

UTILIZANDO UML E PADRÓES

3^a Edição

Uma introdução à análise e ao projeto orientados
a objetos e ao desenvolvimento iterativo

CRAIG LARMAN

Prefácio de Philippe Kruchten



"Com freqüência me perguntam qual é o melhor livro para conhecer o projeto orientado a objeto. Desde que o conheci, *Utilizando UML e Padrões* é a minha sugestão."

Martin Fowler, autor de *UML Essencial*



Obra originalmente publicada sob o título
Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3rd Edition
© 2005, Pearson Education, Inc.

ISBN 0-13-148906-2

Tradução autorizada a partir do original em língua inglesa publicado por Pearson Education, Inc.,
sob o selo Prentice Hall PTR.

Capa: *Mário Röhnelt*

Leitura final: *Elisa Viali*

Supervisão editorial: *Arysinha Jacques Affonso e Denise Weber Nowaczyk*

Editoração eletrônica: *Laser House*

Reservados todos os direitos de publicação, em língua portuguesa, à
ARTMED® EDITORA S.A.
(BOOKMAN® COMPANHIA EDITORA é uma divisão da ARTMED® EDITORA S.A.)
Av. Jerônimo de Ornelas, 670 – Santana
90040-340 Porto Alegre RS
Fone (51) 3027-7000 Fax (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em parte, sob quaisquer
formas ou por quaisquer meios (eletrônico, mecânico, gravação, fotocópia, distribuição na Web e outros), sem
permissão expressa da Editora.

SÃO PAULO
Av. Angélica, 1091 – Higienópolis
01227-100 São Paulo SP
Fone (11) 3665-1100 Fax (11) 3667-1333

SAC 0800 703-3444

IMPRESSO NO BRASIL
PRINTED IN BRAZIL

CONTEÚDO DOS PRINCIPAIS TÓPICOS

Este livro introduz os tópicos pouco a pouco, desenvolvendo-os ao longo dos capítulos com os estudos de caso. Essa metodologia é útil, mas gera um problema: como encontrar a maior parte do conteúdo sobre um assunto principal (por exemplo, Projeto de Objetos)? O Índice é uma solução, mas é muito detalhado; esta listagem é outra opção.

Modelagem Ágil

- O que é modelagem ágil? 57
- O que é PU ágil? 59
- Modelagem ágil e desenho leve UML 235
- Mais sobre Desenvolvimento Iterativo e Gestão de Projetos Ágeis 665

Arquitetura

- Arquitetura Lógica e Diagrama de Pacotes UML 219
- Análise Arquitetural 542
- Refinamento da Arquitetura Lógica 560
- Projeto de Pacotes 578
- Documentação da Arquitetura: UML e o Modelo das N+1 Visões 647

Modelagem de Domínio

- Modelos de Domínio 157
- Qual é o Relacionamento entre a Camada de Domínio e o Modelo de Domínio? 228
- Refinamento do Modelo de Domínio 511

GRASP

- GRASP: Projeto de Objetos com Responsabilidades 287
- GRASP: Mais Objetos com Responsabilidades 421
- Tratamento de pagamentos com polimorfismo e fazê-lo eu mesmo 606
- Exemplo: Banco Imobiliário 611

Padrões de Projeto GoF

- O que são padrões? 294
- Aplicação de Padrões de Projeto GoF 443
- Mais Projeto de Objetos com Padrões GoF 586
- Projeto de um Framework de Persistência com Padrões 620

Desenvolvimento Iterativo

- Iterativo, Evolutivo e Ágil 45
- Requisitos Evolutivos 80
- Iteração 1 – Conceitos Básicos 149

Dos Requisitos para o Projeto – Iterativamente

- Mais sobre Desenvolvimento Iterativo e Gestão de Projetos Ágeis 665

Análise e Projeto Orientados

- Veja Modelagem de Domínio e Contratos de Operação

Projeto de Objetos

- Veja GRASP e Padrões de Projeto GoF
- Projeto de Objetos 234
- Projeto de Pacotes 578

Contratos de Operação

- Contratos de Operação 203
- Contratos de Operação e Realizações de Casos de Uso 339
- Mais sobre DSSs e Contratos 506

Padrões

- Veja GRASP e Padrões de Projeto GoF
- O que são padrões? 294

Programação

- Mapeamento de Projetos para Código 379
- Desenvolvimento Dirigido por Teste e Refatoração 395

