universidade de São Paulo e à CAPES, pelo auxílio financeiro para o programa de mestrado.

Resumo

A criação do projeto conceitual de um banco de dados que represente adequadamente um determinado domínio de aplicação continua sendo um dos principais desafios da área de banco de dados. Por outro lado, a discussão sobre métodos ágeis de desenvolvimento de *software* alcançou, recentemente, a comunidade de banco de dados. Este trabalho apresenta o projeto conceitual de bancos de dados sob a luz de métodos ágeis de desenvolvimento. Desenvolvemos uma extensão do arcabouço *Naked Objects* que permite uma validação ágil e precisa do projeto conceitual junto ao especialista do domínio. Em nossa abordagem, o projeto conceitual de bancos de dados é descrito por meio de anotações que representam as abstrações de dados em um ambiente dinâmico de validação.

Abstract

Creating a conceptual database design that adequately represents a specific application domain continues to be one of the main challenges in the database research. On the other hand, the discussion regarding agile methods of software development has recently become a subject of interest to the database community. This work presents a new approach to create a conceptual database design according to agile methods. We have created an extension of the *Naked Objects* framework that allows an agile and precise validation of the conceptual database design by the domain specialist. In our approach, the conceptual database design is described through annotations that represent data abstractions in a dynamic validation environment.

Índice

1	Introdução	1
1.1	Objetivo	2
1.2	Hipótese	2
1.3	Justificativas e principal contribuição	2
1.4	Organização do trabalho	3
2	Fundamentos	5
2.1	Abstrações de dados	5
2.1.1	Classificação	6
2.1.2	Relacionamento	6
2.1.3	Generalização-especialização	6
2.1.4	Composição	8
2.1.5	Objeto-relacionamento	8
2.2	Abordagens relacionadas	9
2.3	Desenvolvimento ágil	11
2.4	Naked Objects	13
2.4.1	Criando um Naked Object	14
2.5	Conclusão	16
3	Anotações	17
3.1	Implementação inicial da abstração de relacionamento	18
3.1.1	Implementação da abstração de relacionamento no arcabouço estendido	19
3.1.2	Implementação da abstração de relacionamento utilizando anotações	22
3.2	Anotações para as abstrações de dados	26
3.2.1	Classificação - Entidade	26

3.2.2	Generalização-especialização	28
3.2.3	Relacionamento	33
3.2.4	Composição	34
3.2.5	Objeto-Relacionamento	35
3.3	Conclusão	37
4	Ferramenta	39
4.1	Introdução	39
4.2	Núcleo da ferramenta	41
4.2.1	Padrão <i>Observer</i>	41
4.2.2	Diagrama de classes	42
4.2.3	Associação de componentes geradores à ferramenta	43
4.3	Gerador de código para Naked Objects	44
4.4	Gerador de código SQL	45
4.5	Mapa de tipos SQL para Naked Objects	45
4.6	Extensões	46
4.7	Conclusão	46
5	Estudo de caso	49
5.1	Acervo e Pessoa	49
5.2	Conclusão	57
6	Conclusões	59
6.1	Contribuições	60
6.1.1	Agilidade	60
6.1.2	Precisão	61
6.1.3	Projeto físico	62
6.2	Trabalhos futuros	62
A	Naked Objects	63
A.1	<i>Behavioural Completeness</i>	64
A.1.1	Orientação a processos de negócio	66
A.1.2	Interfaces de usuário otimizadas a tarefas	67
A.1.3	Métodos orientados a <i>use-cases</i>	68

A.1.4	O padrão <i>Model-View-Controller</i>	69
B	Catálogo de Anotações	71
B.1	Classificação - Entidade	71
B.1.1	Intenção	71
B.1.2	Motivação	71
B.1.3	Estrutura	72
B.1.4	Participantes	72
B.1.5	Consequências	72
B.1.6	Implementação	73
B.1.7	Exemplo de código	73
B.2	Generalização-especialização	75
B.2.1	Intenção	75
B.2.2	Motivação	75
B.2.3	Estrutura	76
B.2.4	Participantes	76
B.2.5	Consequências	76
B.2.6	Implementação	76
B.2.7	Exemplo de código	78
B.3	Relacionamento	82
B.3.1	Intenção	82
B.3.2	Motivação	82
B.3.3	Estrutura	82
B.3.4	Participante	83
B.3.5	Consequências	83
B.3.6	Implementação	83
B.3.7	Exemplo de código	84
B.4	Composição	91
B.4.1	Intenção	91
B.4.2	Motivação	91
B.4.3	Estrutura	92
B.4.4	Participante	92
B.4.5	Consequências	92

B.4.6	Implementação	92
B.4.7	Exemplo de código	93
B.5	Objeto-Relacionamento	101
B.5.1	Intenção	101
B.5.2	Motivação	101
B.5.3	Estrutura	102
B.5.4	Participante	102
B.5.5	Colaborações	102
B.5.6	Consequências	102
B.5.7	Implementação	102
B.5.8	Exemplo de código	103
C	Código SQL	107
C.1	Generalização-especialização	107
C.2	Objeto-relacionamento	110
D	Ferramenta - Extensões	113
D.1	Depêndencias e listas de classes	113
D.1.1	Anotações para representação das abstrações de dados	113
D.1.2	Extensões do arcabouço Naked Objects	114
D.1.3	Núcleo da ferramenta	115
D.1.4	Gerador de código para Naked Objects	117
D.1.5	Gerador de código SQL	118
D.1.6	Mapa de tipos SQL para Naked Objects	120
D.2	Criando novas anotações	120
D.2.1	Predicados compostos	120
D.2.2	Estendendo o núcleo	122
D.2.3	Estendendo os demais componentes	125

Capítulo 1

Introdução

A concepção do projeto conceitual de um banco de dados envolve a transformação de um problema real em uma representação implementável [8]. Essa transformação consiste em abstrair os dados do mundo real e construir um esquema que os represente. Esse esquema de dados é composto por um conjunto de abstrações semanticamente integradas que representam os dados [13].

Na maioria das abordagens de desenvolvimento de sistemas tradicionais, a atividade de concepção do projeto conceitual de bancos de dados encontra-se distante do especialista de domínio ¹. Etapas iniciais buscam identificar os requisitos do sistema. Etapas posteriores utilizam documentos contendo os requisitos especificados como base para o projeto conceitual de bancos de dados.

Muitos problemas de especificação conceitual necessitam de informações que somente os especialistas do domínio podem fornecer. É comum que essas informações não estejam disponíveis no momento da alteração e refinamento do projeto de banco de dados, devido à má identificação inicial dos requisitos e ao fato de os especialistas de domínio não mais estarem acessíveis para esclarecer dúvidas ou complementar as informações.

Para reduzir problemas de especificação conceitual decorrentes da má identificação dos requisitos, buscamos idéias contidas nos métodos ágeis de desenvolvimento [1, 3]. Os métodos ágeis seguem os valores e princípios descritos no Manifesto Ágil [2]. Os quatro valores defendidos pelo Manifesto Ágil são: **indivíduos e interações** em detrimento de processos e ferramentas; ***software* em funcionamento**

¹Neste trabalho, utilizamos o termo especialista de domínio de aplicação ou simplesmente *especialista de domínio* como sinônimo de *especialista de negócio*.

em detrimento de documentação detalhada; **colaboração do cliente** em detrimento de negociação de contratos; e **adaptação às mudanças** em detrimento de seguir um plano.

Autores como Schuh [28] procuram aproximar o ambiente de bancos de dados dos métodos ágeis de desenvolvimento. Em seu livro [3], Ambler compila os fundamentos para um desenvolvimento ágil de dados. Em [4], Ambler apresenta uma introdução ao desenvolvimento ágil de *software* utilizando não apenas tecnologias orientadas a objetos, mas também tecnologias de bancos de dados relacionais. Em [5], Ambler aborda diretamente a modelagem de dados partindo do domínio de aplicação.

Uma interpretação exagerada e cega desses princípios pode sugerir o abandono do projeto conceitual de bancos de dados. Ao invés disso, o projeto conceitual de bancos de dados deve ser considerado como uma etapa inicial do desenvolvimento de um sistema de computação [13]. Essa etapa consiste na exploração dos requisitos e validação dos conceitos.

1.1 Objetivo

O principal objetivo deste trabalho é incorporar os princípios estabelecidos pelos métodos ágeis ao projeto conceitual de bancos de dados, sem, contudo, abrir mão da precisão proporcionada pela correta utilização das abstrações de dados.

1.2 Hipótese

Para alcançar tal agilidade e precisão, é fundamental que a validação do projeto conceitual de bancos de dados conte com a participação do especialista de domínio. Essa é a principal hipótese que norteia este trabalho.

1.3 Justificativas e principal contribuição

O projeto conceitual de bancos de dados alcançou um sucesso considerável como meio de representação dos requisitos de dados de um domínio, principalmente quando utilizados diagramas ER ou UML. Apesar de representar o projeto conceitual com a precisão necessária, os diagramas são de difícil validação pelo

especialista de domínio. Dessa forma, sua utilização não contribui para melhorar a agilidade de concepção do projeto conceitual.

Existem diversas ferramentas que auxiliam o projeto conceitual de bancos de dados, mas elas não priorizam a interação entre o especialista de domínio e as abstrações de dados.

A principal contribuição desse trabalho é alcançar, ao mesmo tempo, precisão e agilidade na validação do projeto conceitual de bancos de dados. Para isso, propomos uma abordagem que permite validar de forma ágil o projeto conceitual de bancos de dados. A precisão é fundamentada na criação do projeto utilizando abstrações de dados [13]. Para viabilizar essa abordagem, buscamos os princípios ágeis de desenvolvimento [1, 2], tornando o projeto conceitual de bancos de dados mais fácil de compreender e, ao mesmo tempo, manipulável. Nossa opção foi inspirada nos conceitos propostos pela iniciativa *Naked Objects* (NO) [22, 26, 27]. A adoção do arcabouço *Naked Objects* como a forma de representação do projeto conceitual de bancos de dados é justificada pelo fato desse arcabouço permitir ao usuário de um sistema desempenhar o papel de solucionador de problemas, o que na perspectiva do projeto conceitual de bancos de dados significa que o especialista de domínio será o responsável por solucionar os problemas de especificação conceitual de bancos de dados por meio da manipulação e validação do projeto representado.

1.4 Organização do trabalho

Este primeiro capítulo apresenta a nossa proposta de trabalho. No Capítulo 2, resumimos os conceitos relevantes para a concepção deste projeto. No Capítulo 3, descrevemos a parte principal deste trabalho, incluindo um catálogo de definições de abstrações de dados utilizando anotações. O Capítulo 4 traz a descrição da ferramenta desenvolvida. No Capítulo 5, apresentamos um estudo de caso para um domínio real utilizando a ferramenta desenvolvida. Concluimos nossa discussão no Capítulo 6 e incluímos outras informações nos Apêndices.

Capítulo 2

Fundamentos

O principal conceito de Naked Objects [22,26,27] utilizado neste trabalho une os aspectos de precisão e agilidade: *o papel do usuário de um sistema, sob a perspectiva Naked Objects, deve ser o de solucionador de problemas*. Seguindo esse conceito, o especialista de domínio é quem possui as informações para solucionar os conflitos do projeto conceitual. Com o uso do arcabouço Naked Objects, essas informações são estimuladas a serem explicitadas no momento que os requisitos do sistema são explorados e identificados.

Entretanto, o arcabouço Naked Objects não oferece diretamente todas as abstrações necessárias para a concepção do projeto conceitual de bancos de dados. Neste trabalho, o arcabouço Naked Objects foi estendido de modo a melhor atender às várias formas de relacionamentos entre objetos e abstrações de composição e generalização-especialização.

Neste capítulo apresentamos as abstrações de dados utilizadas, abordagens para representação de projetos conceituais de bancos de dados, as características dos métodos ágeis de desenvolvimento, o arcabouço Naked Objects e como relacionamos esses diferentes conceitos para criar uma abordagem ágil para concepção do projeto de conceitual de bancos de dados.

2.1 Abstrações de dados

A utilização das abstrações de dados como um denominador comum entre o desenvolvedor e o especialista de domínio possibilita mapear comportamentos específicos para cada abstração no ambiente

de validação. Essas abstrações já foram vastamente estudadas pela área de banco de dados [9, 13, 32], constituindo o alicerce de um projeto conceitual adequado.

Nem todas as abstrações de dados possuem uma mesma notação e significado na área de computação. Apresentamos sucintamente as abstrações utilizadas neste trabalho para tornar a compreensão do texto uniforme, evitando interpretações equivocadas.

2.1.1 Classificação

A primeira abstração representada neste trabalho é a abstração de classificação [13]. Essa abstração possibilita representar um objeto do domínio como uma **classe** ou **tipo de entidade**. Um tipo de entidade contém um nome, E , e um conjunto de atributos com seus domínios.

2.1.2 Relacionamento

De acordo com [13], um **tipo de relacionamento** R entre n tipos de entidades $E_1, E_2, \dots, E_n$ é um conjunto de associações entre entidades desses tipos. Formalmente, R é um conjunto de instâncias de relacionamento r_i , no qual cada r_i associa n entidades $(e_1, e_2, \dots, e_n)$, e cada entidade e_j em r_i é um membro do tipo de entidade E_j , $1 \leq j \leq n$, sendo um tipo de relacionamento uma relação matemática em $E_1, E_2, \dots, E_n$. Dizemos que cada um dos tipos de entidade $E_1, E_2, \dots, E_n$ **participa** do relacionamento R , assim como cada uma das entidades específicas $(e_1, e_2, \dots, e_n)$ participa da instância de relacionamento $r_i = (e_1, e_2, \dots, e_n)$. Neste trabalho consideramos apenas relacionamentos binários e suas respectivas restrições de cardinalidade: **um para um** (1 : 1); **um para muitos** (1 : N); **muitos para um** (N : 1); e **muitos para muitos** (N : N).

2.1.3 Generalização-especialização

Especialização é o processo de definir um conjunto de subclasses de um tipo de entidade. A classe que foi especializada é então denominada superclasse da especialização. O conjunto de subclasses que formam a especialização é definido com base em algumas características distintas das entidades da superclasse. É possível termos diversas especializações do mesmo tipo de entidade tendo como base diferentes características. A abstração de **generalização** pode ser entendida como a operação inversa da espe-

cialização, na qual as diferenças entre diversos tipos de entidade podem ser suprimidas, identificando as características comuns, e **generalizando-as** em uma única superclasse da qual os tipos de entidade originais são subclasses. Elmasri e Navathe [13] assim formalizaram esses conceitos ¹:

*Uma **classe** é um conjunto ou coleção de entidades (...). Uma **subclasse** S é a classe cujas entidades devem sempre ser de um subconjunto de entidades de outra classe, chamada **superclasse** C do relacionamento **superclasse/subclasse** (ou É-UMA). Denotamos tal relacionamento por C/S . Onde devemos sempre ter*

$$S \subset C.$$

*Uma **especialização** $Z = \{S_1, S_2, \dots, S_n\}$ é um conjunto de subclasses que possuem a mesma superclasse G , ou seja, G/S_i é um relacionamento superclasse/subclasse para $i = 1, 2, \dots, n$. Assim G é chamado **tipo de entidade generalizado** (ou a **superclasse** da especialização, ou a **generalização** das subclasses $\{S_1, S_2, \dots, S_n\}$). Z é considerada **total** se sempre tivermos (em algum ponto do tempo)*

$$\bigcup_{i=1}^n S_i = G.$$

*Senão, Z é considerada **parcial**. Por outro lado, Z é considerada **disjunta** se sempre tivermos*

$$S_i \cap S_j = \emptyset \text{ (conjunto vazio) para } i \neq j.$$

*Senão, Z é considerada **sobreponível**.*

*Uma subclasse S de C é considerada **definida por predicado** se o predicado p sobre os atributos de C é usado para especificar quais entidades em C pertencem a S , ou seja, $S = C[p]$ onde $C[p]$ é o conjunto de entidades em C que satisfazem p . Uma subclasse que não é definida por um predicado é chamada **definida pelo usuário**.*

*Uma especialização Z (ou generalização) é considerada **definida por atributos** se um predicado $(A = c_i)$, onde A é um atributo de G e c_i é um valor constante do domínio de A , é usado para determinar quais entidades em C pertencem a cada subclasse S_i de Z . Note que se $c_i \neq c_j$ para $i \neq j$, e A é um atributo monovalorado, então a especialização será disjunta.*

¹Esse trecho é uma tradução livre de [13].

Para este trabalho, um tipo de entidade E pode ser uma especialização de no máximo um outro tipo de entidade, ou seja, não existe herança múltipla. Um determinado tipo de entidade não deve ser, ao mesmo tempo, ancestral e descendente de um mesmo tipo de entidade. Dessa forma, podemos construir apenas hierarquias no formato de florestas.

2.1.4 Composição

Neste trabalho denominaremos por **abstração de composição** o conceito de abstração para construir um objeto composto a partir dos objetos que o compõem. Dessa forma, essa abstração pode ser entendida como o relacionamento entre o todo e suas partes [13]. Muitas vezes essa mesma abstração é apresentada com o nome de *agregação* como em [32]. A abstração de composição pode ainda ser classificada como: física ou lógica. A principal diferença estrutural entre essas duas classificações está relacionada à exclusão. Ao excluir uma instância do objeto composto definido em uma determinada composição lógica, os objetos participantes podem continuar existindo. No caso da composição física tal exclusão significa a exclusão de todos os objetos que o compõe.

Não existe um consenso na área de computação para a denominação dessa abstração. Em orientação a objetos, o conceito de composição lógica, em UML [34], por exemplo, também é denominado de *agregação*, enquanto a composição física é referenciada como *agregação de composição*.

2.1.5 Objeto-relacionamento

Optamos por chamar de **objeto-relacionamento** a idéia comum de representar um relacionamento como uma entidade própria. Em muitos modelos não existe uma representação direta de relacionamentos que possuem dados próprios ou relacionamentos que se relacionam com outras entidades além daquelas que os definem. Esse conceito possui muitas denominações. Na versão em português de [20], ao discutir a representação de relacionamentos entre relacionamentos no modelo entidade relacionamento, o termo utilizado é *agregação*:

Agregação é uma abstração através da qual relacionamentos são tratados como entidades de nível mais alto.

Assim, o conceito de objeto-relacionamento une os conceitos de entidade e de relacionamento. Ou seja,

representa a idéia de que um relacionamento tornou-se um objeto.

2.2 Abordagens relacionadas

Em [31], Teorey apresenta as principais etapas da metodologia de projeto no contexto do ciclo de vida de um banco de dados e ilustra as principais abordagens utilizadas. Essas etapas podem ser classificadas como *análise de requisitos* e *modelagem de dados conceitual*.

A análise de requisitos constitui entrevistar os especialistas do domínio, determinando a finalidade e o que o banco de dados precisa conter. Os objetivos básicos da análise de requisitos, listados em [31], são:

- delinear os requisitos de dados da empresa em termos dos elementos de dados básicos;
- descrever a informação sobre os elementos de dados e os relacionamentos entre eles necessários para modelar esses requisitos de dados;
- determinar os tipos de transação que devem ser executadas no banco de dados e a interação entre as transações e os elementos de dados;
- definir quaisquer restrições de desempenho, de integridade, de segurança ou administrativas que tenham que ser impostas sobre o banco de dados resultante;
- especificar quaisquer restrições de projeto e de implementação, tais como tecnologias, *hardware* e *software*, linguagens de programação, políticas, padrões ou interfaces externas específicas;
- documentar por completo todos os itens anteriores em uma especificação de requisitos detalhada. Os elementos de dados também podem ser definidos em um sistema de dicionário de dados, normalmente fornecido como parte integral do sistema de gerenciamento de banco de dados.

A modelagem de dados conceitual normalmente é feita ao mesmo tempo que a análise de requisitos, podendo ser considerada parte da análise. As principais atividades dessa etapa, descritas em maiores detalhes por Teorey em [31], são:

- classificar entidades e atributos;
- identificar as hierarquias de generalização;

- definir relacionamentos.

Neste trabalho preferimos nos referenciar à etapa de análise de requisitos como *exploração dos requisitos*, incluindo não apenas os requisitos importantes para o banco de dados, mas para todo o projeto computacional. De modo semelhante, utilizamos *concepção do projeto conceitual*, ao invés de modelagem de dados conceitual.

O modelo de dados conceitual alcançou um sucesso considerável como meio de interação entre o desenvolvedor e o especialista de domínio, principalmente quando utilizados diagramas ER ou UML para sua representação. Razões para tal aceitação são a maior facilidade de compreensão dos modelos representados por meio desses diagramas e a adoção de construções simples, que representam as abstrações de dados. O modelo de dados conceitual ajuda os desenvolvedores a capturarem com precisão os requisitos de dados reais, pois exige atenção aos detalhes semânticos dos relacionamentos e dos dados.

A utilização de diagramas, ER ou UML, baseando-se em abstrações, permite alcançar uma precisão adequada para conceber o projeto conceitual de bancos de dados. Porém, é possível obter uma alternativa que apresente maior agilidade para a concepção e, sobretudo, validação do projeto conceitual de bancos de dados. Apesar de facilitar a representação do projeto conceitual, como meio de interação entre o desenvolvedor e o especialista de domínio, os diagramas ER e UML são de difícil validação pelo especialista. Para confirmar se um determinado diagrama realmente representa todos os conceitos necessários do domínio de aplicação, o especialista de domínio necessitaria possuir profundos conhecimentos dos diagramas e das abstrações de dados.