Gestão de Projetos

- Veja Modelagem Ágil e Desenvolvimento Iterativo
- Mais sobre Desenvolvimento Iterativo e Gestão de Projetos Ágeis 665

Requisitos

- Veja Casos de Uso
- Requisitos Evolutivos 80
- Outros Requisitos 127
- Dos Requisitos para o Projeto – Iterativamente 217
- Diagramas de Atividades UML e Modelagem 483
- Diagramas de Máquina de Estados em UML e Modelagem 490

Diagramas de Seqüência do Sistema

- Diagramas de Seqüência do Sistema 195
- DSSs, Operações do Sistema, Diagramas de Intereração e Realizações de Casos de Uso 338
- Mais sobre DSSs e Contratos 506

Testando

- Desenvolvimento Dirigido por Teste e Refatoração 395

UML

- O que é UML? 39
- Aplicação de UML: Diagramas de Casos de Uso 115
- Modelos de Domínio 157
- Aplicação da UML: Diagramas de Seqüência 198
- Aplicação de UML: Diagramas de pacotes 223
- Diagramas de Intereração UML 241
- Diagramas de Classe UML 266
- Diagramas de Atividades UML e Modelagem 483
- Diagramas de Máquina de Estados em UML e Modelagem 490
- Diagramas de Caso de Uso 504
- Diagramas UML de Implantação e de Componentes 616

Unified Process

- O que é PU ágil? 59
- Há Outras Práticas Importantes de PU? 60
- Mais sobre Desenvolvimento Iterativo e Gestão de Projetos Ágeis 665

Casos de Uso

- Casos de Uso 87
- Qual é o relacionamento entre DSSs e casos de uso? 199
- O Que é Uma Realização de Caso de Uso? 336
- Como Relacionar Casos de Uso 498
- Como planejar Iterações com casos de uso e cenários 669

SUMÁRIO RESUMIDO

Parte I	Introdução.....	29
1	Análise e Projeto Orientados a Objetos	31
2	Iterativo, Evolutivo e Ágil	45
3	Estudos de Caso	69
Parte II	Concepção.....	73
4	Concepção não é a Fase de Requisitos	75
5	Requisitos Evolutivos	80
6	Casos de Uso	87
7	Outros Requisitos	127
Parte III	Elaboração: Iteração 1 – Conceitos Básicos	147
8	Iteração 1 – Conceitos Básicos	149
9	Modelos de Domínio	157
10	Diagramas de Seqüência do Sistema	195
11	Contratos de Operação	203
12	Dos Requisitos para o Projeto – Iterativamente	217
13	Arquitetura Lógica e Diagrama de Pacotes UML	219
14	Projeto de Objetos	234
15	Diagramas de Interação UML	241
16	Diagramas de Classe UML	266
17	GRASP: Projeto de Objetos com Responsabilidades	287
18	Exemplos de Projeto de Objetos com GRASP	335
19	Projetar para Visibilidade	373
20	Mapeamento de Projetos para Código	379
21	Desenvolvimento Dirigido por Teste e Refatoração	395
Parte IV	Elaboração: Iteração 2 – Mais Padrões	405
22	Ferramentas UML e UML como Documentação	407
23	Rápida Atualização da Análise	411
24	Iteração 2 – Requisitos	416
25	GRASP: Mais Objetos com Responsabilidades	421
26	Aplicação de Padrões de Projeto GoF	443
Parte V	Elaboração: Iteração 3 – Tópicos Intermediários	479
27	Iteração 3 – Requisitos	481
28	Diagramas de Atividades UML e Modelagem	483
29	Diagramas de Máquina de Estados em UML e Modelagem	490
30	Como Relacionar Casos de Uso	498
31	Mais sobre DSSs e Contratos	506
32	Refinamento do Modelo de Domínio	511
33	Análise Arquitetural	542
34	Refinamento da Arquitetura Lógica	560
35	Projeto de Pacotes	578
36	Mais Projeto de Objetos com Padrões GoF	586
37	Diagramas UML de Implantação e de Componentes	616
38	Projeto de um Framework de Persistência com Padrões	620
Parte VI	Tópicos Especiais	647
39	Documentação da Arquitetura: UML e o Modelo das N+1 Visões	649
40	Mais sobre Desenvolvimento Iterativo e Gestão de Projetos Ágeis	664

SUMÁRIO

Parte I	Introdução	29
1	Análise e Projeto Orientados a Objetos	31
1.1	O que você aprenderá? Será útil?	31
1.2	Qual é o objetivo mais importante do aprendizado?	34
1.3	O que são análise e projeto?	34
1.4	O que são análise e projeto orientados a objetos?	35
1.5	Um pequeno exemplo	36
1.6	O que é UML?	39
1.7	Modelagem visual é uma boa coisa	42
1.8	Histórico	42
1.9	Leituras recomendadas	44
2	Iterativo, Evolutivo e Ágil.	45
2.1	O que é PU? Os outros métodos são complementares?	46
2.2	O que é desenvolvimento iterativo e evolutivo?	47
2.3	O que dizer do ciclo de vida em cascata?	51
2.4	Como fazer análise e projeto iterativos e evolutivos?	53
2.5	O que é planejamento iterativo guiado por risco e guiado pelo cliente?	55
2.6	O que são métodos e atitudes ágeis?	55
2.7	O que é modelagem ágil?	57
2.8	O que é PU ágil?	59
2.9	Há outras práticas importantes de PU?	60
2.10	Quais são as fases PU?	61
2.11	Quais são as disciplinas do PU?	62
2.12	Como personalizar o processo? A pasta de desenvolvimento PU	64
2.13	Você fica sabendo que não compreendeu o desenvolvimento iterativo ou o PU quando...	66
2.14	Histórico	66
2.15	Leituras recomendadas	67
3	Estudos de Caso	69
3.1	O que é e não é abordado nos estudos de caso?	69
3.2	Estratégia dos estudos de caso: desenvolvimento iterativo + aprendizado iterativo	71
3.3	Caso um: o sistema PDV ProxGer	71
3.4	Caso dois: o sistema do jogo Banco Imobiliário	72

Parte II Concepção	73
4 Concepção não é a Fase de Requisitos	75
4.1 O que é concepção?	76
4.2 Quanto deve durar a concepção?	77
4.3 Quais artefatos podem ser iniciados na concepção?	77
4.4 Você sabe que não compreendeu a concepção quando...	79
4.5 Quanto de UML durante a concepção?	79
5 Requisitos Evolutivos	80
5.1 Definição: requisitos	81
5.2 Requisitos evolutivos vs. em cascata	81
5.3 Quais são os meios racionais de eliciar requisitos?	82
5.4 Quais são os tipos e categorias de requisitos?	83
5.5 Como os requisitos são organizados em artefatos PU?	84
5.6 O livro contém exemplo desses artefatos?	85
5.7 Leituras recomendadas	85
6 Casos de Uso	87
6.1 Exemplo	89
6.2 Definição: o que são atores, cenários e casos de uso?	89
6.3 Casos de uso e o modelo de casos de uso	90
6.4 Motivação: por que casos de uso?	90
6.5 Definição: os casos de uso são requisitos funcionais?	91
6.6 Definição: quais são os três tipos de atores?	92
6.7 Notação: quais são os três formatos comuns de casos de uso?	92
6.8 Exemplo: processar venda em estilo completo	93
6.9 Qual é o significado das seções?	99
6.10 Notação: existem outros formatos? Uma variante de duas colunas	105
6.11 Diretriz: escreva casos de uso em um estilo essencial, independente da IU	106
6.12 Diretriz: escreva casos de uso enxutos	108
6.13 Diretriz: escreva casos de uso caixa preta	108
6.14 Diretriz: considere a perspectiva do ator e do objetivo do ator	108
6.15 Diretriz: como encontrar casos de uso	109
6.16 Diretriz: que testes podem ajudar a encontrar casos de uso úteis?	113
6.17 Aplicação de UML: diagramas de casos de uso	115
6.18 Aplicação da UML: diagramas de atividade	118
6.19 Motivação: outros benefícios de casos de uso? Requisitos em contexto	118
6.20 Exemplo: Banco Imobiliário	119
6.21 Processo: como trabalhar com casos de uso em métodos iterativos?	121
6.22 Histórico	126
6.23 Leituras recomendadas	126
7 Outros Requisitos	127
7.1 Quão completos são os exemplos?	128
7.2 Diretriz: deve-se fazer a análise em profundidade durante a concepção?	128
7.3 Diretriz: esses artefatos devem ficar no site da Web do projeto?	129
7.4 Exemplo ProxGer: especificação suplementar (parcial)	130
7.5 Comentário: especificação suplementar	133
7.6 Exemplo ProxGer: visão (parcial)	135