Para superar essa dificuldade nossa abordagem apresenta como meio de interação entre o desenvolvedor e o especialista um protótipo manipulável e de fácil alteração, hipótese apresentada na Seção 1.1. Da mesma forma que os diagramas, esse protótipo representa um projeto conceitual utilizando abstrações de dados. Portanto, validar o projeto concebido passa a ser uma atribuição do especialista de domínio. Para isso, o protótipo explicita os comportamentos das abstrações de dados em uma interface orientada a objetos. O especialista precisa apenas dominar como interagir com a interface, um conhecimento muito mais acessível do que entender como prever comportamentos a partir de um diagrama.

Interagindo com o modelo junto ao desenvolvedor é possível indentificar se os comportamentos apresentados realmente representam o domínio de aplicação. Mudanças na representação devem ser de execução rápida e simples, refletindo rapidamente em mudanças no comportamento apresentado. Isso torna o pro-

cesso de validação e ajuste do projeto muito mais dinâmico, permitindo alcançar um excelente nível de validação, aumentando a precisão do projeto conceitual de bancos de dados criado.

A prática de envolver o especialista de domínio é uma característica de métodos ágeis de desenvolvimento. Para viabilizar essa abordagem, buscamos os princípios ágeis de desenvolvimento, tornando o projeto conceitual de bancos de dados mais fácil de compreender e, ao mesmo tempo, manipulável. Nossa opção foi utilizar o arcabouço Naked Objects, representando não apenas os conceitos do domínio de aplicação, mas também explicitando os comportamentos derivados das abstrações de dados escolhidas.

2.3 Desenvolvimento ágil

Durante a década de 90, muitas metodologias de desenvolvimento de *software* atraíram atenção [1] ao combinar antigas e novas idéias, apresentando alguns importantes pontos em comum. Em 2001, durante um *workshop* em Snowbird, Utah, EUA, o termo “*ágil*” foi escolhido para representar as metodologias que compartilham essas características. Os responsáveis pelo termo compuseram o *Manifesto for Agile Software Development* [2], destacando essas características comuns, no qual um dos trechos considerados mais importantes define os valores ágeis:

- **indivíduos e interações** em detrimento de processos e ferramentas (*individuals and interactions over processes and tools*);
- **software em funcionamento** em detrimento de documentação detalhada (*working software over comprehensive documentation*);
- **colaboração do cliente** em detrimento de negociação de contratos (*customer collaboration over contract negotiation*);
- **adaptação às mudanças** em detrimento de seguir um plano (*responding to change over following a plan*).

Completando-o com a frase:

(...)

*That is, while there is value in the items on the right, we value the items on the left more.*²

(...)

Considerando que enquanto os itens à direita possuem seu valor, os itens à esquerda são mais valorizados na visão ágil. Além disso, também foi elaborada uma declaração de princípios utilizados no desenvolvimento ágil. Os princípios abordados neste projeto são:

- priorizar a satisfação do cliente com entregas contínuas de *software* em funcionamento, começando as entregas o mais cedo possível;
- aceitar e incentivar mudanças nos requisitos, mesmo que tarde no desenvolvimento;
- entregas freqüentes de *software* em funcionamento, com pequenos intervalos de poucas semanas ou meses, preferindo escalas de tempo menores;
- os especialistas de negócio e desenvolvedores devem trabalhar juntos, diariamente, durante o projeto;
- criar projetos com indivíduos motivados, fornecendo a eles o ambiente e apoio necessários, confiando neles para finalizar o trabalho;
- a melhor forma de transferir e obter informações é por meio de conversas *cara-a-cara*;
- *software* em funcionamento é a principal medida de progresso;
- atenção contínua à excelência técnica e bom *design* aumentam a agilidade;
- simplicidade - a arte de maximizar a quantidade de trabalho não feito - é essencial.

Esses conceitos, valores e princípios forneceram inspiração para a abordagem deste projeto. Procuramos aproximar o máximo possível a abordagem desses valores, a fim de tornar o projeto conceitual de bancos de dados mais ágil. Para isso, começamos nossa abordagem utilizando a iniciativa Naked Objects, que traz agregada parte desses valores e princípios.

²Os termos originais foram transcritos para preservar a forma mais conhecida pela comunidade.

2.4 Naked Objects

A iniciativa Naked Objects [22, 23, 26] apresenta grande sinergia com os métodos ágeis de desenvolvimento de *software*, consistindo de um conjunto de idéias que, de acordo com [27], podem ser compreendidas como³:

(...) an architectural pattern whereby core business objects are exposed directly to the user instead of being masked behind the conventional constructs of a user interface (...)

O arcabouço Naked Objects [22], que foi projetado especificamente para atender a esse padrão arquitetural, permite definir objetos como classes em Java seguindo um conjunto pré-estabelecido de convenções de código, tornando possível criar automaticamente uma interface de usuário orientada a objetos. Esse arcabouço possui um ambiente que inclui um mecanismo de visualização capaz de criar, em tempo de execução, uma representação manipulável dos objetos de domínio.

A criação automática de uma interface gráfica que permite uma manipulação direta dos objetos representados é um dos aspectos centrais da abordagem Naked Objects. Dessa forma, os objetos representados são expostos diretamente para o usuário que pode utilizá-los livremente para solucionar os problemas necessários, ao contrário de interfaces procedurais restritivas que obrigam o usuário a seguir uma sequência rígida de passos pré-estabelecidos. Uma discussão mais abrangente é apresentada no Apêndice A.

O aspecto mais interessante desse arcabouço é sua utilização para a exploração dos requisitos de um sistema. Nessa exploração, os objetos podem ser identificados por meio de conversas entre o especialista de domínio e o desenvolvedor. O desenvolvedor então implementa um protótipo no arcabouço. Esse protótipo é apresentado ao especialista de domínio, que pode manipular os requisitos identificados. Um ciclo rápido é então estabelecido: novos requisitos são identificados, o protótipo é atualizado e pode novamente ser manipulado pelo especialista de domínio. Esse processo é repetido até que o projeto conceitual consiga capturar adequadamente os requisitos necessários ao sistema.

Esses ciclos rápidos e dinâmicos de validação aumentam a velocidade do processo de exploração de requisitos. A validação imediata permite aprimorar a precisão dos requisitos identificados e representados no protótipo. Para isso, o arcabouço permite construir, eficientemente, protótipos de sistemas de com-

³(...) um padrão arquitetural no qual objetos do domínio são expostos diretamente ao usuário ao invés de serem escondidos atrás de estruturas convencionais de interface de usuário (...)

putação apenas definindo os objetos do domínio de aplicação específico. Nesse protótipo todas as ações de usuários consistem em criar ou recuperar objetos, especificando seus atributos, estabelecendo associações entre eles, ou invocando métodos em um objeto (ou coleção de objetos).

2.4.1 Criando um Naked Object

O arcabouço representa o protótipo do sistema como uma interface orientada a objetos com as seguintes características:

- classes ou tipos de entidade são representados por meio de ícones a partir dos quais é possível criar novas instâncias;
- uma instância também é representada como um ícone ou como um formulário listando os atributos dessa instância e seus valores;
- valores dos atributos podem ser editados por meio dos formulários das instâncias;
- métodos podem ser invocados por meio de um *menu de contexto*⁴. Métodos cujo parâmetro seja uma instância de um outro objeto podem ser executados *arrastando* uma instância parâmetro e *soltando-a* sobre a instância alvo da execução.

Nossa abordagem de projeto visa identificar elementos do domínio especificando-os como um **Naked Object**. Para que uma classe em Java seja um **Naked Object** é necessário que ela implemente a interface *NakedObject*. A maneira mais usual de implementar essa interface é estender a classe *AbstractNakedObject*. Além disso, é necessário implementar o método *title*. Abaixo apresentamos um exemplo de **Naked Object**:

```
public class Book extends AbstractNakedObject
{
    private final TextString name = new TextString();

    public TextString getName()
```

⁴Um menu de contexto é um recurso de interfaces de usuário que permite acessar funções específicas por meio de um pequeno menu que surge ao se clicar com o botão secundário em um elemento da interface gráfica.

```
{  
    return name;  
}  
  
public Title title()  
{  
    return getName().title();  
}  
}
```

Nesse exemplo definimos uma classe denominada `Book`. Para torná-la um **Naked Object** estendemos a classe `AbstractNakedObject`. Além de estender a classe `AbstractNakedObject` é necessário seguir as convenções esperadas pelo arcabouço Naked Objects. Entre elas, definir os tipos dos atributos, ou variáveis membro da classe, como **Naked Objects**. Para isso, são disponibilizados **Naked Objects** que representam os tipos mais comuns em Java. No exemplo, o tipo **Naked Object** do atributo `name`, `TextString`, é correspondente ao tipo da linguagem Java `String`. Esses tipos são denominados *Naked Values*. Outra convenção do arcabouço é a criação de métodos acessores (*gets* e *sets*) para os atributos. Para *Naked Values*, é apenas necessário o método `get`, por isso, no exemplo, foi implementado apenas o método `getName`. O método `title` é especificado na interface *Naked Object* e, como não é fornecido pela classe `AbstractNakedObject`, deve ser sempre implementado. Sua função é fornecer um título para a instância de um determinado **Naked Object**, no exemplo, o título será o valor de seu atributo `name`.

Na Figura 2.1, podemos verificar a interface gráfica gerada para esse exemplo. Em (a), à esquerda, verificamos um ícone que representa os livros como um tipo de objeto do sistema, **Books**. Esse ícone permite listar as instâncias existentes, procurar por uma instância específica, ou criar novas instâncias do objeto, tudo isso por meio de um menu de contexto, em (b). Ao lado direito desse ícone, podemos ver a representação de uma instância de um **Book** como um formulário que permite a visualização e alteração dos dados do objeto. Abaixo uma outra instância é representada como um ícone.

Para mais detalhes sobre como especificar Naked Objects sugerimos a leitura de [25]. Outras informações sobre o arcabouço Naked Objects e os princípios da iniciativa são apresentados no Apêndice A.

Figura 2.1: Interface gerada pelo arcabouço Naked Objects

2.5 Conclusão

Para atingir a precisão necessária na concepção do projeto conceitual de bancos de dados, utilizamos como alicerce de nossa abordagem um conjunto de abstrações de dados formado pelas abstrações de classificação, relacionamento, generalização-especialização, composição e objeto-relacionamento.

Para alcançar maior agilidade na concepção do projeto conceitual de bancos de dados, buscamos os princípios ágeis de desenvolvimento, procurando tornar sua representação manipulável, facilitando a validação pelo especialista de domínio.

O ponto de partida para a abordagem foram os conceitos presentes na iniciativa Naked Objects, que apresentam grande sinergia com os métodos ágeis de desenvolvimento.

Esses aspectos nos levaram à criação de um arcabouço que permite descrever o projeto conceitual de bancos de dados utilizando anotações. O arcabouço então disponibiliza uma interface para manipulação e validação do projeto representado.

Capítulo 3

Anotações

Para alcançar um projeto conceitual de dados que seja preciso na representação dos conceitos do domínio de aplicação, utilizamos abstrações de dados como base deste projeto. Para representar as abstrações de dados de forma precisa e dinâmica em um ambiente ágil, como o arcabouço Naked Objects, optamos por utilizar um conjunto de *metadados* implementados como *anotações* da linguagem Java. Nesse sentido, esses metadados ou anotações, devem ser entendidos como informações estruturadas que resumem, enriquecem ou complementam os objetos. O aspecto importante a considerar é que informações adicionais a respeito dos elementos do ambiente devem ser indicadas, permitindo seu tratamento diferenciado para representar as abstrações de dados. Como a especificação para o ambiente Naked Objects é feita utilizando classes em Java, representamos as abstrações como classes em Java anotadas.

As abstrações de dados representam, normalmente, mais de um tipo de entidade e como esses tipos se associam e relacionam. De forma semelhante, algumas abstrações necessitam de classes com diferentes anotações para representar adequadamente o projeto conceitual. Cada abstração será discutida em detalhes mais adiante, o importante é ressaltar que uma abstração não é representada apenas pela anotação, mas pela aplicação de anotações a classes, que juntas representam a abstração de dados.

A próxima seção explora a evolução da abordagem, enfatizando os problemas encontrados e o surgimento da abordagem de anotações para representar as abstrações de dados, as seções seguintes descrevem, em detalhes, como utilizar anotações para representar cada uma das abstrações de dados citadas anteriormente.

A seção seguinte apresenta as anotações desenvolvidas. Essa seção é um resumo do Apêndice B que têm como primeiro objetivo definir um catálogo de como representar cada uma das abstrações de dados mais utilizadas. O formato desse catálogo foi inspirado no famoso catálogo de padrões [15]. Os conceitos envolvidos nas abstrações de dados e seus resultados como objetos do sistema são vastamente empregados no projeto conceitual de bancos de dados. Porém, variações ocorrem, principalmente, na nomenclatura utilizada, como citado anteriormente. Utilizamos uma nomenclatura própria que, muitas vezes, remete diretamente ao nome da abstração ou de seu principal resultado.

Como segundo objetivo desse apêndice, destacamos a representação das abstrações utilizando um conjunto de metadados aplicáveis aos objetos de dados. Esses metadados indicam o papel de cada objeto de dado envolvido em uma abstração. Apresentamos os metadados como anotações que possibilitam sua aplicação direta sobre classes de linguagens orientadas a objetos. Por outro lado, a abordagem de metadados para indicar o papel de um objeto de dados em uma abstração poderia ser aplicada a qualquer modelo que possibilite a representação de objetos de dados e a interpretação de metadados.

Como último objetivo apresentamos exemplos para a ferramenta implementada, contendo o código para utilização com a ferramenta, a interface de validação gerada e o script SQL (*Structured Query Language*) gerado.

3.1 Implementação inicial da abstração de relacionamento

Muitas das afirmações desta seção são baseadas nas dificuldades encontradas para se representar uma abstração de dados diretamente no arcabouço Naked Objects. Essa experiência permite ilustrar a necessidade de representar os conceitos envolvidos nas abstrações de dados de maneira mais simples e precisa, o que possibilitaria sua aplicação direta em diversos modelos de representação. A utilização das abstrações de dados no arcabouço Naked Objects permite unir o projeto conceitual de bancos de dados à exploração de requisitos proposta pelo arcabouço. Dessa forma, apresentamos os limites da representação Naked Object e como eles foram superados.

O arcabouço Naked Objects permite representar restrições sobre as suas operações. Como apresentado na Seção 2.4, os tipos dos atributos de um **Naked Object** devem ser um tipo **Naked Object**. Para os atributos mais simples existem os *Naked Values*, porém, quando um dos atributos é um outro

objeto criado pelo desenvolvedor, a atribuição de um objeto como atributo de um outro caracteriza uma associação entre esses dois objetos. O arcabouço disponibiliza convenções de métodos para restringir essas associações. Além disso, múltiplas associações podem ser especificadas representando atributos como coleções de **Naked Objects**.

A primeira dificuldade encontrada foi representar restrições de inserção nas coleções de objetos, comportamento que precisou ser incluído. A segunda, e mais importante dificuldade, foi a complexidade de representação das abstrações de dados apenas com os construtores disponibilizados no arcabouço original **Naked Objects**. Para superar esse obstáculo utilizamos anotações.

Para ilustrar o processo de extensão que resultou na criação das anotações e da ferramenta desenvolvida, será apresentado em detalhes um exemplo de implementação da abstração de relacionamento com cardinalidade *um para muitos* (1:N).

Esse caso apresenta um domínio simples que possui duas classes fundamentais: **Pessoa** e **Conta**. **Pessoa** possui um atributo **nome**. **Conta** possui atributos **numero** e **tipo**. Assumiremos, por simplicidade, que todos os atributos citados podem possuir valores compostos de quaisquer quantidade de caracteres (*String*). Nesse domínio uma **Pessoa** pode possuir diversas **Contas** e uma **Conta** pertence a uma única **Pessoa**. Dessa forma, a abstração de relacionamento com cardinalidade *um para muitos* deve ser aplicada.

A definição desse domínio deve garantir que: se uma **Conta** estiver associada a uma determinada **Pessoa**, essa **Conta** não pode ser associada a uma outra **Pessoa**. Essa restrição simples não pode ser implementada seguindo apenas as convenções usuais do arcabouço **Naked Objects**. Essa limitação do arcabouço, constatada na versão utilizada (versão 1.2.2), nos levou à primeira extensão necessária para permitir a representação adequada da abstração por meio desse arcabouço.

3.1.1 Implementação da abstração de relacionamento no arcabouço estendido

Para superar a dificuldade de restrições para as coleções, analisamos profundamente o funcionamento interno do arcabouço e criamos novas coleções que permitem criar restrições semelhantes às disponibilizadas para atributos simples. A forma como foi implementada essa extensão não é apresentada em detalhes, mas exigiu um grande volume de trabalho, sobretudo para compreender o funcionamento do arcabouço. Essa implementação foi de fundamental importância para permitir o restante deste trabalho.

Essa extensão permite a especificação do caso em estudo seguindo os moldes das demais convenções de restrição do arcabouço. Para detalhes da codificação segundo as convenções do arcabouço, sugerimos a leitura de [25]. O código descrito e apresentado a seguir ilustra como implementar o exemplo em estudo na versão estendida do arcabouço.

Para incluir o relacionamento em questão e obter o comportamento esperado é necessário criar na classe `Conta` a variável membro `pessoa`, seguindo da criação dos respectivos métodos acessores `getPessoa()` e `setPessoa()`, assim como os métodos de associação esperados pelo arcabouço `associatePessoa()` e `dissociatePessoa()`.

Na classe `Pessoa` deve-se criar a variável membro `contas`. Após isso deve-se criar o método acessor `getContas()`, os métodos de associação `associateContas()` e `dissociateContas()` e, por fim, um método rebuscado de controle de associação `aboutAssociateContas()`.

O arcabouço original não permite criar métodos de controle (como `aboutAssociateContas`) para os métodos de associação (`associateContas`). A extensão implementada permite a definição de métodos de controle para associações seguindo convenções semelhantes às utilizadas para se definir métodos de controle no arcabouço.

```
public class Conta extends AbstractNakedObject {

    private final TextString numero = new TextString();
    private final TextString tipo = new TextString();
    private Pessoa pessoa;
    public TextString getNumero() {
        return numero;
    }
    public TextString getTipo() {
        return tipo;
    }
    public Pessoa getPessoa() {
        resolve(pessoa);
        return pessoa;
    }
}
```

```
}

public void setPessoa(Pessoa pessoa) {
    this.pessoa = pessoa;
    objectChanged();
}

public void associatePessoa(Pessoa pessoa) {
    pessoa.associateContas(this);
}

public void dissociatePessoa(Pessoa pessoa) {
    pessoa.dissociateContas(this);
}

public Title title() {
    return getNumero().title().append(" - ", tipo);
}

...

public class Pessoa extends AbstractNakedObject {

    private final TextString nome = new TextString();
    private final ExtendedInternalCollection contas =
        new ExtendedInternalCollection(Conta.class, this);
    public TextString getNome() {
        return nome;
    }

    public ExtendedInternalCollection getContas() {
        return contas;
    }

    public About aboutAssociateContas(Conta conta) {
        if(conta != null && conta.getPessoa() != null){
            return new AbstractAbout(null, null, Allow.DEFAULT,
                new Veto("Can't be associated more than one time " + conta.title()))
```

```
        ) { };
    else {
        return new AbstractAbout(null, null, Allow.DEFAULT,
                                   new Allow()
        ) { };
    }
}

public void associateContas(Conta conta) {
    getContas().add(conta);
    conta.setPessoa(this);
}

public void dissociateContas(Conta conta) {
    getContas().remove(conta);
    conta.setPessoa(null);
}

public Title title() {
    return getNome().title();
}
}
```

3.1.2 Implementação da abstração de relacionamento utilizando anotações

Alterar as características do relacionamento no arcabouço não é uma tarefa simples. Uma possível alteração poderia ser, por exemplo, trocar a restrição de cardinalidade. Para implementar essa simples alteração, adequando o comportamento esperado, diversos métodos deveriam ser alterados. Além disso, poderia ser necessário criar novos métodos ou mesmo excluir alguns dos existentes. Dessa forma, consideramos que essa alteração não apresenta a agilidade necessária.

Buscamos obter uma representação mais simples dessa mesma semântica de relacionamento, sobretudo da perspectiva do desenvolvedor. Simultaneamente, esperamos que essa representação seja também muito mais dinâmica, ampliando a sua capacidade de adaptação a alterações. Estabelecemos, então, como

implementar a abstração de relacionamento com cardinalidade *um para muitos*. Abaixo apresentamos a implementação para o caso discutido, seguida da sua explicação:

@Entity

```
public class Conta extends AbstractNakedObject {

    private final TextString numero = new TextString();
    private final TextString tipo = new TextString();
    private Pessoa pessoa;

    public Pessoa getPessoa() {
        resolve(this.pessoa);
        return this.pessoa;
    }

    public void setPessoa(Pessoa pessoa) {
        this.pessoa = pessoa;
        this.objectChanged();
    }

    public TextString getNumero() {
        return numero;
    }

    public TextString getTipo() {
        return tipo;
    }

    public Title title() {
        return getNumero().title().append(" - ", tipo);
    }
}
```

@Entity

```
public class Pessoa extends AbstractNakedObject {
```

```
private final TextString nome = new TextString();
@RelationshipAssociation(
    cardinality = RelationshipAssociation.Cardinality.OneToMany,
    relatedWith = Conta.class,
    fieldRelatedName = "pessoa"
) protected final ExtendedInternalCollection contas =
    new ExtendedInternalCollection(Conta.class, this);
public TextString getNome() {
    return nome;
}
public Title title() {
    return getNome().title();
}
}
```

Para utilizar a abordagem de anotações, na classe **Conta**, é necessário incluir a anotação **@Entity** que identifica essa classe como uma entidade do domínio de aplicação que pertence ao projeto conceitual de bancos de dados, e a variável membro que representará o relacionamento para essa classe: **pessoa**, com seus respectivos métodos acessores, seguindo as convenções do arcabouço. Todo o conteúdo restante da classe **Conta** se refere a seus atributos e comportamentos específicos, independentemente do relacionamento.

Na classe **Pessoa**, além da anotação **@Entity**, é necessário definir o relacionamento. Para isso, a variável membro **contas** deve ser criada e uma anotação vinculada a essa variável.

A anotação em questão é do tipo **RelationshipAssociation**, que define uma abstração de relacionamento e espera três parâmetros: a restrição de cardinalidade (**cardinality**), no caso *um para muitos* (**OneToMany**); a outra classe pertencente ao relacionamento (**relatedWith**) e o nome da variável membro, **fieldRelatedName**, que representa o relacionamento na classe indicada no parâmetro anterior. A anotação é feita no seguinte trecho:

```
@RelationshipAssociation(
```

```
    cardinality = RelationshipAssociation.Cardinality.OneToMany,  
    relatedWith = Conta.class,  
    fieldRelatedName = "pessoa"  
    ) protected final ExtendedInternalCollection contas =  
        new ExtendedInternalCollection(Conta.class, this);
```

Porém, essa mesma anotação pode ser especificada de forma ainda mais concisa:

```
@RelationshipAssociation(Cardinality.OneToMany, Conta.class, "pessoa")  
    protected final ExtendedInternalCollection contas =  
        new ExtendedInternalCollection(Conta.class, this);
```

A simplicidade em definir o relacionamento pode ser mensurada com relação à redução da quantidade de código necessário. Inicialmente, eram necessárias diversas definições de métodos nas duas classes. Com o novo padrão, basta uma simples utilização de uma anotação em uma das classes participantes, além das convenções para se definir atributos como variáveis-membros.

A redução no volume de código não implica, necessariamente, maior facilidade de utilização ou alteração. Porém, podemos perceber pelo exemplo que a utilização da anotação é clara e intuitiva para desenvolvedores. A facilidade de alteração do modelo pode ser observada no caso de se trocar a restrição de cardinalidade de *um para muitos* para *muitos para muitos*. Nesse padrão, bastaria alterar o valor da restrição de cardinalidade (parâmetro `cardinality` da anotação) de `OneToMany` para `ManyToMany` e ajustar a variável membro relacionada para armazenar múltiplos valores seguindo as convenções do arcabouço.

3.2 Anotações para as abstrações de dados

3.2.1 Classificação - Entidade

De forma genérica, uma implementação de anotação capaz de identificar a classe como um tipo de entidade do domínio de aplicação, uma Entidade, é consideravelmente simples. Anotações, como esta, que não possuem parâmetros, são denominadas *marcações*.

A utilização da anotação Entidade para identificar a classe `Book`, representada no arcabouço `Naked Objects`, como uma entidade do domínio de aplicação seria:

```
@Entity
public class Book extends AbstractNakedObject {
    ...
    private final WholeNumber edition = new WholeNumber();
    ...
    public WholeNumber getEdition() {
        return edition;
    }
    ...
}
```

Figura 3.1: Interface gerada para a abstração de classificação

A representação da abstração utilizando a anotação é muito simples, basta adicionar a anotação

`@Entity` à classe que se deseja identificar como uma entidade do domínio (no exemplo, `Book`). Com essa implementação a ferramenta é capaz de gerar automaticamente a interface gráfica apresentada na Figura 3.1, na qual é possível identificar, à esquerda, um ícone que representa a entidade `Book`, por meio do qual é possível criar novas instâncias da entidade e listar (ou procurar) instâncias existentes. Ao centro, uma entidade é visualizada como um formulário, o que permite editar seus valores de atributos. À direita, a mesma instância é representada na interface apenas como um ícone.

Para complementar a implementação da classe `Book` seguindo as convenções do arcabouço e obter a interface da Figura 3.1, seria necessário incluir um método que retornasse o título da instância. Para esse exemplo foi codificado um método que simplesmente retorna o título *Mil e uma noites* para todas as instâncias de `Book`.

3.2.2 Generalização-especialização

Diversas áreas da computação utilizam hierarquias para representar estruturas de dados obtidas por meio da abstração de generalização e especialização. Para a comunidade de banco de dados, sua utilização evita replicação desnecessária de dados e permite centralizar a representação de conjuntos comuns de atributos.

Consideremos que no nosso exemplo, domínio da biblioteca, seja necessário representar os estudantes que se cadastraram para retirar livros. Para isso, cria-se a entidade Estudante (**Student**) utilizando a abstração de classificação. Por outro lado, é necessário representar os empregados da biblioteca. Cria-se, então, uma entidade Empregado (**Employee**), novamente utilizando a abstração de classificação.

Durante a criação da entidade Empregado, percebe-se que um grande conjunto de atributos, referentes a dados pessoais como data de nascimento, número de documentos, nome, endereço e outros, estão presentes em ambas entidades. Além disso, alguns empregados da biblioteca são também estudantes, e esses dados estariam duplicados nos dois cadastros. Torna-se evidente a possibilidade de generalizar essas entidades, criando uma nova entidade que represente esse conjunto de atributos comuns, evitando a replicação dos dados. Seria então criada uma nova entidade chamada, por exemplo, de Pessoa (**Person**) estabelecendo uma hierarquia entre as entidades, de acordo com a figura anterior.

Para representar a abstração de generalização-especialização são necessárias duas anotações: generalização e especialização. Ambas as anotações devem ser utilizadas em conjunto para representar a abstração.

A anotação de generalização deve ser aplicada à classe que desempenha o papel de tipo de entidade pai em uma relação de especialização.

```
annotation Generalization{
    enum Completeness{Total, Partial}
    enum Disjointness{Disjoint, Overlapping}
    Completeness completeness();
    Disjointness disjointness();
}
```

A anotação de generalização possui dois parâmetros: o primeiro, `completeness`, indica se a generalização é `Total` ou parcial (`Partial`); o segundo, `disjointness`, permite identificar a generalização como disjunta (`Disjoint`) ou sobreponível (`Overlapping`).

Duas diferentes anotações podem ser aplicadas a um tipo de entidade filha da relação de generalização: a anotação de *Especialização* ou a anotação de *Especialização definida por Predicado*.

```
annotation Specialization{
    class specializes();
}
```

A anotação de Especialização possui um único parâmetro: (`specializes`), que indica de qual tipo de entidade pai a classe anotada é filha. A anotação de Especialização definida por Predicado é uma extensão da anotação de Especialização:

```
annotation PredicatedSpecialization {
    class specializes();
    string fieldName();
    Operator operator();
    string value();
}
```

```
enum Operator {
    equalTo, notEqualTo, lessThan,
    lessThanEqualTo, greaterThan,
    greaterThanEqualTo
}
```

```
}
```

A anotação possui quatro parâmetros: o primeiro (**specializes**) indica de qual tipo de entidade pai a classe anotada é filha; o segundo é o nome do atributo do tipo de entidade pai que define o predicado (**fieldName**); o terceiro é o operador condicional do predicado (**operator**); e o quarto o valor de comparação do predicado (**value**).

Para representar o exemplo anteriormente citado com as classes **Student**, **Employee** e **Person**, podemos utilizar as três anotações do seguinte modo:

```
@Generalization(  
    completeness = Completeness.Partial,  
    disjointness = Disjointness.Overlapping  
)  
Class Person {  
    WholeNumber age;  
    ...  
}  
...  
@Specialization(  
    specializes = Person.class  
)  
@Entity  
Class Student{  
    ...  
}  
...  
@PredicatedSpecialization(  
    specializes = Person.class,  
    fieldName = "age",  
    operator = Operator.greaterThanEqualTo,  
    value = "18"
```

```

)
@Entity
Class Employee{
    ...
}

```

O comportamento da interface será dependente dos parâmetros da generalização. A interface gerada pela ferramenta é apresentada na Figura 3.2, sendo a generalização sobreponível e parcial. O fato da generalização ser parcial permite criar uma instância do tipo de entidade **Person** sem que ela seja uma instância de algum dos dois tipos de entidades filhos, por exemplo, na Figura B.2a, a instância **Jack** é apenas uma instância do tipo de entidade **Person**. Caso a generalização fosse total, para se obter uma instância da entidade **Person** seria necessário criar uma instância de **Employee** ou **Student**. Por outro lado, a generalização é sobreponível, o que permite uma instância de **Person** ser especializada como mais de um tipo de entidade filho. Ou seja, uma mesma instância, por exemplo **Jack**, pode ser especializada como uma instância de **Student** e de **Employee** simultaneamente.

Figura 3.2: Ambiente Naked Objects

Na Figura 3.2b a instância **Jack** já foi especializada como **Student** e ainda pode ser especializada

como **Employee**. Caso a generalização fosse disjunta, apenas uma única especialização seria permitida a uma mesma instância do tipo de entidade pai. Por fim, percebemos a diferença entre a utilização da anotação **Specialization** na classe **Student** e da anotação **PredicatedSpecialization** na classe **Employee**. Como não existe um predicado associado à classe **Student** sempre é possível especializar uma instância de **Person** (o menu de contexto estará habilitado, como *New Student...* na Figura 3.2a), desde que ainda não seja especializada como **Student** (o menu será desabilitado, como *New Student...* na Figura 3.2b). Por outro lado, para especializar uma instância de **Person** como um **Employee** o predicado deve ser atendido, no caso a idade (**age**) deve ser maior ou igual a 18 (estando desabilitado na Figura 3.2a, e habilitado na Figura 3.2b).

O código SQL gerado é uma possibilidade de representação da abstração de generalização-especialização em um banco de dados relacional e está apresentado integralmente na Seção C.1.

3.2.3 Relacionamento

Para representar a associação de relacionamento é necessário criar apenas mais uma anotação, que deve ser aplicada a um atributo de uma das duas entidades participantes da associação de relacionamento. Esse atributo deve ser do tipo da outra entidade participante. Da mesma forma, a outra entidade também deve possuir um atributo do tipo da entidade que teve o atributo anotado. Desse modo, é necessário apenas anotar um único atributo em apenas uma das duas entidades.

```
annotation RelationshipAssociation {  
    Cardinality cardinality();  
    class relatedWith();  
    string fieldRelatedName();  
}  
  
enum Cardinality{  
    OneToOne, OneToMany, ManyToOne, ManyToMany  
}
```

Essa anotação possui três parâmetros: o primeiro é a restrição de cardinalidade (`cardinality`) que pode assumir os valores um para um (`OneToOne`), um para muitos (`OneToMany`), muitos para um (`ManyToOne`) e muitos para muitos (`ManyToMany`); o segundo é a outra classe pertencente à associação (`relatedWith`); e o terceiro é o nome do atributo da outra classe da associação (`fieldRelatedName`).

3.2.4 Composição

Representar uma associação de composição é semelhante a representar uma associação de relacionamento. Da mesma forma, a única nova anotação deve ser aplicada a um atributo de uma das duas entidades participantes da associação. Esse atributo deve ser do tipo da outra entidade participante. A outra entidade também deve possuir um atributo do tipo da entidade que teve o atributo anotado, porém, é necessário apenas anotar um único atributo em apenas uma das duas entidades.

```
annotation CompositeAssociation {  
    enum CompositeType {Logical, Physical}  
    Cardinality cardinality();  
    class relatedWith();  
    CompositeType compositeType();  
    string fieldRelatedName();  
}  
  
enum Cardinality{  
    OneToOne, OneToMany, ManyToOne  
}
```

Essa anotação possui quatro parâmetros: o primeiro é a restrição de cardinalidade (`cardinality`) que pode assumir os valores um para um(`OneToOne`), um para muitos(`OneToMany`) e muitos para um (`ManyToOne`); o segundo é a outra classe pertencente à associação(`relatedWith`); o terceiro indica o tipo de composição (`compositeType`) que essa anotação representa, uma composição lógica(`Logical`) ou uma composição física(`Physical`); e o quarto é o nome do atributo da outra classe que faz parte da associação (`fieldRelatedName`).

É interessante ressaltar que as cardinalidades possíveis possuem sempre um lado com multiplicidade *um*, isso ocorre uma vez que a entidade composta e as suas entidades componentes estão fortemente vinculadas, não podendo assim uma mesma instância de uma entidade ser componente de mais de uma instância de uma entidade composta.

3.2.5 Objeto-Relacionamento

Quando uma associação possui atributos próprios, muitas abordagens recomendam que esses atributos sejam colocados em um dos dois tipos de entidades que participam da associação. Porém, essa não é uma solução adequada, pois ela desvincula o dado do local a que realmente pertence. Além disso, algumas associações se associam a outras entidades diretamente. Essas associações apresentam características de entidades, atributos e associações, desempenhando um papel de maior destaque no seu domínio de aplicação, do que as demais associações.

Consideremos o domínio de um consultório dentário. Um projeto inicial poderia identificar diretamente as entidades paciente (*Patient*) e dentista (*Dentist*). Poderíamos criar uma associação de relacionamento entre essas entidades, vinculando um dentista a um paciente. Esse vínculo poderia ser denominado, por exemplo, tratamento *Treatment*. Esse projeto simples poderia ser suficiente para representar algum domínio de aplicação, porém, nesse exemplo, um paciente pode ser atendido por mais de um dentista em um mesmo tratamento. Além disso, é necessário que, em cada vez que ocorra um atendimento, seja registrada a data desse evento. Ajustando o projeto inicial para atender a esses requisitos, percebemos que a associação entre um dentista e um paciente é para o atendimento, e não para todo o tratamento. Podemos então denominar o atendimento de consulta (*Appointment*). Mas isso acarretaria que a associação denominada de consulta possuísse o atributo data e que um tratamento fosse uma composição de consultas. Dessa forma, uma consulta apresenta simultaneamente as características e comportamentos de uma associação e de uma entidade.

A abstração de Objeto-Relacionamento possui uma única anotação que deve ser aplicada como a anotação da abstração de relacionamento, sobre um atributo de uma das duas Entidades Associadas. Porém, diferentemente do que ocorre na abstração de relacionamento, esse atributo deve ser do tipo da entidade do Objeto-Relacionamento. Além disso, essa entidade deve possuir atributos de associação do tipo das duas Entidades Associadas. A Entidade Associada que não teve um atributo anotado também deve possuir um atributo do tipo da entidade do Objeto-Relacionamento.

```
annotation RelationshipObject {
    Cardinality cardinality();
}
```

```

    class relatedWith();
    string fieldRelatedName();
    class compositeClass();
    string compositeFieldName();
    string compositeFieldRelatedName();
}
enum Cardinality{
    OneToOne, OneToMany, ManyToOne, ManyToMany
}

```

Essa anotação possui seis parâmetros: o primeiro é a restrição de cardinalidade (**cardinality**) que pode assumir os valores um para um (**OneToOne**), um para muitos (**OneToMany**), muitos para um (**ManyToOne**) e muitos para muitos (**ManyToMany**); o segundo é a outra Entidade Associada pertencente ao relacionamento(**relatedWith**); o terceiro é o nome do atributo do objeto-relacionamento que referencia a Entidade Associada anotada (**fieldRelatedName**); o quarto indica a classe que implementa o objeto-relacionamento (**compositeClass**) respectivo dessa associação; o quinto e o sexto são os nomes dos atributos responsáveis por representar a associação do objeto-relacionamento como a outra Entidade Associada (**compositeFieldName** na Entidade Associada e **compositeFieldRelatedName**) no Objeto-Relacionamento.

A interface gerada automaticamente está apresentada nas Figuras 3.3(a) e 3.3(b). Na 3.3(a) podemos observar, à esquerda, os ícones que representam os tipos de entidade: **Dentists**, **Patients** e **Appointments**. Logo abaixo, podemos identificar uma instância do tipo de entidade **Patient**, **Bob**, representada como um ícone. À direita, uma outra instância de **Patient**, **John**, é representada por meio de um formulário, assim como a instância de **Dentist**, **Dr.Brown**, acima.

Quando arrastamos a instância **John** o cursor torna-se um ícone. Quando posicionado sobre a instância **Dr.Brown**, a borda do formulário que representa essa instância torna-se verde, indicando a possibilidade de se soltar a instância arrastada para executar uma ação, nesse caso, a criação de um relacionamento que será representado por uma instância de **Appointment**. Na Figura 3.3(b), vemos o resultado da ação, criando-se uma instância de **Appointment** e o preenchimento automático dos campos responsáveis pelo relacionamento. Apesar de existir uma representação do tipo de entidade, **Appointments**, percebemos que a opção de criação de uma instância está desabilitada. A única maneira de se criar uma instância

Figura 3.3: Ambiente Naked Objects

do objeto-relacionamento é por meio da ação de arrastar e soltar descrita anteriormente. Dessa forma, o padrão representa na interface gráfica as restrições contidas no conceito da abstração a que se refere.

O código SQL é apresentado na seção C.2.

3.3 Conclusão

Apresentamos neste capítulo a representação utilizando anotações das abstrações de classificação, generalização-especialização, relacionamento, composição e objeto-relacionamento.

O principal limite para a utilização da abstração de classificação é que não é possível especializar diretamente uma entidade anotada apenas com a anotação Entidade. Para especializar uma entidade, deve ser utilizada a anotação de generalização (abstração de generalização-especialização 3.2.2).

A anotação de generalização deve incluir todos os comportamentos representados pela anotação entidade, dessa forma, definindo uma entidade do domínio que pode ser especializada. Por outro lado, as anotações de especialização apenas podem ser aplicadas a entidades devidamente definidas ou pela

anotação de entidade, ou pela anotação de generalização. Dessa forma, é possível criar diversos níveis de hierarquia, bastando generalizar uma entidade que, por sua vez, é uma especialização de outra.

A abstração de relacionamento é a mais simples abstração que estabelece associações entre duas entidades, sendo utilizada para representar a maioria das associações. Porém, existem alguns tipos de associações que devem ser representados por outras abstrações: quando a associação entre duas entidades é caracterizada como uma relação de hierarquia, a abstração de generalização-especialização deve ser utilizada; quando a associação apresenta uma relação de todo e parte, com uma entidade sendo composta pela outra, a abstração de composição é a mais indicada; quando a associação entre duas entidades possui um destaque relevante, a ponto de ser representada como uma entidade do domínio, com atributos e/ou associações próprias, a abstração que deve ser utilizada é a abstração de objeto-relacionamento.

As semelhanças entre a abstração de composição e a abstração de relacionamento são muitas. Na verdade, o comportamento de uma composição lógica é idêntico ao comportamento de um relacionamento com a mesma cardinalidade, porém, a composição lógica não pode assumir a cardinalidade *muitos para muitos*.

A abstração de objeto-relacionamento deve ser utilizada quando a associação entre duas entidades possui um destaque relevante, a ponto de ser representada como uma entidade do domínio, com atributos e associações próprias.

Capítulo 4

Ferramenta

Este capítulo apresenta a ferramenta desenvolvida. São descritos os diversos componentes e suas respectivas funções. Ao contrário do restante do texto, este capítulo se aprofunda tecnicamente na implementação concebida. Por outro lado, procuramos resumir o máximo possível as descrições para facilitar o acompanhamento. Para detalhes da implementação desenvolvida, disponibilizamos o código fonte completo no endereço <http://www.ime.usp.br/~jef/mbroinizi>.

4.1 Introdução

A ferramenta desenvolvida neste trabalho permite especificar o projeto conceitual de um banco de dados por meio de classes em Java anotadas que representam as abstrações de dados. Esse processo é ilustrado na Figura 4.1. Após a compilação, essas classes são analisadas pela ferramenta, por meio de introspecção, uma propriedade de sistemas orientados a objetos que qualifica a existência de mecanismos para descobrir, em tempo de execução, informações estruturais sobre um programa e seus objetos. A ferramenta cria e introduz nas classes compiladas o *Java bytecode* representando o comportamento da abstração identificada pela anotação, utilizando para isso a biblioteca Javassist [18] de instrumentação de código, uma técnica que permite a modificação do código e estrutura de um programa, após este ter sido compilado. Esse comportamento é definido por um conjunto de métodos e atributos que seguem a convenção de implementação de restrições do arcabouço Naked Objects. O arcabouço estendido Naked Objects cria, automaticamente e em tempo de execução, uma interface gráfica que representa o projeto

conceitual de bancos de dados descrito pelas classes instrumentadas. Essa interface permite que o especialista de domínio manipule e valide os conceitos do projeto conceitual de bancos de dados de forma ágil.

Figura 4.1: Processo

Podemos dividir a solução em seis componentes:

- anotações para representação das abstrações de dados;
- extensões do arcabouço Naked Objects;
- núcleo da ferramenta;
- gerador de código para Naked Objects;
- gerador de código SQL;
- mapa de tipos SQL para Naked Objects.

As anotações foram detalhadas no capítulo anterior. Apesar da abordagem ser baseada na aplicação das anotações em classes em Java para o arcabouço Naked Objects, seria possível utilizar o mesmo

conjunto de anotações em qualquer classe em Java, ou até mesmo, adaptar esse conjunto para outras linguagens.

Uma discussão das extensões do arcabouço Naked Objects foi apresentada na Seção 3.1. Foi necessário apenas acrescentar um novo tipo de coleção de Naked Objects que permitisse definir métodos de controle para incluir ou remover um Naked Object da coleção. Uma nova coleção foi criada, assim como algumas classes auxiliares. A classe `ClassHelper` precisou ser alterada para incluir a utilização da nova coleção em todo o arcabouço. Os outros componentes serão abordados nas próximas seções.

4.2 Núcleo da ferramenta

Esse componente representa, efetivamente, a ferramenta para o tratamento de anotações. Dessa forma, tem a responsabilidade de controlar a execução de toda a ferramenta, identificar as anotações e disparar os componentes que construirão os produtos esperados, como as classes Naked Objects ou o código SQL.