7.7 Comentário: visão	137
7.8 Exemplo ProxGer: um Glossário (parcial)	140
7.9 Comentário: Glossário (dicionário de dados)	141
7.10 Exemplo ProxGer: regras de negócio (regras de domínio)	142
7.11 Comentário: regras de domínio	143
7.12 Processo: requisitos evolutivos em métodos iterativos	143
7.13 Leituras recomendadas	145
Parte III Elaboração: Iteração 1 – Conceitos Básicos	147
8 Iteração 1 – Conceitos Básicos	149
8.1 Requisitos e ênfase da iteração 1: habilidades fundamentais em A/POO	150
8.2 Processo: concepção e elaboração	152
8.3 Processo: planejamento da próxima iteração	155
9 Modelos de Domínio	157
9.1 Exemplo	158
9.2 O que é um modelo de domínio?	159
9.3 Motivação: por que criar um modelo de domínio?	163
9.4 Diretriz: como criar um modelo de domínio?	165
9.5 Diretriz: como encontrar classes conceituais?	165
9.6 Exemplo: encontrar e desenhar as classes conceituais	168
9.7 Diretriz: modelagem ágil – esboço de um diagrama de classes	169
9.8 Diretriz: modelagem ágil – manter o modelo em uma ferramenta?	170
9.9 Diretriz: objetos-relatório – incluir o "Recibo" no modelo?	170
9.10 Diretriz: pense como um cartógrafo; use termos do domínio	171
9.11 Diretriz: como modelar o mundo irreal?	171
9.12 Diretriz: um engano comum relativo a atributo versus classes	172
9.13 Diretriz: quando modelar com classes 'descritivas'?	172
9.14 Associações	175
9.15 Exemplo: associações nos modelos de domínio	182
9.16 Atributos	183
9.17 Exemplo: atributos em modelos de domínio	190
9.18 Conclusão: o modelo de domínio está correto?	192
9.19 Processo: modelagem de domínio iterativa e evolutiva	192
9.20 Leituras recomendadas	194
10 Diagramas de Seqüência do Sistema	195
10.1 Exemplo: DSS do ProxGer	196
10.2 O que são diagramas de seqüência do sistema?	197
10.3 Motivação: por que desenhar um DSS?	198
10.4 Aplicação da UML: diagramas de seqüência	198
10.5 Qual é o relacionamento entre DSSs e casos de uso?	199
10.6 Como denominar eventos e operações do sistema?	199
10.7 Como modelar DSSs envolvendo outros sistemas externos?	200
10.8 Que informação do DSS colocar no Glossário?	200
10.9 Exemplo: DSS do Banco Imobiliário	201
10.10 Processo: DSSs iterativos e evolutivos	201
10.11 Histórico e leituras recomendadas	202

11	Contratos de Operação	203
11.1	Exemplo	204
11.2	Definição: o que são seções de um contrato?	205
11.3	Definição: o que é uma operação do sistema	205
11.4	Definição: pós-condições	206
11.5	Exemplo: pós-condições de entrarItem	208
11.6	Diretriz: devemos atualizar o modelo de domínio?	209
11.7	Diretriz: quando os contratos são úteis?	209
11.8	Diretriz: como criar e redigir contratos	210
11.9	Exemplo: contratos PDV ProxGer	211
11.10	Exemplo: contratos do Banco Imobiliário	213
11.11	Aplicação de UML: operações, contratos e a OCL	213
11.12	Processo: contratos de operações no PU	214
11.13	Histórico	214
11.14	Leituras recomendadas	215
12	Dos Requisitos para o Projeto – Iterativamente.....	217
12.1	Fazer a coisa certa, fazer certo a coisa, iterativamente	218
12.2	Provocação das modificações iniciais	218
12.3	Toda aquela análise e modelagem não demorou semanas para ser feita?	218
13	Arquitetura Lógica e Diagrama de Pacotes UML	219
13.1	Exemplo	221
13.2	Qual é a arquitetura lógica? E as camadas?	221
13.3	Que camadas são o foco dos estudos de caso?	222
13.4	O que é arquitetura do software?	222
13.5	Aplicação de UML: diagramas de pacotes	223
13.6	Diretriz: projeto em camadas	224
13.7	Diretriz: o princípio da separação modelo-visão	230
13.8	Qual é a conexão entre DSSs, operações do sistema e camadas?	232
13.9	Exemplo: arquitetura lógica e diagrama de pacotes ProxGer	233
13.10	Exemplo: arquitetura lógica do Banco Imobiliário?	233
13.11	Leituras recomendadas	233
14	Projeto de Objetos	234
14.1	Modelagem ágil e desenho leve UML	235
14.2	Ferramentas CASE UML	236
14.3	Quanto tempo gastar desenhando UML antes de codificar?	236
14.4	Projeto de objetos: o que são modelagens estática e dinâmica?	237
14.5	A importância da habilidade de projeto de objetos sobre a habilidade com a notação UML	238
14.6	Outras técnicas de projeto de objetos: cartões CRC	239
15	Diagramas de Interação UML	241
15.1	Diagramas de seqüência e de comunicação	242
15.2	Modeladores UML novatos não prestam atenção suficiente aos diagramas de interação	245
15.3	Notação comum de diagramas de interação UML	246
15.4	Notação básica de diagramas de seqüência	247
15.5	Notação básica para diagramas de colaboração	259