4.2.1 Padrão *Observer*

Antes de discutir a estrutura do núcleo, apresentamos uma versão do padrão *Observer* [15] que foi utilizada para determinar as dependências entre os objetos da ferramenta:

Figura 4.2: Padrão *Observer*

A utilização desse padrão permite definir dependências entre objetos de tal forma que, quando o estado de um objeto se modifica, todos seus objetos dependentes são notificados e atualizados automaticamente. Nessa implementação, sempre que um objeto **Subject** identifica a ocorrência de algum evento, ele atualiza seu estado, definindo uma **Notification** relativa ao evento identificado, e em seguida notifica a todos

os objetos **Observers** que foram registrados. Isso é muito utilizado na ferramenta. Por exemplo, quando o **Parser** de classes deve analisar as classes e notificar a ferramenta ao encontrar uma anotação. Nesse momento, a ferramenta solicita a **Notification** ao **Parser** para obter as informações sobre a anotação encontrada e realizar o tratamento necessário.

4.2.2 Diagrama de classes

Figura 4.3: O núcleo da ferramenta

Na Figura 4.3 identificamos a classe responsável por controlar toda a execução: **Tool**. A ferramenta possui em dois objetos principais: um interpretador de classes em Java, **AbstractParser**, responsável por identificar as anotações; e um **AbstractHandler**, responsável por disparar o tratamento das anotações identificadas. Podemos perceber no diagrama 4.3 que a dependência entre a classe **Tool** e o **AbstractParser** utiliza o padrão *Observer*, descrito na Seção 4.2.1.

A função do **AbstractParser** é analisar o conjunto de classes indicado pela ferramenta para identificar

as anotações. Quando isto ocorre, o `AbstractParser` notifica a ferramenta, que, por sua vez, obtém as informações sobre a anotação encontrada.

Essas informações são transmitidas ao `AbstractHandler` que possui um conjunto de classes registradas para tratar cada uma das anotações. Essas classes são classes filhas de `DataAbstraction`, como `EntityAbstraction`. A função dessas classes é estabelecer uma forma de representação interna da abstração identificada pela anotação. Assim, quando a ferramenta passa as informações sobre uma anotação identificada, o `AbstractHandler` cria uma representação adequada para a abstração, contendo todos os dados necessários para seu tratamento. Além disso, o `AbstractHandler` é a parte da ferramenta que se comunica com os componentes geradores. Após uma nova representação ser criada e encapsulada dentro de uma `AnnotationNotification`, o `AbstractHandler` notifica todos os `AbstractBuilders`.

Os `AbstractBuilders` são as interfaces de comunicação entre a ferramenta e os componentes geradores, como os de código para Naked Objects e para banco de dados relacional em SQL. Para que um componente seja utilizado pela ferramenta, é necessário que ele implemente duas interfaces: `AbstractBuilderFactory` e `AbstractBuilder`. A primeira define métodos para instanciação do componente, a segunda define a interface utilizada pelo núcleo para executar o componente.

O pacote da ferramenta possui classes *Factory* [15] *default* que utilizam as implementações já disponíveis para a criação de instâncias da ferramenta e de seus principais objetos internos, como a classe `AnnotatedClassParser` que implementa a interface `AbstractParser`. Essa abordagem é recorrente na maior parte da ferramenta, podendo ser identificada pela utilização de interfaces com o prefixo *Abstract*.

4.2.3 Associação de componentes geradores à ferramenta

O núcleo, assim como o conjunto de anotações, é independente do arcabouço Naked Objects. Sua estrutura utiliza sempre interfaces, o que facilita sua modificação e extensão. A configuração dos componentes responsáveis pela construção dos produtos é feita por um arquivo de propriedades.

Para que novas classes Naked Objects sejam produzidas pela execução da ferramenta, deve-se incluir um parâmetro no arquivo de propriedades definindo a classe responsável pela instanciação do componente gerador de código para Naked Objects. Para construir código SQL, inclui-se um parâmetro definindo a classe responsável pela instanciação do componente gerador de código SQL. Um exemplo de arquivo de configuração para a ferramenta seria:

```
tool_factory=br.usp.ime.tools.semantic.DefaultToolFactory

factory=br.usp.ime.tools.semantic.code.DefaultCodeBuilderFactory
factory=br.usp.ime.tools.semantic.database.relational.DefaultDatabaseBuilderFactory
...
```

Nesse exemplo, a ferramenta é configurada para utilizar a `ToolFactory` default e são definidas duas fábricas de componentes, a primeira para criar classes para Naked Objects e a segunda para criar códigos para banco de dados relacional em SQL.

Essa forma de configuração permite que ou apenas um produto seja obtido, ou ambos, ou mesmo nenhum. Além disso, seria simples configurar novos componentes que construíssem outros tipos de produtos como, por exemplo, um banco de dados orientado a objetos.

4.3 Gerador de código para Naked Objects

A função desse componente é criar e introduzir o comportamento das abstrações representadas pelas anotações como *Java bytecode* na classe anotada. Para isso, esse componente deve ser configurado para receber notificações da ferramenta contendo a representação lógica da anotação encontrada, como explicado em 4.2.3. Com base nessa representação, o componente utiliza técnicas de reflexão ¹, como introspecção e introdução, para criar um novo arquivo de *Java bytecode* que representa a classe original acrescida dos comportamentos da abstração de dados identificada.

A estrutura deste componente é mais específica. As principais classes são os `CodeHandlers`. Existe uma extensão de `CodeHandler` para cada anotação. Os `CodeHandlers` definem como uma determinada abstração é convertida em código para o arcabouço Naked Objects.

A classe que controla a execução do componente é o `CodeBuilder`, que implementa a interface `AbstractBuilder`, para comunicação com o núcleo da ferramenta. Essa classe também define características físicas do código, como o local para gravação do novo arquivo de *Java bytecode*.

¹Capacidade de observar ou até mesmo modificar a sua estrutura ou comportamento.

Esse componente foi escrito para criar código para o arcabouço Naked Objects. Um componente semelhante poderia ser escrito para classes em Java simples, outros arcabouços ou mesmo outras linguagens.

4.4 Gerador de código SQL

A função desse componente é criar um *script* SQL para criação de um banco de dados relacional para o projeto conceitual definido pelas abstrações de dados representadas como classes em Java para Naked Objects anotadas.

As regras de mapeamento do projeto conceitual para o código relacional SQL foram baseados na experiência de modelagem do Laboratório Avançado de Banco de Dados do Instituto de Matemática e Estatística da Universidade de São Paulo associada aos mapeamentos descritos em [13] e [10]. Não é escopo desse texto aprofundar a discussão sobre a melhor representação SQL para o projeto conceitual apresentado.

As principais classes desse componente, os `DatabaseHandlers`, são responsáveis por criar o código SQL. Existe uma dessas classes, filha de `DatabaseHandler`, para cada anotação.

A classe que controla a execução desse componente é a `RelationalDatabaseBuilder`, que define a comunicação com o núcleo e características físicas. Ao contrário do componente anterior, esse componente é incompleto, uma vez que depende de um componente auxiliar descrito a seguir.

4.5 Mapa de tipos SQL para Naked Objects

Esse componente auxiliar é necessário ao gerador de código SQL. Suas funções foram separadas pois é necessário especificar um mapeamento dos tipos de origem para os tipos de destino. Os tipos de origem são tipos Naked Object, uma vez que as anotações são aplicadas sobre classes em Java seguindo as convenções Naked Objects. Os de destino são tipos SQL. Dessa forma, esse componente fornece classes que definem os mapeamentos necessários, como `TypesMapNO`, e a classe *Factory* default para o componente descrito na seção anterior: `DefaultDatabaseBuilderFactory`.

4.6 Extensões

A estrutura da ferramenta, descrita nas seções anteriores, foi criada para facilitar modificações e extensões. O Apêndice D apresenta mais detalhes sobre a ferramenta. A Seção D.1 contém uma lista completa de todas as dependências e classes da ferramenta. Já a Seção D.2 apresenta um exemplo completo da criação de uma nova anotação.

4.7 Conclusão

Existem diversas ferramentas profissionais disponíveis auxiliam o projeto de bancos de dados. A maioria permite representar o projeto conceitual de um banco de dados por meio de diagramas ER ou UML. Analisamos comparativamente as capacidades de representação de duas ferramentas: o Sybase PowerDesigner 9.5 [29] e o ToadTM Data Modeler Freeware 2.24.0.7f, template 2.28 [33].

Na ferramenta PowerDesigner, estão disponíveis construtores específicos para as abstrações de classificação, generalização-especialização e relacionamento. O construtor de classificação possui uma capacidade de representação equivalente ao da ferramenta desenvolvida. O construtor de relacionamento também é equivalente, mas inclui o conceito de totalidade de participação. Já o construtor de generalização-especialização é muito mais simples que o da ferramenta desenvolvida, não especificando restrições de integralidade (total ou parcial) nem predicados. Não existe um construtor para a abstração de composição, mas existe um construtor para definições de associações que representa parte do comportamento da abstração de objeto-relacionamento.

A ferramenta Toad Data Modeler Freeware é mais restrita, definindo apenas construtores específicos para classificação e relacionamento. A ferramenta explora mais conceitos dentro da abstração de relacionamento, além da cardinalidade e totalidade de participação, incluindo classificações de relacionamentos com sutis diferenças, tais como relacionamentos identificados que permitem representar entidades dependentes. Essa ferramenta se concentra em disponibilizar apenas as abstrações mais simples, mas permite que sejam criados *templates*. Esses *templates* funcionam como abstrações definidas pelo usuário, compondo os construtores simples para criar um modelo de construtor mais complexo. A versão comercial acompanha um conjunto de pré-definido de *templates* que não foi avaliado.

De maneira geral, as ferramentas de projeto buscam a representação das abstrações mais simples, concentrando seus esforços no mapeamento para o projeto físico. Dessa forma, essas ferramentas podem criar um projeto físico que represente adequadamente o projeto conceitual. Criar um projeto físico para abstrações mais sofisticadas como especializações definidas por predicados é uma tarefa de maior complexidade que, muitas vezes, depende de parâmetros fora do escopo do projeto conceitual.

A nossa ferramenta evidencia que é possível criar uma abordagem para concepção de um projeto conceitual de banco de dados que vai além de diagramas e documentos de especificação. A ferramenta apresentada é, portanto, um protótipo que permite explorar as idéias definidas nesse projeto. Ela está disponível para estudos, utilização acadêmica e melhorias em <http://www.ime.usp.br/~jef/mbroinizi>.

Ao contrário de outras ferramentas, que priorizam o mapeamento para o projeto físico e não a validação do projeto conceitual ou o incremento de sua precisão e qualidade, essa ferramenta procura explorar diversas e importantes abstrações, permitindo criar e validar um projeto conceitual que represente adequadamente os requisitos do domínio.

Capítulo 5

Estudo de caso

Para ilustrar a utilização da ferramenta selecionamos um projeto do Laboratório Avançado de Banco de Dados (LABD) do Instituto de Matemática e Estatística da Universidade de São Paulo (IME-USP). A biblioteca do IME-USP necessitava de um novo sistema para controlar seu acervo e operações. Um grupo formado por especialistas em bancos de dados e especialistas em programação orientada a objetos e XP (eXtreme Programming [6]) se encarregou do projeto. Os especialistas do domínio eram os funcionários da biblioteca.

O projeto original não utilizou a ferramenta para projeto conceitual de bancos de dados desenvolvida. Revisitamos esse projeto após o desenvolvimento da ferramenta. Mais especificamente recriamos o projeto conceitual dos módulos Acervo e Pessoa. O principal objetivo foi verificar se a ferramenta permite representar um projeto conceitual grande e complexo de um domínio real.

5.1 Acervo e Pessoa

O acervo da biblioteca é responsável por representar os itens existentes, que podem ser emprestados ou consultados. O módulo *Pessoa* contém a representação das entidades cadastradas na biblioteca, desde empresas e outras bibliotecas até usuários. Esse foi o primeiro grande projeto conceitual que utilizou a ferramenta.

O módulo *Acervo* é o módulo com maior número de entidades, enquanto o módulo *Pessoa* foi conside-

rado um dos mais complexos do projeto original, principalmente, por conter muitas especializações. Essa ferramenta tornou simples especificar e validar essas especializações.

Esses dois módulos possuem um grande número de abstrações que se interligam e relacionam, fornecendo uma evidência objetiva da capacidade de representação da nossa abordagem.

Para exemplificar as características desse projeto, apresentamos um diagrama elaborado no projeto original do módulo Pessoa, na Figura 5.1. As telas iniciais dos protótipos que representam os módulos são apresentadas nas Figuras 5.2 e 5.4.

Figura 5.1: Diagrama original de um esboço inicial do módulo Pessoa

Uma comparação simples entre o projeto conceitual obtido com a ferramenta e o projeto conceitual original permitiu concluir que eles são muito semelhantes. Todas as principais abstrações podem ser identificadas em ambos os modelos. Não é do escopo deste trabalho comparar esses projetos conceituais ou identificar sua equivalência. O protótipo criado para esse estudo de caso pode ser obtido no endereço <http://www.ime.usp.br/~jef/mbroinizi>, assim como, diagramas do projeto original.

Para fim de ilustração, na Figura 5.3 pode-se identificar uma das abstrações recorrentes do projeto conceitual do módulo Pessoa: hierarquias de generalização-especialização. Nessa hierarquia, um

Figura 5.2: Ambiente Naked Objects com o projeto conceitual do módulo Acervo

Funcionário pode ser especializado como um Funcionário da Biblioteca ou um *Outro Funcionário* (Outro Func.). Por sua vez, um Funcionário da Biblioteca pode ser um Bibliotecário ou *Outro Funcionário da Biblioteca* (Outro Bibl.). Os detalhes sobre os atributos foram omitidos para tornar o diagrama mais simples. Concentramos nossa apresentação apenas no módulo Pessoa para não tornar essa discussão muito extensa. Foi uma escolha arbitrária, poderíamos ter optado por exemplos do módulo Acervo sem prejuízos para a apresentação.

Figura 5.3: Parte do diagrama original - módulo Pessoa

Na Figura 5.4 apresentamos a interface de validação do projeto conceitual do módulo Pessoa. Os dados da hierarquia de generalização-especialização da Figura 5.3 podem ser visualizados como um conjunto de formulários hierárquicos, ou seja, o formulário mais especializado contém o formulário da generalização. O formulário mais externo representa os dados específicos de um *BibliotecarioIME*. Esse

formulário inclui o formulário de `FuncionarioBibliotecaIME` que, por sua vez, inclui o formulário de um `Funcionario`. A hierarquia de generalização-especialização prossegue com os formulários respectivos de `PessoaUSP`, `PessoaFisica` e `Pessoa`, a raiz de toda a hierarquia. Assim representamos de forma visual uma hierarquia complexa, permitindo ao especialista expandir os formulários internos conforme o necessário, validando as características hierárquicas dos dados de forma muito mais direta que um diagrama em forma de árvore.

Figura 5.4: Ambiente Naked Objects com o projeto conceitual do módulo Pessoa

Para obter a interface da Figura 5.4, as abstrações da Figura 5.3 tiveram a seguinte representação com anotações:

```
...
@Specialization(
    specializes = PessoaUSP.class
)
@Generalization(
    completeness = Completeness.Partial,
    disjointness = Disjointness.Disjoint
)
public class Funcionario
...

...
@Specialization(
    specializes = Funcionario.class
)
@Generalization(
    completeness = Completeness.Partial,
    disjointness = Disjointness.Disjoint
)
public class FuncionarioBibliotecaIME
...

...
@Specialization(
    specializes = FuncionarioBibliotecaIME.class
)
@Entity
public class BibliotecarioIME
...
```

Além de contemplar todas as abstrações necessárias, pode-se perceber que a representação utilizando

anotações é mais concisa, não sendo necessário incluir entidades filhas como *Outro...* para indicar uma generalização parcial. O código SQL gerado é:

```
CREATE TABLE FUNCIONARIO
(
    fk_unidade NUMERIC(10) NOT NULL ,
    id_funcionario NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_FUNCIONARIO
        PRIMARY KEY (id_funcionario)
);

CREATE TABLE FUNCIONARIO_SPEC
(
    date_end DATE,
    fk_funcionario_spec NUMERIC(10) NOT NULL UNIQUE ,
    date_begin DATE,
    fk_funcionario NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_FUNCIONARIO_SPEC
        PRIMARY KEY (fk_funcionario)
);

ALTER TABLE FUNCIONARIO_SPEC
    ADD CONSTRAINT FUNCIONARIO_SPEC_FUNCIONARIO FOREIGN KEY (fk_funcionario)
    REFERENCES FUNCIONARIO (id_funcionario)
    ON DELETE CASCADE
;

CREATE TABLE FUNCIONARIO_SPEC_DESC
(
    id_funcionario_spec_desc NUMERIC(10) NOT NULL ,
    description DATE,
    CONSTRAINT PK_FUNCIONARIO_SPEC_DESC
        PRIMARY KEY (id_funcionario_spec_desc)
);

ALTER TABLE FUNCIONARIO_SPEC
    ADD CONSTRAINT FUNCIONARIO_SPEC_FUNCIONARIO_SPEC_DESC
        FOREIGN KEY (fk_funcionario_spec)
```

```
REFERENCES FUNCIONARIO_SPEC_DESC (id_funcionario_spec_desc)
ON DELETE CASCADE
;

INSERT INTO FUNCIONARIO_SPEC_DESC (id_funcionario_spec_desc, description )
VALUES (0,'FUNCIONARIOBIBLIOTECAIME');

CREATE TABLE FUNCIONARIOBIBLIOTECAIME
(
    id_funcionariobibliotecaime NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_FUNCIONARIOBIBLIOTECAIME
        PRIMARY KEY (id_funcionariobibliotecaime)
);

ALTER TABLE FUNCIONARIOBIBLIOTECAIME
    ADD CONSTRAINT FUNCIONARIOBIBLIOTECAIME_FUNCIONARIO
        FOREIGN KEY (id_funcionariobibliotecaime)
        REFERENCES FUNCIONARIO (id_funcionario)
    ON DELETE CASCADE
;

CREATE TABLE FUNCIONARIOBIBLIOTECAIME_SPEC
(
    fk_funcionariobibliotecaime_spec NUMERIC(10) NOT NULL UNIQUE ,
    date_end DATE,
    fk_funcionariobibliotecaime NUMERIC(10) NOT NULL ,
    date_begin DATE,
    CONSTRAINT PK_FUNCIONARIOBIBLIOTECAIME_SPEC
        PRIMARY KEY (fk_funcionariobibliotecaime)
);

ALTER TABLE FUNCIONARIOBIBLIOTECAIME_SPEC
    ADD CONSTRAINT FUNCIONARIOBIBLIOTECAIME_SPEC_FUNCIONARIOBIBLIOTECAIME
        FOREIGN KEY (fk_funcionariobibliotecaime)
        REFERENCES FUNCIONARIOBIBLIOTECAIME (id_funcionariobibliotecaime)
    ON DELETE CASCADE
;
```

```
CREATE TABLE FUNCIONARIOBIBLIOTECAIME_SPEC_DESC
(
    description DATE,
    id_funcionariobibliotecaime_spec_desc NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_FUNCIONARIOBIBLIOTECAIME_SPEC_DESC
        PRIMARY KEY (id_funcionariobibliotecaime_spec_desc)
);

ALTER TABLE FUNCIONARIOBIBLIOTECAIME_SPEC
    ADD CONSTRAINT FUNCIONARIOBIBLIOTECAIME_SPEC_FUNCIONARIOBIBLIOTECAIME_SPEC_DESC
        FOREIGN KEY (fk_funcionariobibliotecaime_spec)
        REFERENCES FUNCIONARIOBIBLIOTECAIME_SPEC_DESC (id_funcionariobibliotecaime_spec_desc)
        ON DELETE CASCADE
;

INSERT INTO FUNCIONARIOBIBLIOTECAIME_SPEC_DESC
    (id_funcionariobibliotecaime_spec_desc, description )
    VALUES (0,'BIBLIOTECARIOIME');

CREATE TABLE BIBLIOTECARIOIME
(
    crb VARCHAR(256),
    id_bibliotecarioime NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_BIBLIOTECARIOIME
        PRIMARY KEY (id_bibliotecarioime)
);

ALTER TABLE BIBLIOTECARIOIME
    ADD CONSTRAINT BIBLIOTECARIOIME_FUNCIONARIOBIBLIOTECAIME
        FOREIGN KEY (id_bibliotecarioime)
        REFERENCES FUNCIONARIOBIBLIOTECAIME (id_funcionariobibliotecaime)
        ON DELETE CASCADE
;
```

Esse mapeamento para o código SQL é feito diretamente pela ferramenta, evitando erros e imprecisões nessa operação. Pode-se perceber que o resultado cria um conjunto de relações auxiliares, como `FUNCIONARIO_SPEC` e `FUNCIONARIO_SPEC_DESC`, e dependências, como `FUNCIONARIO_SPEC_FUNCIONARIO`.

Para maiores detalhes sobre esse SQL gerado verifique na Seção C.1 o código gerado para o exemplo de generalização-especialização do Capítulo 3. Para verificar cada um dos mapeamentos obtidos pelas diferentes abstrações e suas diversas opções de parâmetros consulte o site <http://www.ime.usp.br/~jef/mbroinizi>.

5.2 Conclusão

Apresentamos neste capítulo um estudo de caso muito mais complexo que os exemplos apresentados nos capítulos anteriores. Pudemos evidenciar a capacidade de representação na situação de um domínio real. As classes anotadas e o SQL gerado possuem um grande volume e devido a isso foram omitidos deste texto, porém estão disponíveis na íntegra no endereço <http://www.ime.usp.br/~jef/mbroinizi>.

Capítulo 6

Conclusões

Criar o projeto conceitual de um banco de dados que represente adequadamente os requisitos de um determinado domínio é o desafio que este trabalho procurou superar. Muitos problemas encontrados na concepção de projetos conceituais de bancos de dados decorrem da má identificação de requisitos e do distanciamento do especialista de domínio nessa etapa do desenvolvimento. Para reduzir esses problemas, buscamos idéias contidas nos métodos ágeis de desenvolvimento.

Estabelecemos uma abordagem que considera o projeto conceitual de bancos de dados como uma etapa inicial do desenvolvimento de um sistema de computação. Essa etapa consiste na exploração dos requisitos e validação dos conceitos. Essa abordagem utiliza como hipótese central a colaboração do especialista de domínio durante o projeto.

Para facilitar e tornar mais ágil a interação entre o especialista de domínio e o desenvolvedor, foi criada uma ferramenta para representar o projeto conceitual de um banco de dados como um protótipo de *software* manipulável. Esse protótipo é criado no arcabouço Naked Objects, utilizando classes em Java com anotações que permitem compor o projeto conceitual como um conjunto de abstrações de dados.

Apresentamos neste capítulo as principais contribuições, os limites e indicações de trabalhos futuros.

6.1 Contribuições

6.1.1 Agilidade

Consideramos que para tornar a nossa abordagem de projeto conceitual de bancos de dados mais ágil os valores descritos no Manifesto Ágil, como apresentado na Seção 2.3, devem estar presentes:

- *indivíduos e interações em detrimento de processos e ferramentas*: A abordagem é baseada em ciclos curtos de especificação e validação do projeto conceitual. O principal responsável por solucionar o problema de criar o projeto conceitual de bancos de dados adequado é o especialista de domínio, único que possui o conhecimento necessário. Porém, para isso, ele conta com o desenvolvedor, utilizando como meio comum para a interação entre os dois o protótipo manipulável do projeto conceitual. A abordagem ressalta o papel dos indivíduos e auxilia a interação. As ferramentas são apenas o meio para facilitar a interação;
- *colaboração do cliente em detrimento de negociação de contratos*: O papel do cliente é desempenhado pelo especialista de domínio, que, como apresentado acima, deve colaborar ativamente para a solução dos problemas, não apenas cobrar o cumprimento de contratos estabelecidos;
- *software em funcionamento em detrimento de documentação detalhada*: O projeto conceitual é disponibilizado como um protótipo manipulável, eliminando a necessidade de documentos adicionais. A utilização de uma aplicação Naked Object, que disponibiliza interfaces e potencializa a capacidade de validação do especialista de domínio ao invés de um conjunto de diagramas, valoriza o *software* em funcionamento;
- *adaptação às mudanças em detrimento de seguir um plano*: A evolução do projeto conceitual de um banco de dados decorre de sucessivas interações entre os envolvidos, validações e alterações no projeto. Mesmo os resultados obtidos devem ser aprimorados, como o código SQL. A abordagem valoriza fortemente as alterações a cada ciclo de validação.

Além disso, retomando alguns dos princípios ágeis, citados na Seção 2.3:

- priorizar a satisfação do cliente com entregas contínuas de *software* em funcionamento, começando as entregas o mais cedo possível;

- aceitar e incentivar mudanças nos requisitos, mesmo que tarde no desenvolvimento;
- entregas freqüentes de *software* em funcionamento, com pequenos intervalos de poucas semanas ou meses, preferindo escalas de tempo menores;
- os especialistas de negócio e desenvolvedores devem trabalhar juntos, diariamente, durante o projeto;
- criar projetos com indivíduos motivados, fornecendo a eles o ambiente e apoio necessários, confiando neles para finalizar o trabalho;
- a melhor forma de transferir e obter informações é por meio de conversas *cara-cara*;
- *software* em funcionamento é a principal medida de progresso;
- atenção contínua à excelência técnica e bom design aumentam a agilidade;
- simplicidade - a arte de maximizar a quantidade de trabalho não feito - é essencial.

Percebemos que os sete primeiros decorrem diretamente, enquanto o penúltimo, *utilização de um bom design*, é favorecido pelo uso de abstrações de dados como os componentes fundamentais do projeto conceitual. Já o último, deve ser observado com cautela, uma vez que o projeto conceitual deve representar todos os requisitos necessários ao domínio. Porém, não se deve buscar representações adicionais que não integrem o domínio atual.

6.1.2 Precisão

O aspecto fundamental da abordagem que permite alcançar uma maior precisão do projeto conceitual é a colaboração entre o desenvolvedor e o especialista de domínio. Ambos desempenham papéis específicos. Ao desenvolvedor cabe conhecer as abstrações de dados, identificar qual abstração de dados melhor representa cada requisito do domínio, compondo diferentes abstrações de dados para representar todos os requisitos necessários. O especialista deve conhecer profundamente o domínio a ser representado e ser capaz de utilizar uma interface gráfica para validar o projeto conceitual concebido. A interação direta entre os dois permitirá capturar os requisitos na forma de abstrações de dados.

Uma vez que o projeto conceitual, composto por abstrações de dados, é validado diretamente pelo especialista, sua precisão de representação do domínio será equivalente ao conhecimento que o especialista

possui. Para domínios muito extensos ou complexos, pode ser necessário interagir com mais de um especialista. No caso da biblioteca, Capítulo 5, foi necessário consultar dois especialistas, um para cada módulo.

6.1.3 Projeto físico

Uma contribuição secundária da utilização da ferramenta é a criação de uma versão inicial do projeto físico do banco de dados relacional correspondente. Esse projeto é criado automaticamente a partir do projeto conceitual, sendo disponibilizado como um conjunto de código SQL para criação do banco de dados relacional em um gerenciador apropriado. Apesar de tratado como um produto adicional da abordagem, é importante ressaltar que a concepção automática do projeto físico evita erros no processo de mapeamento do projeto conceitual para o projeto físico. Dessa forma, a obtenção do projeto físico inicial torna-se mais rápida, não envolve esforço humano e reduz a quantidade de imprecisões de mapeamento. O projeto físico obtido representa apenas o projeto conceitual, sendo necessários ajustes, otimizações e definições físicas para torná-lo um projeto físico completo.

6.2 Trabalhos futuros

O projeto conceitual funcional, com ênfase nos comportamentos, não foi abordado neste trabalho. Dessa forma, uma sugestão de um importante trabalho futuro é o estudo completo sobre a representação dos comportamentos em um protótipo conceitual.

Outro possível trabalho futuro, que complementaria a representação das abstrações de dados, é o tratamento do conceito de totalidade de participação nos relacionamentos ou hierarquias com herança múltipla e ciclos, como descrito em em [10]. A Seção D.2 descreve o processo de inclusão de uma nova anotação na ferramenta.

Já do ponto de vista de melhoria da ferramenta para o usuário final pode-se considerar como importantes trabalhos futuros: a inclusão de um gerenciador de banco de dados orientado a objetos ou a utilização do modelo EJB [12]; e a alteração da interface gráfica de modo a criar automaticamente a classe Java anotada a partir da especificação do projeto conceitual.

Apêndice A

Naked Objects

O arcabouço *Naked Objects* é um pacote de *software open-source* escrito em *Java* que facilita a construção completa de aplicações de negócio a partir de **Naked Objects**.

Integra um ambiente de execução, que inclui um mecanismo de visualização que cria, em tempo real, uma representação manipulável pelo usuário de qualquer **Naked Object** que o usuário precisar acessar e um mecanismo, que torna os **Naked Objects** persistentes via uma classe de persistência específica. O arcabouço já vem com uma classe de persistência que armazena cada **Naked Object** como um arquivo XML. Isso agiliza o processo de prototipação, mas não é escalável. Classes de persistência mais sofisticadas devem ser escritas em sistemas reais.

Ambos os mecanismos funcionam de forma autônoma utilizando um mecanismo comum de reflexão para inspecionar suas definições de objetos de negócio quando necessários. O programador não precisa se preocupar diretamente com os mecanismos de visualização ou de persistência. O ambiente de execução também fornece a interface entre os **Naked Objects** e outros serviços de infra-estrutura tais como para segurança e autorização.

Além disso inclui também um conjunto de arcabouço de testes. O arcabouço de testes de unidade é uma simples extensão do popular arcabouço *JUnit* [19], tratando as capacidades específicas dos objetos criados no arcabouço *Naked Objects*. Existe um arcabouço separado para testes de aceitação que facilita a confecção dos testes de aceitação antes de escrever os comportamentos (como defende *eXtreme Programming*). Como a maioria desses cenários de teste representam tarefas comuns executadas pelos

usuários, este arcabouço possui a vantagem adicional de poder gerar a documentação do usuário dessas tarefas diretamente a partir da definição dos testes.

Para construir sistemas com base no arcabouço Naked Objects, o importante é se concentrar nos objetos de domínio e deixar a interface e a persistência para o arcabouço. Esses objetos são definidos como classes em *Java* seguindo um pequeno número de convenções de codificação.

O arcabouço Naked Objects permite construir sistemas de negócio apenas definindo os objetos do domínio específico. No sistema todas as ações de usuários consistem em criar ou recuperar objetos, especificando seus atributos, estabelecendo associações entre eles, ou invocando métodos em um objeto (ou coleção de objetos), ou seja, utilizando uma interface com o usuário realmente orientada a objetos.

Os autores consideram como principais benefícios da utilização do arcabouço para construir sistemas:

- o aumento na produtividade de desenvolvimento, uma vez que não é necessário escrever a interface com o usuário;
- o aumento na produtividade para manter e evoluir o sistema, considerando que os sistemas se tornam mais ágeis sendo capazes de melhor acomodar futuras alterações nos requisitos de negócio;
- a redução na dificuldade de identificar os requisitos de negócio, uma vez que os *naked objects* podem constituir uma linguagem comum entre desenvolvedores e usuários.

A.1 *Behavioural Completeness*

Esta seção, fortemente baseada em [23] e [25], apresenta o conceito de *behavioural completeness*, conceito que levou ao surgimento do arcabouço Naked Objects. Discutimos brevemente sua relevância, analisando fraquezas do projeto de sistemas, assim como, os fatores que influenciam negativamente no projeto.

Objetos que possuem a propriedade de *behavioural completeness* são denominados simplesmente de **naked objects**.

A idéia foi assim definida por Pawson e Mathews [25]:

Um naked object não apenas conhece os atributos da classe do domínio que representa, mas

também sabe como modelar o comportamento dessa classe. Dessa forma, todos os comportamentos associados a um objeto, que são necessários à aplicação em desenvolvimento, devem pertencer a esse objeto ao invés de serem implementados em algum outro lugar do sistema.

Em Simula [11], retomando as origens da orientação a objetos, essa idéia já estava presente. Para se construir sistemas utilizavam-se objetos que representavam algum elemento do domínio a ser simulado. Simular o sistema em funcionamento significava criar instâncias individuais de objetos e permitir a interação entre eles.

A palavra **encapsulamento** já foi utilizada para representar essa idéia. Mas ao analisar mais profundamente seus significados em Inglês, verificamos que seu primeiro significado indica uma existência fechada, como uma cápsula medicinal. Esse significado é facilmente mapeado para orientação a objetos: um objeto é fechado por uma interface de acesso, com sua implementação interna escondida, ou seja, a idéia da caixa-preta, contida também em outras formas de desenvolvimento baseado em componentes. Ao verificar o segundo significado identificamos uma idéia ainda mais importante para a modelagem orientada a objetos do que a simples noção de caixa-preta: encapsular é ação de algo exemplificar, ou deter, as características essenciais de outra coisa, como em “este documento encapsula nossa estratégia de marketing”. Esse segundo significado de encapsulamento remete à idéia de *behavioural completeness*.

Segundo Pawson e Mathews [25], esse princípio é essencial porque:

(...) é a chave para obter-se um dos principais benefícios da orientação a objetos: a habilidade de lidar com as mudanças inesperadas nos requisitos.

Pawson e Mathews [25] afirmam que as pessoas acham que estão usando a orientação a objetos para projetar e desenvolver sistemas, quando não estão:

As pessoas projetam sistemas que separam o processo de seus dados, embora revestidos com linguagens e tecnologias orientadas a objetos. A separação dos processos de seus dados pode estar relacionada principalmente a inércia, ou seja, as pessoas aprenderam a projetar sistemas dessa forma e têm dificuldades de pensar de outra maneira.

A inércia individual não é a única culpada. Segundo os autores ela é reforçada por práticas organizacionais consagradas que “(...) forcem a separação dos processos de seus dados, mesmo que o

desenvolvedor de *software* queira adotar uma abordagem mais pura de orientação a objetos”. Dentre tais práticas podemos destacar:

- orientação a processos de negócio;
- interfaces de usuário otimizadas a tarefas;
- métodos orientados a *use-cases*;
- o padrão *Model-View-Controller*.

Essas práticas foram projetadas para reduzir riscos no processo de desenvolvimento de *software* e claramente trazem benefícios; não podemos simplesmente descartá-las. O que os autores evidenciam é que tais práticas provocam o efeito colateral de desencorajar a *behavioural completeness* do projeto.

A.1.1 Orientação a processos de negócio

Muitos profissionais ainda definem um sistema de informação como um mecanismo de transformação de dados de entrada em informações de saída, através da aplicação seqüencial de pequenas transformações. A metáfora utilizada para isso é a da linha de produção. Atualmente, uma maneira ultrapassada e pobre de descrever as modernas capacidades da área.

A orientação a processos de negócio baseia-se em duas idéias:

- focar, organizar e alcançar os resultados definidos externamente (tais como preencher um pedido) ao invés de atividades puramente definidas internamente;
- tarefas de negócio podem e devem ser reduzidas a um processo determinista que transforma entradas em saídas.

A primeira não traz problemas quando consideramos sistemas orientados a objetos, sendo até útil. O problema é que muitas tarefas de negócio simplesmente não se encaixam no modelo de processos [7]. Essas tarefas apresentam como características a dificuldade de identificar-se entradas e saídas discretas e, de forma ainda mais séria, os passos seqüenciais que realizam a transformação.

Segundo Pawson e Mathews [25]:

Ao invés de apenas imaginar o papel dos sistemas de negócio como um meio de executar um processo determinístico, que transforma informações de entrada em informações de saída através de uma seqüência de passos que adicionam valor, precisamos encontrar metáforas alternativas (...) por meio das quais o usuário possa construir uma solução para um problema específico. (Sendo muito fácil adicionar roteiros/*scripts* otimizados para um modelo de linha de produção.)

A.1.2 Interfaces de usuário otimizadas a tarefas

A mais comum interface de usuário para sistemas de negócio expõe apenas um conjunto restrito de tarefas programadas, comumente por meio de um menu de tarefas. Uma interface pobre, mas fácil de mapear para um conjunto de transações de negócio que manipulam seqüencialmente as estruturas de dados internas.

Uma idéia na qual se baseia essa abordagem é a de que um conjunto de ações programadas é a chave para a otimização, idéia historicamente atribuída a Frederick Winslow Taylor [30] e seus princípios de gerenciamento científico. Parte do seu trabalho sugere “remover todos os direitos de decisões dos trabalhadores criando roteiros de todas as suas ações”. Muitos sistemas de negócio atuais parecem seguir essa sugestão e tratam o usuário como um mero seguidor de roteiros. O sistema controla todo o processo, subcontratando o usuário somente para aquelas subtarefas que ele não está capacitado a realizar autonomamente.

Segundo Pawson e Mathews [25] a abordagem alternativa seria projetar sistemas que tratam os usuários como solucionadores de problemas:

Muitos negócios já possuem alguns sistemas que são, por natureza, problemas a serem resolvidos, como programas de desenho, indo do PowerPoint aos sistemas CAD/CAE, passando pelas planilhas eletrônicas. No entanto, na maioria dos negócios, esses sistemas não são considerados *tradicionais*. Os sistemas tradicionais estão normalmente preocupados com o processamento padronizado das transações de negócio, e são otimizados para um conjunto de tarefas, transações que são quase sempre implementadas como processos seqüenciais.

Muitas pessoas acham que sistemas de problemas a serem resolvidos e sistemas transacionais refletem duas necessidades muito diferentes dentro do negócio, que não existe necessidade de

uni-los e que fazer tal coisa só iria tirar a otimização do processamento das tarefas padrão que representam a maior parte das atividades de negócio.

Pawson [24] sugere que existe uma necessidade muito real de trazer as duas idéias próximas uma da outra, ou seja, tornar os sistemas transacionais tradicionais tão expressivos quanto um programa de desenho. Por fim os autores [25] concluem que “ao invés de perseguir a eficiência ótima na execução de cada um dos conjuntos finitos de tarefas roteirizadas, deve-se projetar uma forma de interação com o usuário que maximize a eficácia geral do usuário em satisfazer sua gama de responsabilidades”.

A.1.3 Métodos orientados a *use-cases*

Muitos dos mais populares métodos orientados a objetos se baseiam em *use-cases* para identificar os requisitos de um sistema de negócio, capturando os objetos comuns descritos nesses *use-cases*.

Muitos problemas dessa abordagem já foram resumidos por Don Firesmith [14] e apresentados por Pawson e Mathews em [25]:

Use-cases não são orientados a objetos; cada *use-case* captura uma grande abstração funcional que pode causar inúmeros problemas associados à decomposição funcional que a tecnologia de objetos supostamente deveria evitar; e uma vez que os *use-cases* são criados antes que os objetos e classes tenham sido identificados, *use-cases* ignoram o encapsulamento de atributos e operações em objetos. Firesmith afirma ainda que uma abordagem orientada a *use-cases* resulta em um protótipo arquitetural de subsistemas, com um objeto de controle funcional isolado representando a lógica de um *use-case* individual, e vários objetos de classe com poucos comportamentos controlados pelos objetos controladores. Tais arquiteturas exibem, normalmente, encapsulamento pobre, excessivo acoplamento, e uma distribuição inadequada da inteligência da aplicação entre as classes.

Autores como Jacobson [16] afirmam que *use-cases* servem ao propósito de testar o sistema resultante: “quando todos os *use-cases* tiverem sido testados (em vários níveis) o sistema estará testado em sua totalidade”.

Pawson e Mathews [25] por fim sugerem identificar os objetos de negócio por meio de conversas diretas e não estruturadas entre usuários e desenvolvedores:

Sendo o necessário para isso um meio correto de capturar um modelo de objetos emergente na forma de um protótipo de trabalho, no qual tanto usuários quanto desenvolvedores possam entender e contribuir. Não significando o desenvolvimento de um protótipo convencional que capture os requisitos das tarefas do usuário em termos de formulários e menus mas sim um protótipo com uma interface de usuário orientada a objetos, na qual o usuário possa ver na tela, a representação direta dos relacionamentos existentes entre os objetos de negócio fundamentais, não apenas com relação à atributos e associações, mas também comportamento.

A.1.4 O padrão *Model-View-Controller*

Model-View-Controller [21] é um padrão arquitetural muito difundido que divide os objetos em três papéis distintos:

- modelo (*Model*), objetos de negócio fundamentais correspondentes às classes de negócio;
- visão (*View*), objetos que permitem a visualização do modelo ao usuário;
- controlador (*Controller*), objetos que controlam a interação entre o usuário e o modelo.

Pawson e Mathews [25] citam uma versão deste padrão [21] que é o *Model-View-Controller*, ou MVC, utilizado por eles como referência. Um outro padrão similar é o *Entity-Boundary-Controller* [17].

O MVC traz a idéia de separação de conceitos, considerando que os objetos de negócio fundamentais podem ser visualizados de várias maneiras diferentes, por exemplo, em diferentes plataformas ou diferentes apresentações visuais. Se todo o conhecimento necessário para apresentar os objetos em cada forma diferente estiver nos próprios objetos fundamentais, tais objetos ficariam inchados por possuírem muitos comportamentos duplicados.

O MVC resolve esse problema separando os conceitos. Objetos de visão especificam o que deve ser apresentado assim como de que forma isso deve aparecer em cada apresentação visual, enquanto que os objetos do modelo representam os objetos fundamentais de negócio e não têm conhecimento das diferentes apresentações visuais. Os objetos controladores têm a responsabilidade de interligar os anteriores, povoando as visões a partir dos objetos do modelo e chamando métodos dos objetos do modelo em resposta a eventos de interação do usuário com a visão.

Um efeito da abordagem MVC é descrito por Mathews e Pawson [25] como um grande problema que influencia na separação dos dados e comportamentos dos objetos:

Embora não tenha sido a intenção original da abordagem MVC, os objetos Controladores tendem a ser uma representação explícita das tarefas de negócio, especialmente se a abordagem de projeto for orientada a *use-case*, mas isso ocorre também em outros casos. Esses objetos deixam de exercer o papel limitado de ser apenas intermediadores entre interfaces de usuários e objetos de negócio, e passam a assumir o papel de roteiro de tarefas, incorporando não apenas a seqüência de atividades otimizadas, como também regras de negócio, com isso usurpando as responsabilidades que deveriam ser dos objetos de negócio fundamentais. O resultado final é que o conhecimento específico de negócio é espalhado pelos domínios do Modelo, Visão e Controlador. Qualquer mudança no Modelo de objetos fundamentais irá potencialmente exigir mudanças num grande conjunto de objetos Visão e Controlador. No entanto, não existe nada no MVC que force esta tendência, mas a prática sugere que devemos contestá-lo veementemente quando procuramos por objetos comportamentalmente completos.

Por outro lado, ressaltam que “(...) uma abordagem alternativa deve evitar cair no problema para o qual o MVC foi projetado para evitar: devendo facilitar a portabilidade de uma aplicação entre múltiplas plataformas técnicas, até entre múltiplos estilos de interação, sem necessitar que o modelo de negócio seja editado; ao mesmo tempo, ela deve acomodar a necessidade de representar múltiplas visões do modelo sob a mesma plataforma, quando genuinamente exista tal necessidade.”

Pawson e Mathews sugerem que seja fornecido um mecanismo genérico de visualização, incorporando os papéis dos objetos de visão e controlador:

(...) isso significa escrever um mecanismo de visualização para cada plataforma de cliente solicitada (por exemplo: Windows, Linux, browser web ou Palm Pilot). Uma vez que um mecanismo de visualização genérica exista para a plataforma alvo, tudo que o desenvolvedor precisa é escrever os objetos do modelo de negócio. Essa abordagem não viola a essência do MVC, mas é uma re-interpretação radical de como aplicá-la. Uma maneira de considerar seria que ela gera os objetos de visão e controlador considerando o modelo.

Apêndice B

Catálogo de Anotações

B.1 Classificação - Entidade

B.1.1 Intenção

Identificar a classe como um tipo de entidade do domínio de aplicação.

B.1.2 Motivação

Em qualquer projeto conceitual de dados para um determinado domínio de aplicação, a primeira abstração utilizada é a abstração de classificação. Em um projeto conceitual baseado, por exemplo, em prototipação, é importante identificar os elementos do protótipo que representam entidades do domínio de aplicação separando-os, dessa forma, dos elementos auxiliares.

Considere, por exemplo, um domínio de aplicação que represente uma biblioteca. As iterações iniciais com os especialistas do domínio permitem identificar claramente entidades como Livro e Editora. Além disso, por exemplo, os livros podem possuir códigos de identificação que são gerados com base em seus demais atributos. Na construção do protótipo para representar esse domínio, são sugeridas a implementação de três classes: **Book**, **Publisher** e **IdFactory**. Os papéis desempenhados por essas classes no protótipo que representa o projeto conceitual do domínio de aplicação são distintos. As classes **Book** e **Publisher** representam entidades conceituais próprias do domínio, enquanto a classe **IdFactory** representa apenas o protótipo de um comportamento identificado, não devendo ser considerada como uma entidade do

domínio.

Alguns autores defendem que apenas entidades do domínio deveriam ser representadas no projeto conceitual, considerando a separação de comportamentos em elementos que não fazem parte do domínio como um erro de projeto, como pode ser visto no Apêndice A. Não pretendemos aprofundar essa discussão, mas sim fornecer uma forma para identificar os elementos que realmente pertençam ao domínio de aplicação.

B.1.3 Estrutura

B.1.4 Participantes

- **Entidade** (Book, Publisher) - entidade do domínio de aplicação.

B.1.5 Conseqüências

Quando uma classe é criada e a abstração de classificação é aplicada, explicita-se o papel desempenhado por essa classe como elemento integrante do domínio de aplicação.

Apesar de aparentemente simples, a aplicação de forma precisa da abstração de classificação é fundamental para alcançar um projeto conceitual que represente adequadamente o domínio de aplicação.

Com a obrigatoriedade de somente poder aplicar outras abstrações sobre entidades devidamente identificadas como tais, a identificação de associações obriga a identificação adequada das entidades do domínio de aplicação, fortalecendo a precisão da representação obtida.

B.1.6 Implementação

De forma genérica, uma implementação de anotação capaz de representar a Entidade é consideravelmente simples. Anotações, como esta, que não possuem parâmetros, são denominadas *marcações*.

Definição:

```
annotation Entity { }
```

B.1.7 Exemplo de código

A implementação da abstração de classificação na nossa ferramenta:

```
@Target(ElementType.TYPE)
public @interface Entity {
}
```

A utilização da anotação Entidade para identificar a classe `Book`, representada no arcabouço `Naked Objects`, como uma entidade do domínio de aplicação seria:

```
@Entity
public class Book extends AbstractNakedObject {
    ...
    private final WholeNumber edition = new WholeNumber();
    ...
    public WholeNumber getEdition() {
        return edition;
    }
    ...
}
```

A representação da abstração utilizando a anotação é muito simples, basta adicionar a anotação `@Entity` à classe que se deseja identificar como uma entidade do domínio (no exemplo, `Book`). Com essa implementação a ferramenta é capaz de gerar automaticamente a interface gráfica abaixo:

Figura B.1: Interface gerada para a abstração de classificação

Na Figura B.1 é possível identificar, à esquerda, um ícone que representa a entidade **Book**, por meio do qual é possível criar novas instâncias da entidade e listar (ou procurar) instâncias existentes. Ao centro, uma entidade é visualizada como um formulário, o que permite editar seus valores de atributos. À direita, a mesma instância é representada na interface apenas como um ícone.

Para complementar a implementação da classe **Book** seguindo as convenções do arcabouço e obter a interface da Figura B.1, seria necessário incluir um método que retornasse o título da instância. Para esse exemplo foi codificado um método que simplesmente retorna o título *Mil e uma noites* para todas as instâncias de **Book**.

O outro produto da ferramenta é o código SQL gerado para o projeto conceitual especificado:

```
CREATE TABLE BOOK
(
    edition INTEGER,
    id_book NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_BOOK PRIMARY KEY (id_book)
);
```

O código SQL que representa a abstração também é simples. Uma tabela é definida para representar a entidade, com um identificador único como chave primária. Os atributos são incluídos com seus tipos apropriados como colunas nessa tabela.

B.2 Generalização-especialização

B.2.1 Intenção

Identificar hierarquias de dados entre as entidades.

B.2.2 Motivação

Diversas áreas da computação utilizam hierarquias para representar estruturas de dados obtidas por meio da abstração de generalização e especialização. Para a comunidade de banco de dados, sua utilização evita replicação desnecessária de dados e permite centralizar a representação de conjuntos comuns de atributos.

Consideremos que no nosso exemplo, domínio da biblioteca, seja necessário representar os estudantes que se cadastraram para retirar livros. Para isso, cria-se a entidade Estudante (**Student**) utilizando a abstração de classificação. Por outro lado, é necessário representar os empregados da biblioteca. Cria-se, então, uma entidade Empregado (**Employee**), novamente utilizando a abstração de classificação.

Durante a criação da entidade Empregado, percebe-se que um grande conjunto de atributos, referentes a dados pessoais como data de nascimento, número de documentos, nome, endereço e outros, estão presentes em ambas entidades. Além disso, alguns empregados da biblioteca são também estudantes, e esses dados estariam duplicados nos dois cadastros. Torna-se evidente a possibilidade de generalizar essas entidades, criando uma nova entidade que represente esse conjunto de atributos comuns, evitando a replicação dos dados. Seria então criada uma nova entidade chamada, por exemplo, de Pessoa (**Person**) estabelecendo uma hierarquia entre as entidades, de acordo com a figura anterior.

B.2.3 Estrutura

B.2.4 Participantes

- **Entidade pai** (Person) - entidade que contém o conjunto comum de atributos da hierarquia estabelecida;
- **Entidade filha** (Student, Employee) - entidade que herda o conjunto de atributos da entidade pai.

B.2.5 Conseqüências

A utilização desta abstração explicita as generalizações e especializações e permite explorar conceitos avançados dessas relações. O primeiro conceito evidenciado é a totalidade ou parcialidade da generalização; o segundo conceito é se a generalização é disjunta ou sobreponível; e, o terceiro, se a especialização é definida por predicado ou pelo usuário. A identificação e validação desses conceitos permitem uma profunda compreensão dos requisitos do domínio, permitindo alcançar uma maior precisão no projeto conceitual de bancos de dados.

B.2.6 Implementação

Para representar a abstração de generalização-especialização são necessárias duas anotações: generalização e especialização. Ambas as anotações devem ser utilizadas em conjunto para representar a abstração.

A anotação de generalização deve ser aplicada à classe que desempenha o papel de tipo de entidade pai em uma relação de especialização.

```
annotation Generalization{
    enum Completeness{Total, Partial}
    enum Disjointness{Disjoint, Overlapping}
    Completeness completeness();
    Disjointness disjointness();
}
```

A anotação de generalização possui dois parâmetros: o primeiro, **completeness**, indica se a generalização é **Total** ou parcial (**Partial**); o segundo, **disjointness**, permite identificar a generalização como disjunta (**Disjoint**) ou sobreponível (**Overlapping**).

Duas diferentes anotações podem ser aplicadas a um tipo de entidade filha da relação de generalização: a anotação de *Especialização* ou a anotação de *Especialização definida por Predicado*.

```
annotation Specialization{
    class specializes();
}
```

A anotação de Especialização possui um único parâmetro: (**specializes**), que indica de qual tipo de entidade pai a classe anotada é filha. A anotação de Especialização definida por Predicado é uma extensão da anotação de Especialização:

```
annotation PredicatedSpecialization {
    class specializes();
    string fieldName();
    Operator operator();
    string value();
}

enum Operator {
```

```

    equalTo, notEqualTo, lessThan,
    lessThanEqualTo, greaterThan,
    greaterThanEqualTo
}

```

A anotação possui quatro parâmetros: o primeiro (**specializes**) indica de qual tipo de entidade pai a classe anotada é filha; o segundo é o nome do atributo do tipo de entidade pai que define o predicado (**fieldName**); o terceiro é o operador condicional do predicado (**operator**); e o quarto o valor de comparação do predicado (**value**).

B.2.7 Exemplo de código

A implementação desta abstração na ferramenta apresenta as três anotações descritas.

Generalização:

```

@Target(ElementType.TYPE)
public @interface Generalization{
    public enum Completeness{Total, Partial}
    public enum Disjointness{Disjoint, Overlapping}
    Completeness completeness();
    Disjointness disjointness();
}

```

Especialização:

```

@Target(ElementType.TYPE)
public @interface Specialization{
    Class specializes();
}

```

Especialização definida por Predicado:

```
@Target(ElementType.TYPE)
public @interface PredicatedSpecialization {
    Class specializes();
    String fieldName();
    Operator operator();
    String value();
}
```

Para representar o exemplo anteriormente citado com as classes `Student`, `Employee` e `Person`, podemos utilizar as três anotações do seguinte modo:

```
@Generalization(
    completeness = Completeness.Partial,
    disjointness = Disjointness.Overlapping
)
Class Person {
    WholeNumber age;
    ...
}
...
@Specialization(
    specializes = Person.class
)
@Entity
Class Student{
    ...
}
...
@PredicatedSpecialization(
    specializes = Person.class,
    fieldName = "age",
    operator = Operator.greaterThanEqualTo,
```

```

    value = "18"
)
@Entity
Class Employee{
    ...
}

```

O comportamento da interface será dependente dos parâmetros da generalização. A interface gerada pela ferramenta é apresentada na Figura B.2, sendo a generalização sobreponível e parcial. O fato da generalização ser parcial permite criar uma instância do tipo de entidade **Person** sem que ela seja uma instância de algum dos dois tipos de entidades filhos, por exemplo, na Figura B.2a, a instância **Jack** é apenas uma instância do tipo de entidade **Person**. Caso a generalização fosse total, para se obter uma instância da entidade **Person** seria necessário criar uma instância de **Employee** ou **Student**. Por outro lado, a generalização é sobreponível, o que permite uma instância de **Person** ser especializada como mais de um tipo de entidade filho. Ou seja, uma mesma instância, por exemplo **Jack**, pode ser especializada como uma instância de **Student** e de **Employee** simultaneamente.

Figura B.2: Ambiente Naked Objects

Na Figura B.2b a instância **Jack** já foi especializada como **Student** e ainda pode ser especializada como **Employee**. Caso a generalização fosse disjunta, apenas uma única especialização seria permitida a uma mesma instância do tipo de entidade pai. Por fim, percebemos a diferença entre a utilização da anotação **Specialization** na classe **Student** e da anotação **PredicatedSpecialization** na classe **Employee**. Como não existe um predicado associado à classe **Student** sempre é possível especializar uma instância de **Person** (o menu de contexto estará habilitado, como *New Student...* na Figura B.2a), desde que ainda não seja especializada como **Student** (o menu será desabilitado, como *New Student...* na Figura B.2b). Por outro lado, para especializar uma instância de **Person** como um **Employee** o predicado deve ser atendido, no caso a idade (**age**) deve ser maior ou igual a 18 (estando desabilitado na Figura B.2a, e habilitado na Figura B.2b).

O código SQL gerado é uma possibilidade de representação da abstração de generalização-especialização em um banco de dados relacional e está apresentado integralmente na Seção C.1.

B.3 Relacionamento

B.3.1 Intenção

Identificar associações que representam relacionamentos quaisquer entre duas entidades do domínio.

B.3.2 Motivação

A identificação das diversas entidades de um domínio de aplicação, apesar de representar os dados do domínio, não é suficiente para representar o comportamento e as associações entre esses dados. A mais simples forma de associação entre entidades, uma associação de relacionamento, deve ser devidamente identificada e representada no modelo conceitual de dados.

No exemplo da biblioteca podemos identificar diversas associações entre as entidades. Consideremos, por exemplo, o caso que um livro **Book** possui uma associação de relacionamento com a editora **Publisher**. Uma editora pode ter diversos livros publicados, mas um mesmo livro é publicado por apenas uma editora. Uma mesma obra pode ser publicada por mais de uma editora, porém isso caracteriza dois livros distintos, provavelmente com o mesmo título e autor, mas editoras e códigos diferentes. As restrições de cardinalidade nessa associação qualificam o relacionamento, ressaltando a necessidade de se representar adequadamente as associações no projeto conceitual.

B.3.3 Estrutura

B.3.4 Participante

- **Entidades Associadas** (Book, Publisher) - as duas entidades que pertencem à associação de relacionamento;
- **Associação de Relacionamento** (publishes) - a associação de relacionamento que vincula as duas entidades, comumente, pode ser expressada como um verbo.

B.3.5 Conseqüências

A utilização desta abstração, além de explicitar as associações de relacionamento entre as entidades do domínio, permite identificar também a cardinalidade de cada uma dessas associações.

B.3.6 Implementação

Para representar a associação de relacionamento é necessário criar apenas mais uma anotação, que deve ser aplicada a um atributo de uma das duas entidades participantes da associação de relacionamento. Esse atributo deve ser do tipo da outra entidade participante. Da mesma forma, a outra entidade também deve possuir um atributo do tipo da entidade que teve o atributo anotado. Desse modo, é necessário apenas anotar um único atributo em apenas uma das duas entidades.

```
annotation RelationshipAssociation {  
    Cardinality cardinality();  
    class relatedWith();  
    string fieldRelatedName();  
}
```

```
enum Cardinality{  
    OneToOne, OneToMany, ManyToOne, ManyToMany  
}
```

Essa anotação possui três parâmetros: o primeiro é a restrição de cardinalidade (**cardinality**) que pode assumir os valores um para um (**OneToOne**), um para muitos (**OneToMany**), muitos para um

(`ManyToOne`) e muitos para muitos (`ManyToMany`); o segundo é a outra classe pertencente à associação (`relatedWith`); e o terceiro é o nome do atributo da outra classe da associação (`fieldRelatedName`).

B.3.7 Exemplo de código

A anotação de associação de relacionamento foi assim implementada na ferramenta:

```
@Target(ElementType.FIELD)
public @interface RelationshipAssociation {
    Cardinality cardinality();
    Class relatedWith();
    String fieldRelatedName();
}
```

A representação da associação de relacionamento entre as entidades `Item`(`Item`), uma generalização da entidade livro (`Book`), e Editora(`Publisher`) seria feita da seguinte forma:

```
@Entity
public class Item extends AbstractNakedObject{
    ...
    @RelationshipAssociation(
        cardinality = Cardinality.ManyToOne,
        relatedWith = Publisher.class,
        fieldRelatedName = "items",
    ) private Publisher publisher;
    ...
    public Publisher getPublisher() {
        resolve(this.publisher);
        return this.publisher;
    }

    public void setPublisher(Publisher publisher) {
```

```
        this.publisher = publisher;
        this.objectChanged();
    }
    ...
}

...

@Entity
public class Publisher extends AbstractNakedObject{
    ...
    private final ExtendedInternalCollection items =
        new ExtendedInternalCollection("Items",Item.class, this);
    ...
    public ExtendedInternalCollection getItems(){
        return items;
    }
    ...
}
```

Para estabelecer uma associação com a entidade **Publisher**, inserimos na classe **Item** um atributo que denominamos de **publisher**, do tipo **Publisher**. Para isso criamos uma variável membro privada do tipo adequado e métodos públicos de acesso **get** e **set**, seguindo as convenções do arcabouço Naked Objects. Com a classe definida de acordo com o arcabouço Naked Objects, começamos a anotá-la. Primeiramente, a classe **Item** é uma entidade do domínio e deve ser assim identificada. Anotamos a classe **Item** com a anotação de entidade. Em seguida, podemos então utilizar a abstração de relacionamento, anotando a variável membro **publisher** com a anotação de associação de relacionamento. A cardinalidade identificada dessa associação é *muitos para um*, assim, o valor do parâmetro **cardinality** deve ser **Cardinality.ManyToOne**. A classe com a qual se está criando uma associação é **Publisher**, identificada no parâmetro **relatedWith**. O nome do atributo que identifica a associação na classe **Publisher** será **items**, como indicado no parâmetro **fieldRelatedName**.

Para finalizar o relacionamento, basta identificar a classe `Publisher` como uma entidade do domínio e inserir o atributo `items`. Segundo as convenções do arcabouço Naked Objects, um atributo que possa possuir mais de um valor deve ser representado como uma coleção, mais precisamente uma instância da classe `InternalCollection`. Porém, como citado anteriormente, foi necessário aumentar a capacidade de representação do arcabouço Naked Objects, dessa forma deve-se utilizar uma instância da classe `ExtendedInternalCollection` que herda as características da classe `InternalCollection`, ampliando sua capacidade de representação. Para criar uma instância dessa coleção é necessário informar como parâmetros o nome do campo que conterá a coleção (com a primeira letra maiúscula), o tipo dos elementos que serão inseridos na coleção e a instância que possuirá como atributo a coleção a ser criada.

Na Figura B.3(a) vemos a representação das entidades como ícones no canto superior esquerdo. Uma instância da entidade de item, `Mil e uma noites`, é apresentada como um formulário enquanto outra é apresentada como um ícone, `A ilha do tesouro`. Semelhantemente, existem duas instâncias da entidade editora, `Faz de conta` e `Copiadora`.

Na Figura B.3(b) uma associação de relacionamento será criada ao soltar a instância `Ilha do tesouro` sobre o atributo `items` da instância `Faz de conta`. O contorno do atributo torna-se verde, indicando a possibilidade de criar a associação. O resultado dessa operação é apresentado na Figura B.4(a).

Ao criar-se a associação os atributos `items` da instância `Faz de conta` e `publisher` da instância `Ilha do tesouro` foram atualizados. Na Figura B.4(b), uma nova associação será criada, arrastando a instância `Faz de conta` e soltando-a sobre o atributo `publisher` da instância `Mil e uma noites`. Novamente essa associação é possível, representada pelo contorno verde do atributo. Podemos, assim, verificar que a cardinalidade de *muitos para um* é respeitada, permitindo-se associar mais de um item a uma mesma editora. O resultado é apresentado na Figura B.5(a)

Na Figura B.5(a) os atributos da associação foram devidamente atualizados. Na Figura B.5(b) tentamos associar o item `Mil e uma noites` à instância `Copiadora`. Para representar que essa associação não é possível, o contorno torna-se vermelho, respeitando a cardinalidade de *muitos para um*, não permitindo que um mesmo item seja associado a mais de uma editora.

O código SQL gerado para o exemplo explora um mapeamento otimizado muito utilizado para representar associações de relacionamento no modelo relacional. As tabelas `ITEM` e `PUBLISHER` representam as entidades. Para criar a associação de relacionamento, uma chave-estrangeira `fk_publisher` é inserida em

(a)

(b)

Figura B.3: Interface gerada pela ferramenta para a abstração de associação de relacionamento

ITEM e uma restrição de chave-estrangeira, ITEM_PUBLISHER, é criada para relacionar a chave-estrangeira com a chave-primária da tabela PUBLISHER.

(a)

(b)

Figura B.4: Interface gerada pela ferramenta para a abstração de associação de relacionamento

(a)

(b)

Figura B.5: Interface gerada pela ferramenta para a abstração de associação de relacionamento

```
CREATE TABLE ITEM
(
    title VARCHAR(256),
    id_item NUMERIC(10) NOT NULL ,
    fk_publisher NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_ITEM PRIMARY KEY (id_item)
);

CREATE TABLE PUBLISHER
(
    id_publisher NUMERIC(10) NOT NULL ,
    name VARCHAR(256),
    CONSTRAINT PK_PUBLISHER PRIMARY KEY (id_publisher)
);

ALTER TABLE ITEM
    ADD CONSTRAINT ITEM_PUBLISHER FOREIGN KEY (fk_publisher)
    REFERENCES PUBLISHER (id_publisher)
    ON DELETE SET NULL
;
```


B.4 Composição

B.4.1 Intenção

Identificar uma associação de *todo e parte*, ou seja, uma associação na qual uma das entidades é composta pela outra.

B.4.2 Motivação

Associações de composição são comuns em diversos domínios. Seu comportamento é muito semelhante ao das associações de relacionamento. Porém, em alguns casos, a associação de composição implica uma dependência existencial entre as entidades: quando uma instância da entidade composta deixa de existir, as instâncias das entidades que compõem essa instância também deixam de existir, sofrendo uma exclusão em cascata.

No exemplo da biblioteca, revistas são consideradas itens periódicos (**PeriodicItem**). Esses itens não são emprestados, podendo apenas ser consultados na biblioteca. Cada item representa uma determinada revista, por exemplo, *Communications*. Porém, é necessário representar os diversos volumes (**Volume**) dessa revista recebidos a cada período de tempo, que possuem números diferentes. É interessante notar que a associação entre o item periódico e seus volumes é de composição, uma vez que podemos compreender o item como a coleção de todos os volumes de um determinado título periódico. Além disso, caso uma instância de um item periódico deixe de existir para o cadastro da biblioteca, não poderia continuar existindo nenhum volume daquele item periódico na biblioteca. Isso configura a dependência existencial característica de muitas associações de composição.

B.4.3 Estrutura

B.4.4 Participante

- **Entidade Composta** (*PeriodicItem*) - uma instância dessa entidade é constituída de instâncias da entidade componente;
- **Entidade Componente** (*Volume*) - instâncias dessa entidade fazem parte de instâncias da entidade composta;
- **Associação de Composição** (*has*) - vínculo entre as entidades que determina a associação de composição, representada por um verbo que caracteriza a associação como uma associação de *todo e parte*. Uma composição pode apresentar ou não dependência existencial, sendo denominada de, respectivamente, composição física ou composição lógica.

B.4.5 Conseqüências

Esta abstração representa um forte vínculo entre classes. Esse vínculo representa uma associação entre *todo e parte*. Uma das entidades é parte componente da outra. Além de representar a cardinalidade dessa associação, a abstração de composição permite representar adequadamente se a composição é lógica ou física.

Quando uma instância de uma entidade composta que possui uma associação de composição física com alguma instância da entidade componente é removida do sistema, todas as instâncias da entidade componente também serão removidas em cascata.

B.4.6 Implementação

Representar uma associação de composição é semelhante a representar uma associação de relacionamento. Da mesma forma, a única nova anotação deve ser aplicada a um atributo de uma das duas entidades

participantes da associação. Esse atributo deve ser do tipo da outra entidade participante. A outra entidade também deve possuir um atributo do tipo da entidade que teve o atributo anotado, porém, é necessário apenas anotar um único atributo em apenas uma das duas entidades.

```
annotation CompositeAssociation {
    enum CompositeType {Logical, Physical}
    Cardinality cardinality();
    class relatedWith();
    CompositeType compositeType();
    string fieldRelatedName();
}

enum Cardinality{
    OneToOne, OneToMany, ManyToOne
}
```

Essa anotação possui quatro parâmetros: o primeiro é a restrição de cardinalidade (`cardinality`) que pode assumir os valores um para um(`OneToOne`), um para muitos(`OneToMany`) e muitos para um(`ManyToOne`); o segundo é a outra classe pertencente à associação(`relatedWith`); o terceiro indica o tipo de composição (`compositeType`) que essa anotação representa, uma composição lógica(`Logical`) ou uma composição física(`Physical`); e o quarto é o nome do atributo da outra classe que faz parte da associação (`fieldRelatedName`).

É interessante ressaltar que as cardinalidades possíveis possuem sempre um lado com multiplicidade *um*, isso ocorre uma vez que a entidade composta e as suas entidades componentes estão fortemente vinculadas, não podendo assim uma mesma instância de uma entidade ser componente de mais de uma instância de uma entidade composta.

B.4.7 Exemplo de código

A anotação de associação de composição implementada pela ferramenta é a seguinte:

```
@Target(ElementType.FIELD)
```

```

public @interface CompositeAssociation {
    public enum CompositeType{ Logical, Physical}
    Cardinality cardinality();
    Class relatedWith();
    CompositeType compositeType();
    String fieldRelatedName();
}

```

A utilização para representar o exemplo da biblioteca seria:

```

@Entity
public class PeriodicItem extends AbstractNakedObject{
    ...
    @CompositeAssociation(
        cardinality = Cardinality.OneToMany,
        relatedWith = Volume.class,
        fieldRelatedName = "periodicItem",
        compositeType = CompositeType.Physical
    )private final ExtendedInternalCollection volumes = new
        ExtendedInternalCollection("Volumes",Volume.class, this);

    public ExtendedInternalCollection getVolumes() {
        return volumes;
    }
    ...
}

...
@Entity
public class Volume extends AbstractNakedObject{
    ...
    private PeriodicItem periodicItem;
}

```

```
public PeriodicItem getPeriodicItem() {
    resolve(periodicItem);
    return periodicItem;
}

public void setPeriodicItem(PeriodicItem periodicItem) {
    this.periodicItem = periodicItem;
    objectChanged();
}

...
}
```

A interface gerada para a abstração de composição apresenta um comportamento muito semelhante ao da abstração de relacionamento, tornando a utilização da ferramenta uniforme. Na Figura B.6(a) podemos identificar as entidades item periódico (**PeriodicItem**) e **Volume**. Existem duas instâncias de itens periódicos, uma para a revista **Communications** e uma para a revista **Nature**, além de três volumes ainda não associados a nenhum item periódico.

Na Figura B.6(b) a operação de arrastar e soltar cria a associação, nesse caso definida como uma composição, entre a instância de item periódico 32256 e o volume 17. O resultado da operação pode ser visto na Figura B.7(a).

Na Figura B.7(b) verifica-se que o controle de cardinalidade acontece como na Associação de Relacionamento. Como o volume 17 já está associado com um item periódico, não é possível associá-lo à instância 56845, respeitando a cardinalidade definida como *um para muitos*.

A Figura B.8(a) apresenta a situação após as associações do volume 37 ao item periódico 32256 e do volume 42 ao item periódico 56845, representando assim que os volumes 17 e 37 são da revista **Communications**, enquanto o volume 42 é da revista **Nature**. Na verdade, essa é a representação de um caso no qual uma determinada instância componente, o volume, sempre pertencerá à mesma instância composta, já que um volume impresso de uma revista sempre será daquela mesma revista. Porém, outras composições podem ser menos restritivas, dessa forma, o padrão permite a flexibilidade de desassociar instâncias. Na Figura B.8(b), será ativada a ação de remoção da instância composta 56845. O resultado é apresentado na Figura B.9(a).

(a)

(b)

Figura B.6: Interface gerada pela ferramenta para a abstração de composição

(a)

(b)

Figura B.7: Interface gerada pela ferramenta para a abstração de composição

(a)

(b)

Figura B.8: Interface gerada pela ferramenta para a abstração de composição

(a)

(b)

Figura B.9: Interface gerada pela ferramenta para a abstração de composição

Na Figura B.9(a), a remoção da instância composta 56845, causou a remoção em cascata das instâncias 17 e 37. Além disso, a operação inversa não é verdadeira, como a ação de remover o volume 42 ativada na Figura B.9(a), não removerá em cascata a instância do item periódico 32256, apenas desassociará a instância componente que será removida, como pode ser observado com o resultado da ação apresentado na Figura B.9(b).

O script SQL gerado para a abstração de composição é equivalente ao gerado para a abstração de relacionamento. A única diferença é na cláusula ON DELETE da restrição de chave-estrangeira, que possui para as composições físicas o valor CASCADE ao invés de SET NULL.

```
CREATE TABLE PERIODICITEM
(
    id_periodicitem NUMERIC(10) NOT NULL ,
    periodicitemissn VARCHAR(256),
    CONSTRAINT PK_PERIODICITEM PRIMARY KEY (id_periodicitem)
);

CREATE TABLE VOLUME
(
    fk_periodicitem NUMERIC(10) NOT NULL ,
    id_volume NUMERIC(10) NOT NULL ,
    number VARCHAR(256),
    CONSTRAINT PK_VOLUME PRIMARY KEY (id_volume)
);

ALTER TABLE VOLUME
    ADD CONSTRAINT VOLUME_PERIODICITEM FOREIGN KEY (fk_periodicitem)
    REFERENCES PERIODICITEM (id_periodicitem)
    ON DELETE CASCADE
;
```

B.5 Objeto-Relacionamento

B.5.1 Intenção

Identificar uma associação que possui um destaque suficiente no domínio de aplicação para ser representada como uma entidade, mas mantendo características de associação entre duas entidades existentes.

B.5.2 Motivação

Quando uma associação possui atributos próprios, muitas abordagens recomendam que esses atributos sejam colocados em um dos dois tipos de entidades que participam da associação. Porém, essa não é uma solução adequada, pois ela desvincula o dado do local a que realmente pertence. Além disso, algumas associações se associam a outras entidades diretamente. Essas associações apresentam características de entidades, atributos e associações, desempenhando um papel de maior destaque no seu domínio de aplicação, do que as demais associações.

Consideremos o domínio de um consultório dentário. Um projeto inicial poderia identificar diretamente as entidades paciente (**Patient**) e dentista (**Dentist**). Poderíamos criar uma associação de relacionamento entre essas entidades, vinculando um dentista a um paciente. Esse vínculo poderia ser denominado, por exemplo, tratamento. Esse projeto simples poderia ser suficiente para representar algum domínio de aplicação, porém, nesse exemplo, um paciente pode ser atendido por mais de um dentista em um mesmo tratamento. Além disso, é necessário que, em cada vez que ocorra um atendimento, seja registrada a data desse evento. Ajustando o projeto inicial para atender a esses requisitos, percebemos que a associação entre um dentista e um paciente é para o atendimento, e não para todo o tratamento. Podemos então denominar o atendimento de consulta (**Appointment**). Mas isso acarretaria que a associação denominada de consulta possuísse o atributo data e que um tratamento fosse uma composição de consultas. Dessa forma, uma consulta apresenta simultaneamente as características e comportamentos de uma associação e de uma entidade.

B.5.3 Estrutura

B.5.4 Participante

- **Entidades Associadas** (Patient e Dentist) - as duas entidades que determinam a existência da associação;
- **Objeto-relacionamento** (Appointment) - a entidade que representa a associação e seus atributos;

B.5.5 Colaborações

Uma instância do Objeto-Relacionamento sempre será criada como consequência da associação de instâncias das Entidades Associadas. Não existe uma instância do Objeto-Relacionamento que não tenha sido criada a partir da associação de instâncias das Entidades Associadas.

B.5.6 Conseqüências

Esta abstração permite explorar e identificar um dos conceitos mais difíceis do projeto conceitual de bancos de dados: um relacionamento que deve ser representado por um tipo de entidade. Inicialmente, pode-se identificar a necessidade de um relacionamento entre dois tipos de entidades do domínio como, por exemplo, um paciente e um dentista. Esse relacionamento poderia ser representado inicialmente por uma abstração de relacionamento. É comum nesses casos conseguir identificar um nome para denominar esse relacionamento como, por exemplo, consulta. Porém, quando o relacionamento é exercitado, explicita-se aspectos, sobretudo atributos e associações próprias, que evidenciam a necessidade de representar esse relacionamento como um Objeto-Relacionamento.

B.5.7 Implementação

A abstração de Objeto-Relacionamento possui uma única anotação que deve ser aplicada como a anotação da abstração de relacionamento, sobre um atributo de uma das duas Entidades Associadas. Porém,

diferentemente do que ocorre na abstração de relacionamento, esse atributo deve ser do tipo da entidade do Objeto-Relacionamento. Além disso, essa entidade deve possuir atributos de associação do tipo das duas Entidades Associadas. A Entidade Associada que não teve um atributo anotado também deve possuir um atributo do tipo da entidade do Objeto-Relacionamento.

```
annotation RelationshipObject {
    Cardinality cardinality();
    class relatedWith();
    string fieldRelatedName();
    class compositeClass();
    string compositeFieldName();
    string compositeFieldRelatedName();
}

enum Cardinality{
    OneToOne, OneToMany, ManyToOne, ManyToMany
}
```

Essa anotação possui seis parâmetros: o primeiro é a restrição de cardinalidade (**cardinality**) que pode assumir os valores um para um (**OneToOne**), um para muitos (**OneToMany**), muitos para um (**ManyToOne**) e muitos para muitos (**ManyToMany**); o segundo é a outra Entidade Associada pertencente ao relacionamento(**relatedWith**); o terceiro é o nome do atributo do objeto-relacionamento que referencia a Entidade Associada anotada (**fieldRelatedName**); o quarto indica a classe que implementa o objeto-relacionamento (**compositeClass**) respectivo dessa associação; o quinto e o sexto são os nomes dos atributos responsáveis por representar a associação do objeto-relacionamento como a outra Entidade Associada (**compositeFieldName** na Entidade Associada e **compositeFieldRelatedName**) no Objeto-Relacionamento.

B.5.8 Exemplo de código

A implementação do exemplo abordado na ferramenta, utilizando esta abstração, seria a seguinte:

```
@Entity
public class Dentist extends AbstractNakedObject {
    ...
    private final TextString name = new TextString();
    @RelationshipObject(
        cardinality = Cardinality.ManyToMany,
        relatedWith = Patient.class,
        fieldRelatedName = "dentist",
        compositeClass = Appointment.class,
        compositeFieldName = "appointments",
        compositeFieldRelatedName = "patient"
    )
    private final ExtendedInternalCollection appointments = new
        ExtendedInternalCollection("Appointments",Appointment.class, this);

    public ExtendedInternalCollection getAppointments() {
        return appointments;
    }
    ...
}
...
@Entity
public class Appointment extends AbstractNakedObject {

    private final TextString date = new TextString();
    private Patient patient;
    private Dentist dentist;
    ...
    public TextString getDate() {
        return date;
    }
}
```

```
public Patient getPatient() {
    resolve(patient);
    return patient;
}

public void setPatient(Patient paciente) {
    this.patient = paciente;
    objectChanged();
}

public Dentist getDentist() {
    resolve(dentist);
    return dentist;
}

public void setDentist(Dentist dentista) {
    this.dentist = dentista;
    objectChanged();
}

...
}

...
@Entity
public class Patient extends AbstractNakedObject {

    private final TextString name = new TextString();
    private final ExtendedInternalCollection appointments = new
        ExtendedInternalCollection("Appointments",Appointment.class, this);

    public ExtendedInternalCollection getAppointments() {
        return appointments;
    }

    ...
}
```

Figura B.10: Ambiente Naked Objects

A interface gerada automaticamente está apresentada nas Figuras B.10(a) e B.10(b). Na B.10(a) podemos observar, à esquerda, os ícones que representam os tipos de entidade: **Dentists**, **Patients** e **Appointments**. Logo abaixo, podemos identificar uma instância do tipo de entidade **Patient**, **Bob**, representada como um ícone. À direita, uma outra instância de **Patient**, **John**, é representada por meio de um formulário, assim como a instância de **Dentist**, **Dr. Brown**, acima.

Quando arrastamos a instância **John** o cursor torna-se um ícone. Quando posicionado sobre a instância **Dr. Brown**, a borda do formulário que representa essa instância torna-se verde, indicando a possibilidade de se soltar a instância arrastada para executar uma ação, nesse caso, a criação de um relacionamento que será representado por uma instância de **Appointment**. Na Figura B.10(b), vemos o resultado da ação, criando-se uma instância de **Appointment** e o preenchimento automático dos campos responsáveis pelo relacionamento. Apesar de existir uma representação do tipo de entidade, **Appointments**, percebemos que a opção de criação de uma instância está desabilitada. A única maneira de se criar uma instância do objeto-relacionamento é por meio da ação de arrastar e soltar descrita anteriormente. Dessa forma, o padrão representa na interface gráfica as restrições contidas no conceito da abstração a que se refere.

O código SQL é apresentado na seção C.2.

Apêndice C

Código SQL

Este Apêndice está diretamente relacionado com os exemplos do Capítulo 3. Para verificar cada um dos mapeamentos obtidos pelas diferentes abstrações e suas diversas opções de parâmetros consulte o site <http://www.ime.usp.br/~jef/mbroinizi>.

C.1 Generalização-especialização

```
CREATE TABLE PERSON
(
    name VARCHAR(256),
    id_person NUMERIC(10) NOT NULL ,
    age INTEGER,
    CONSTRAINT PK_PERSON PRIMARY KEY (id_person)
);

CREATE TABLE STUDENT
(
    id_student NUMERIC(10) NOT NULL ,
    university VARCHAR(256),
    CONSTRAINT PK_STUDENT PRIMARY KEY (id_student)
```

```
);
```

```
CREATE TABLE EMPLOYEE
```

```
(
```

```
    id_employee NUMERIC(10) NOT NULL ,
```

```
    company VARCHAR(256),
```

```
    CONSTRAINT PK_EMPLOYEE PRIMARY KEY (id_employee)
```

```
);
```

```
CREATE TABLE PERSON_SPEC_DESC
```

```
(
```

```
    description VARCHAR(256),
```

```
    id_person_spec_desc NUMERIC(10) NOT NULL ,
```

```
    CONSTRAINT PK_PERSON_SPEC_DESC PRIMARY KEY (id_person_spec_desc)
```

```
);
```

```
CREATE TABLE PERSON_SPEC
```

```
(
```

```
    fk_person_spec NUMERIC(10) NOT NULL ,
```

```
    fk_person NUMERIC(10) NOT NULL ,
```

```
    date_end DATE,
```

```
    date_begin DATE,
```

```
    CONSTRAINT PK_PERSON_SPEC PRIMARY KEY (fk_person, fk_person_spec)
```

```
);
```

```
ALTER TABLE PERSON_SPEC
```

```
    ADD CONSTRAINT PERSON_SPEC_PERSON_SPEC_DESC FOREIGN KEY (fk_person_spec)
```

```
    REFERENCES PERSON_SPEC_DESC (id_person_spec_desc)
```

```
    ON DELETE CASCADE
```

```
;
```

```
ALTER TABLE EMPLOYEE
  ADD CONSTRAINT EMPLOYEE_PERSON FOREIGN KEY (id_employee)
  REFERENCES PERSON (id_person)
  ON DELETE CASCADE
;

ALTER TABLE STUDENT
  ADD CONSTRAINT STUDENT_PERSON FOREIGN KEY (id_student)
  REFERENCES PERSON (id_person)
  ON DELETE CASCADE
;

ALTER TABLE EMPLOYEE
  ADD CHECK (id_employee IN (SELECT id_person FROM PERSON WHERE age >= 18))
;

ALTER TABLE PERSON_SPEC
  ADD CONSTRAINT PERSON_SPEC_PERSON FOREIGN KEY (fk_person)
  REFERENCES PERSON (id_person)
  ON DELETE CASCADE
;

INSERT INTO PERSON_SPEC_DESC (id_person_spec_desc, description )
  VALUES (0, 'EMPLOYEE');
INSERT INTO PERSON_SPEC_DESC (id_person_spec_desc, description )
  VALUES (1, 'STUDENT');
```

Nessa representação, as entidades *Person*, *Student* e *Employee* são tabelas com os respectivos atributos. As possíveis especializações da entidade *Person* são linhas da tabela *PERSON_SPEC_DESC*. As realizações dessas especializações são representadas como linhas da tabela *PERSON_SPEC*, e como referências de chave-estrangeira: *EMPLOYEE_PERSON*, *STUDENT_PERSON* e *PERSON_SPEC_PERSON*. O predicado que define

a especialização da entidade **Employee** é definido como uma restrição de **CHECK** adicionada à tabela da entidade. No fim da representação as possíveis especializações da entidade **Person** são inseridas na tabela **PERSON_SPEC_DESC**.

Essa representação é uma opção que permite otimizações para busca dos dados necessários. Se buscarmos por um elemento de uma entidade filha, após obtermos a chave primária desse elemento podemos buscar na entidade pai diretamente pela chave-primária que apresenta mesmo valor, evitando uma operação de **JOIN** entre as entidades. Por outro lado, para se obter o mesmo efeito de substituição quando buscamos inicialmente na entidade pai, é necessário a utilização das tabelas auxiliares **PERSON_SPEC_DESC** e **PERSON_SPEC** para evitar que a busca seja feita em todas as entidades filhas. Para isso, uma busca é feita no **JOIN** dessas tabelas obtendo-se a lista de entidades filhas que possuem dados sobre a instância da entidade pai de interesse. Para concluir a operação é apenas necessário executar as buscas nas respectivas entidades filhas listadas.

Para que essas otimizações sejam úteis é necessário controlar as inserções nessas tabelas para manter as chaves-primárias da entidade pai com mesmo valor das chaves-primárias das entidades filhas, quando elas forem respectivas a uma mesma instância. Da mesma forma, um novo registro deve ser sempre incluído na tabela auxiliar **PERSON_SPEC_DESC** quando uma nova instância é incluída em alguma entidade filha. Essa tabela possui campos auxiliares para indicar início e término de uma especialização, uma otimização para o controle de histórico de especializações, se isso for necessário.

Apesar de criar todas essas estruturas auxiliares, elas não precisam ser utilizadas. Basta remover o código SQL referente a elas se não for do interesse do desenvolvedor. Além disso, um cuidado especial foi tomado para permitir que esses mapeamento possa ser alterado na ferramenta. Se o desenvolvedor quiser um mapeamento diferente ele pode alterar os resultados produzidos para atender às suas preferências.

C.2 Objeto-relacionamento

O código gerado inicialmente cria as tabelas para as entidades: Consulta (**APPOINTMENT**), Dentista (**DENTIST**) e Paciente (**PATIENT**).

```
CREATE TABLE APPOINTMENT
```

```
(
```

```
    fk_patient NUMERIC(10) NOT NULL ,
    date VARCHAR(256),
    id_appointment NUMERIC(10) NOT NULL ,
    fk_dentist NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_APPOINTMENT PRIMARY KEY (id_appointment)
);

CREATE TABLE DENTIST
(
    id_dentist NUMERIC(10) NOT NULL ,
    name VARCHAR(256),
    CONSTRAINT PK_DENTIST PRIMARY KEY (id_dentist)
);

CREATE TABLE PATIENT
(
    name VARCHAR(256),
    id_patient NUMERIC(10) NOT NULL ,
    CONSTRAINT PK_PATIENT PRIMARY KEY (id_patient)
);

ALTER TABLE APPOINTMENT
    ADD CONSTRAINT APPOINTMENT_PATIENT FOREIGN KEY (fk_patient)
    REFERENCES PATIENT (id_patient)
    ON DELETE CASCADE
;

ALTER TABLE APPOINTMENT
    ADD CONSTRAINT APPOINTMENT_DENTIST FOREIGN KEY (fk_dentist)
    REFERENCES DENTIST (id_dentist)
    ON DELETE CASCADE
```

;

A tabela **APPOINTMENTS** possui chaves-estrangeiras para as tabelas **DENTIST** e **PATIENT**, que são definidas pelas restrições **APPOINTMENT_DENTIST** e **APPOINTMENT_PATIENT**, respectivamente. Ambas as chaves impõem restrições existenciais, por meio da cláusula **ON DELETE** definida como **CASCADE**. Dessa forma, as associações intermediárias, entre as entidades associadas e o objeto-relacionamento, comportam-se como as composições físicas.

Apêndice D

Ferramenta - Extensões

Este apêndice apresenta mais detalhes sobre a ferramenta desenvolvida.

D.1 Dependências e listas de classes

Cada um dos seis componentes foram separados em um arquivo `jar` diferente:

- `semantic-tool-data-annotations.jar` - anotações para representação das abstrações de dados;
- `semantic-tool-NO-extensions.jar` - extensões do arcabouço Naked Objects;
- `semantic-tool.jar` - núcleo da ferramenta;
- `semantic-tool-NO-code.jar` - gerador de código para Naked Objects;
- `semantic-tool-database.jar` - gerador de código SQL;
- `semantic-tool-database-NOtypes.jar` - mapa de tipos SQL para Naked Objects.

D.1.1 Anotações para representação das abstrações de dados

O componente contido no arquivo `semantic-tool-data-annotations.jar` não possui dependências. As estrutura de classes desse componente é:

```
org.nakedobjects.extended
  \-- Cardinality.class
  \-- CompositeAssociation.class
  \-- CompositionOperator.class
  \-- Entity.class
  \-- Generalization.class
  \-- Operator.class
  \-- PredicatedSpecialization.class
  \-- PredicatedSpecialization2.class
  \-- RelationshipAssociation.class
  \-- RelationshipObject.class
  \-- Specialization.class
  \-- Way.class
```

D.1.2 Extensões do arcabouço Naked Objects

O componente contido no arquivo `semantic-tool-NO-extensions.jar` tem as seguintes dependências:

- `nakedobjects.jar` - arcabouço Naked Objects;
- `log4j-1.2.6.jar` - componente para log da ferramenta.

Suas classes estão assim organizadas:

```
org.nakedobjects.extended
  \-- AboutFactory.class
  \-- AutoGeneratedCaller.class

org.nakedobjects.object
  \-- ClassHelper.class

org.nakedobjects.object.collection
  \-- ExtendedInternalCollection.class
```



```
org.nakedobjects.object.reflect
  \-- ExtendedOneToManyAssociation.class
```

D.1.3 Núcleo da ferramenta

O núcleo da ferramenta está no arquivo `semantic-tool.jar`. Possui as seguintes dependências:

- `javassist.jar` - classes do Javassist;
- `log4j-1.2.6.jar` - componente para log da ferramenta;
- `semantic-tool-data-annotations.jar` - anotações para representação das abstrações de dados.

A organização de suas classes é:

```
br.usp.ime.tools.semantic
  \-- AbstractBuilder.class
  \-- AbstractBuilderFactory.class
  \-- AbstractToolFactory.class
  \-- DefaultToolFactory.class
  \-- Suffix.class
  \-- Tool.class
  \-- ToolException.class

br.usp.ime.tools.semantic.abstractions
  \-- Composite.class
  \-- CompositeAssociationAbstraction.class
  \-- Composition.class
  \-- DataAbstraction.class
  \-- EntityAbstraction.class
  \-- GeneralizationAbstraction.class
  \-- Operation.class
  \-- PredicatedSpecializationAbstraction.class
```

```
\-- PredicatedSpecializationAbstraction2.class
\-- PredicateType.class
\-- RelationshipAssociationAbstraction.class
\-- RelationshipObjectAbstraction.class
\-- SpecializationAbstraction.class
```

```
br.usp.ime.tools.semantic.abstractions.handler
```

```
\-- AbstractHandler.class
\-- AbstractHandlerFactory.class
\-- AnnotationHandler.class
\-- DefaultHandlerFactory.class
```

```
br.usp.ime.tools.semantic.notification
```

```
\-- AnnotatedNotification.class
\-- AnnotationAbstractionNotification.class
\-- ExceptionNotification.class
\-- Notification.class
```

```
br.usp.ime.tools.semantic.observer
```

```
\-- Observer.class
\-- Subject.class
\-- SubjectImpl.class
```

```
br.usp.ime.tools.semantic.parser
```

```
\-- AbstractParser.class
\-- AbstractParserFactory.class
\-- AnnotatedClassParser.class
\-- DefaultParserFactory.class
```

```
br.usp.ime.tools.semantic.properties
```

```
\-- SemanticToolProperties.class
```

```
\-- SemanticToolPropertiesImpl.class
\-- ToolProperties.class
\-- ToolPropertiesException.class
```

D.1.4 Gerador de código para Naked Objects

O arquivo `semantic-tool.jar` contém o componente que apresenta as seguintes dependências:

- `nakedobjects.jar` - arcabouço Naked Objects;
- `javassist.jar` - classes do Javassist;
- `log4j-1.2.6.jar` - componente para log da ferramenta;
- `semantic-tool-data-annotations.jar` - anotações para representação das abstrações de dados;
- `semantic-tool.jar` - núcleo da ferramenta.

O gerador de código para Naked Objects está estruturado como segue:

```
br.usp.ime.tools.semantic.code
\-- AbstractCodeBuilder.class
\-- CodeBuilder.class
\-- DefaultCodeBuilderFactory.class

br.usp.ime.tools.semantic.code.elements
\-- Composite.class
\-- Composition.class
\-- Node.class
\-- Operation.class
\-- PredicateTypesMap.class

br.usp.ime.tools.semantic.code.handlers
\-- CodeHandler.class
\-- CompositeAssociationCodeHandler.class
```

```
\-- EntityCodeHandler.class
\-- GeneralizationCodeHandler.class
\-- PredicatedSpecializationCodeHandler.class
\-- RelationshipAssociationCodeHandler.class
\-- RelationshipObjectCodeHandler.class
\-- SpecializationCodeHandler.class
```

D.1.5 Gerador de código SQL

Esse componente está no arquivo `semantic-tool-database.jar`. Suas dependências são:

- `javassist.jar` - classes do Javassist;
- `log4j-1.2.6.jar` - componente para log da ferramenta;
- `semantic-tool-data-annotations.jar` - anotações para representação das abstrações de dados;
- `semantic-tool.jar` - núcleo da ferramenta.

Além disso, para sua execução é necessário um componente auxiliar que contenha o mapa de tipos SQL para a arquitetura de origem adotada. A ferramenta traz um componente com o mapa de origem contendo os tipos do arcabouço Naked Objects, esse componente auxiliar é descrito na Seção D.1.6. O componente gerador de código SQL apresenta a seguinte estrutura de classes:

```
br.usp.ime.tools.semantic.database
\-- AbstractDatabaseBuilder.class

br.usp.ime.tools.semantic.database.handlers
\-- DatabaseHandler.class

br.usp.ime.tools.semantic.database.relational
\-- DatabaseWriter.class
\-- RelationalDatabaseBuilder.class
\-- SQLwriter.class
```

```
br.usp.ime.tools.semantic.database.relational.handlers
```

```
\-- CompositeAssociationDatabaseHandler.class  
\-- EntityDatabaseHandler.class  
\-- GeneralizationDatabaseHandler.class  
\-- PredicatedSpecializationDatabaseHandler.class  
\-- RelationshipAssociationDatabaseHandler.class  
\-- RelationshipObjectDatabaseHandler.class  
\-- SpecializationDatabaseHandler.class
```

```
br.usp.ime.tools.semantic.database.relational.schema
```

```
\-- AlterTableAdd.class  
\-- Assertion.class  
\-- Attribute.class  
\-- AttributeType.class  
\-- Check.class  
\-- Composite.class  
\-- Constraint.class  
\-- DatabaseElement.class  
\-- DatabaseSchema.class  
\-- Entity.class  
\-- Operation.class  
\-- PredicateTypesMapRelational.class  
\-- Reference.class  
\-- Select.class  
\-- SQLStatements.class  
\-- TypesMap.class  
\-- Unique.class  
\-- Where.class
```

D.1.6 Mapa de tipos SQL para Naked Objects

Contido no arquivo `semantic-tool-database-N0types.jar` esse componente é dependente de:

- `nakedobjects.jar` - arcabouço Naked Objects;
- `log4j-1.2.6.jar` - componente para log da ferramenta;
- `semantic-tool-data-annotations.jar` - anotações para representação das abstrações de dados;
- `semantic-tool.jar` - núcleo da ferramenta;
- `semantic-tool-database.jar` - gerador de código SQL.

Sua simples estrutura de classe complementa o componente anterior:

```
br.usp.ime.tools.semantic.database.relational
  \-- DefaultDatabaseBuilderFactory.class

br.usp.ime.tools.semantic.database.relational.schema
  \-- Composition.class
  \-- PredicateTypesMapRelationalN0.class
  \-- TypesMapN0.class
```

D.2 Criando novas anotações

Na lista de anotações apresentada no arquivo `semantic-tool-data-annotations.jar`, pode-se identificar uma classe de anotação que não foi citada no Capítulo 3, `PredicatedSpecialization2`. Essa classe foi incluída na ferramenta para ilustrar como extensões podem ser criadas. Apresentaremos nessa seção o processo para inclusão dessa nova anotação na ferramenta.

D.2.1 Predicados compostos

A idéia da anotação `PredicatedSpecialization2` é estender os possíveis tipos de predicados aceitos para as especializações, criando uma nova abstração de especialização que aceite dois predicados equivalentes

aos utilizados na especialização definida por predicados, combinando esses predicados com operadores lógicos. Apresentamos o código completo da classe em Java que define a nova anotação:

```
package org.nakedobjects.extended;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface PredicatedSpecialization2 {

    Class specializes();

    String fieldName();

    Operator operator();

    String value();

    CompositionOperator compOperator();

    String fieldName2();

    Operator operator2();

    String value2();
}
```

Essa classe é muito similar a classe `PredicatedSpecialization`, a diferença é a inclusão de quatro novos parâmetros: `compOperator` que conterà o operador lógico utilizado para compor os predicados e um segundo bloco de parâmetros para a definição do segundo predicado composto pelos parâmetros `fieldName2`, `operator2` e `value2`. Além disso, uma nova classe foi definida para representar o operador lógico:

```
package org.nakedobjects.extended;
```

```
public enum CompositionOperator {  
    and, or  
}
```

D.2.2 Estendendo o núcleo

Para permitir que a nova abstração seja adequadamente identificada pelo núcleo é necessário criar uma classe que a represente internamente, implementando a classe abstrata `DataAbstraction`. Como essa abstração é uma extensão de outra já existente, o mais simples é herdar da abstração original, `PredicatedSpecializationAbstraction`. Classes auxiliares devem ser criadas conforme a necessidade da abstração, como a classe `Composition`, utilizada para representar a composição dos predicados.

As associações entre a anotação as classes que definem a representação interna são feitas durante a instanciação da ferramenta. Para se obter uma instância da ferramenta é necessário utilizar uma implementação de `AbstractToolFactory`. Uma implementação, denominada `DefaultToolFactory`, é disponibilizada junto com a ferramenta. A configuração da ferramenta determina qual classe será utilizada para instanciar a ferramenta. Um exemplo de uma nova fábrica poderia ser:

```
public class ExampleToolFactory extends AbstractToolFactory{  
  
    private static final AbstractHandlerFactory  
        handlerFactory = new ExampleHandlerFactory();  
    private static final AbstractParserFactory  
        parserFactory = new DefaultParserFactory();  
  
    public AbstractParser createParser(Observer obs) {  
        return parserFactory.createParser(obs);  
    }  
  
    public AbstractParser createParser() {  
        return parserFactory.createParser();  
    }  
}
```



```
}

public AbstractHandler createHandler(Observer obs) {
    return handlerFactory.createHandler(obs);
}

public AbstractHandler createHandler() {
    return handlerFactory.createHandler();
}
}
```

A única diferença com relação à fábrica default é que o atributo `handlerFactory` recebe uma instância de uma nova fábrica de tratadores, diferente da default. Essa nova fábrica de tratadores seria:

```
public class ExampleHandlerFactory extends AbstractHandlerFactory{

    public AbstractHandler createHandler(Observer obs){
        AbstractHandler ret = new AnnotationHandler(obs);
        configureHandler(ret);
        return ret;
    }

    public AbstractHandler createHandler(){
        AbstractHandler ret = new AnnotationHandler();
        configureHandler(ret);
        return ret;
    }

    private void configureHandler(AbstractHandler handler){
        handler.registerAbstractionClass(
            Entity.class.getName(),
            EntityAbstraction.class);
    }
}
```

```
handler.registerAbstractionClass(  
    Generalization.class.getName(),  
    GeneralizationAbstraction.class);  
handler.registerAbstractionClass(  
    Specialization.class.getName(),  
    SpecializationAbstraction.class);  
handler.registerAbstractionClass(  
    PredicatedSpecialization.class.getName(),  
    PredicatedSpecializationAbstraction.class);  
  
handler.registerAbstractionClass(  
    RelationshipAssociation.class.getName(),  
    RelationshipAssociationAbstraction.class);  
handler.registerAbstractionClass(  
    CompositeAssociation.class.getName(),  
    CompositeAssociationAbstraction.class);  
handler.registerAbstractionClass(  
    RelationshipObject.class.getName(),  
    RelationshipObjectAbstraction.class);  
  
handler.registerAbstractionClass(  
    PredicatedSpecialization2.class.getName(),  
    PredicatedSpecializationAbstraction2.class);  
}  
}
```

No último comando, essa nova fábrica associa a nova anotação à nova classe de representação interna da abstração.

...

```
handler.registerAbstractionClass(  
    PredicatedSpecialization2.class.getName(),  
    PredicatedSpecializationAbstraction2.class);  
...
```

A classe `DefaultHandlerFactory` já realiza a associação para a classe `PredicatedSpecialization2`, os códigos foram apresentados para ilustrar em mais detalhes o processo completo.

Em resumo, para novas anotações é necessário criar novas classes de fábrica que realizem as associações necessárias e configurar a ferramenta para utilizar essa nova sequência de fábricas.

D.2.3 Estendendo os demais componentes

Os outros componentes, responsáveis por gerar os produtos da ferramenta, também devem ser estendidos para criar novos resultados para a nova anotação. A extensão do núcleo permite identificar a anotação, disparando os componentes geradores para obter os resultados. Todos os componentes geradores utilizados precisam estar preparados para tratar todas as anotações identificadas.

Os geradores que acompanham a ferramenta apresentam uma estrutura semelhante à estrutura da própria ferramenta, utilizando fábricas e associações de classes que geram o resultado para cada anotação identificada pela ferramenta. Nesses geradores as classes `PredicatedSpecializationCodeHandler` e `PredicatedSpecializationDatabaseHandler` tratam uniformemente as anotações de especialização definida por predicados. Isso é possível uma vez que a nova classe `PredicatedSpecializationAbstraction2` do núcleo utiliza uma representação interna compatível com a anotação `PredicatedSpecialization`. Para abstrações semelhantes às já existentes, essa é uma solução possível, sendo apenas necessário associar na fábrica do gerador a representação do núcleo `PredicatedSpecializationAbstraction2` às classes responsáveis por gerar os produtos, como em:

```
public class DefaultCodeBuilderFactory implements AbstractBuilderFactory {  
  
    public AbstractCodeBuilder createBuilder(String destinyDirectory){  
        AbstractCodeBuilder ret = new CodeBuilder(destinyDirectory);  
        configureCodeBuilder(ret);  
    }  
}
```

```
        return ret;
    }

    private void configureCodeBuilder(AbstractCodeBuilder cb){
        cb.registerCodeHandler(
            EntityAbstraction.class.getName(),
            new EntityCodeHandler());

        cb.registerCodeHandler(
            GeneralizationAbstraction.class.getName(),
            new GeneralizationCodeHandler());
        cb.registerCodeHandler(
            SpecializationAbstraction.class.getName(),
            new SpecializationCodeHandler());

        cb.registerCodeHandler(
            RelationshipAssociationAbstraction.class.getName(),
            new RelationshipAssociationCodeHandler());
        cb.registerCodeHandler(
            CompositeAssociationAbstraction.class.getName(),
            new CompositeAssociationCodeHandler());
        cb.registerCodeHandler(
            RelationshipObjectAbstraction.class.getName(),
            new RelationshipObjectCodeHandler());

        cb.registerCodeHandler(
            PredicatedSpecializationAbstraction.class.getName(),
            new PredicatedSpecializationCodeHandler());
        cb.registerCodeHandler(
            PredicatedSpecializationAbstraction2.class.getName(),
            new PredicatedSpecializationCodeHandler());
    }
}
```

```
}  
}
```

Nos últimos dois comandos podemos verificar que a mesma classe geradora é utilizada para as duas representações de especialização definida por predicado. Porém, para anotações diferentes, pode ser necessário criar novas classes geradoras e novas fábricas. Quando novas fábricas dos componentes geradores forem desenvolvidas, é necessário substituir no arquivo de configuração da ferramenta a entrada relativa ao gerador alterado, indicando a nova fábrica criada.

Dessa forma, essa ferramenta apresenta uma estrutura elaborada para estimular e simplificar a criação de extensões. Essas extensões podem ser novas anotações representando novas abstrações de dados ou novos componentes geradores, que permitam obter novos produtos com base no projeto conceitual, sejam eles classes para outros arcabouços ou linguagens, projetos físicos para outras abordagens, ou qualquer outro tipo de produto como documentos ou diagramas.

Referências Bibliográficas

- [1] Agile Alliance. Disponível em <http://www.agilealliance.org/>. Acessado em Outubro de 2006.
- [2] Agile Manifesto for software development. Disponível em <http://www.agilemanifesto.org/>. Acessado em Outubro de 2006.
- [3] S. Ambler. *Agile Database Techniques*. Wiley Publishing, Inc, 2003.
- [4] S. Ambler. *The Object Primer Third Edition Agile Model-Driven Development with UML 2.0*. Cambridge, Cambridge, UK, 2004.
- [5] S. Ambler. *Agile/Evolutionary Data Modeling: From Domain Modeling to Physical Modeling*. Disponível em <http://www.agiledata.org/essays/agileDataModeling.html>, Acessado em Outubro de 2006.
- [6] K. Beck. *Extreme Programming Explained: Embracing Change*. Addison-Wesley, 1999.
- [7] J. S. Brown and P. Duguid. *The Social Life of Information*. Harvard Business School Press, 2000.
- [8] P. P. Chen. The entity relationship model - toward an unified view of data. *ACM Transactions on Database Systems*, 1(1):9, Mar. 1976. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan Kaufmann, San Mateo, CA, 1988.
- [9] E. F. Codd. Extending the relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):394–434, Dec. 1979.
- [10] A. S. da Silva, A. H. F. Laender, and M. A. Casanova. On the relational representation of complex specialization structures. *Inf. Syst*, 25(6-7):399–415, 2000.
- [11] O. J. Dahl and K. Nygaard. Simula, an algol-based simulation language. *Communications of the ACM*(9):671-678, 1996.
- [12] Enterprise Java Beans. Disponível em <http://java.sun.com/products/ejb/>. Acessado em Outubro de 2006.
- [13] R. A. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 4th edition, 2004.
- [14] D. Firesmith. *Use Cases: The Pros and Cons in Wisdom of the Gurus*. SIGS books, 1996.
- [15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

- [16] I. Jacobson. *Object-Oriented Software Engineering. A Use Case Driven Approach*. Addison-Wesley, 1992.
- [17] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
- [18] Javassist. Disponível em <http://www.csg.is.titech.ac.jp/~chiba/javassist/>. Acessado em Outubro de 2006.
- [19] Página do junit. Disponível em <http://junit.sourceforge.net/>. Acessado em Outubro de 2006.
- [20] H. F. Korth and A. Silberschatz. *Sistema de Banco de Dados*. McGraw-Hill, 1989.
- [21] G. Krasner and S. Pope. A cookbook for using model-view-controller user interface paradigm in smaltalk-80. *Journal of Object Oriented Programming* 1(3):26-49, 1988.
- [22] Naked Objects. Disponível em <http://www.nakedobjects.org/>. Acessado em Outubro de 2006.
- [23] R. Pawson. *Naked objects*. PhD thesis, University of Dublin, Trinity College, 2004.
- [24] R. Pawson and J. L. Bravard. The case for expressive systems. *Sloan Management Review Winter 1995:41-48*, 1995.
- [25] R. Pawson and R. Matthews. *Naked Objects*. Wiley and Sons, 2002.
- [26] R. Pawson, R. Matthews, and D. Haywood. The naked object architecture series. Disponível em http://www.theserverside.com/articles/article.tss?l=NakedObjectSeries_1. Acessado em Outubro de 2006.
- [27] R. Pawson and V. Wade. Agile Development Using Naked Objects. In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 25-29, 2003 Proceedings*, volume 2675 of *Lecture Notes in Computer Science*, pages 97–103. Springer, 2003.
- [28] P. Schuh. Agility and the Database. Disponível em <http://www.agilealliance.org/articles/schuhpeteragilityandt/file>. Acessado em Outubro de 2006.
- [29] Sybase PowerDesigner. Disponível em <http://www.sybase.com/>. Acessado em Outubro de 2006.
- [30] F. Taylor. *The Principles of Scientific Management*. W.W. Norton and Co., 1911.
- [31] T. Teorey, S. Lightstone, and T. Nadeau. *Database Modeling & Design*. Morgan Kaufmann Publishers, Inc., 4th edition, 2006.
- [32] T. J. Teorey. *Database Modeling & Design*. Morgan Kaufmann Publishers, Inc., 1999.
- [33] Toad Data Moduler. Disponível em <http://www.quest.com/>. Acessado em Outubro de 2006.
- [34] Unified Model Language. Disponível em <http://www.uml.org/>. Acessado em Outubro de 2006.