16 Diagramas de Classe UML	266
16.1 Aplicação de UML: notação comum de diagrama de classes	267
16.2 Definição: diagrama de classes de projeto	268
16.3 Definição: classificador	268
16.4 Modos de mostrar atributos UML: textual e linhas de associação	269
16.5 Símbolos de anotação: notas, comentários, restrições e corpos de métodos	273
16.6 Operações e métodos	273
16.7 Palavras-chave	275
16.8 Estereótipos, perfis e etiquetas	276
16.9 Propriedades e cadeias de propriedade UML	276
16.10 Generalização, classes abstratas, operações abstratas	277
16.11 Dependência	277
16.12 Interfaces	280
16.13 Composição sobre agregação	281
16.14 Restrições	282
16.15 Associação qualificada	282
16.16 Classe associativa	283
16.17 Classes objeto unitário (singleton)	283
16.18 Classes gabarito e interfaces	284
16.19 Compartimentos definidos pelo usuário	284
16.20 Classe ativa	285
16.21 Qual é o relacionamento entre diagramas de interação e de classes?	285
17 GRASP: Projeto de Objetos com Responsabilidades	287
17.1 UML versus princípios de projeto	288
17.2 Projeto de objetos: exemplos de entradas, atividades e saídas	288
17.3 Responsabilidades e projeto guiado por responsabilidade	292
17.4 GRASP: uma abordagem metódica para projeto OO básico	293
17.5 Qual é a conexão entre responsabilidades, GRASP e diagramas UML?	293
17.6 O que são padrões?	294
17.7 Onde estamos agora?	297
17.8 Um pequeno exemplo de projeto de objetos com GRASP	297
17.9 Aplicação do GRASP ao projeto de objetos	306
17.10 Criador	307
17.11 Especialista na informação (ou especialista)	309
17.12 Acoplamento baixo	314
17.13 Controlador	318
17.14 Coesão alta	330
17.15 Leituras recomendadas	334
18 Exemplos de Projeto de Objetos com GRASP	335
18.1 O que é uma realização de caso de uso?	336
18.2 Comentários sobre artefatos	338
18.3 O que vem a seguir?	341
18.4 Realizações de caso de uso para a iteração do ProxGer	341
18.5 Realizações de casos de uso para a iteração do Banco Imobiliário	360
18.6 Processo: projeto de objetos iterativo e evolutivo	371
18.7 Resumo	372
19 Projetar para Visibilidade	373
19.1 Visibilidade entre objetos	373
19.2 O que é visibilidade?	374

20 Mapeamento de Projetos para Código	379
20.1 Programação e desenvolvimento iterativo e evolutivo	380
20.2 Mapear projetos para código	381
20.3 Criar definições de classe a partir de DCPs	381
20.4 Criação de métodos a partir de diagramas de interação	382
20.5 As classes de coleção em código	383
20.6 Tratamento de erro e exceções	384
20.7 Definição do método Venda.criarLinhaDelItem	384
20.8 Ordem de implementação	385
20.9 Desenvolvimento dirigido por teste ou com testes a priori	386
20.10 Resumo de mapeamento de projetos para código	386
20.11 Introdução à solução do programa PDV ProxGer	386
20.12 Introdução à solução do programa Banco Imobiliário	390
21 Desenvolvimento Dirigido por Teste e Refatoração.	395
21.1 Desenvolvimento dirigido por teste	396
21.2 Refatoração	399
21.3 Leituras recomendadas	404
Parte IV Elaboração: Iteração 2 – Mais Padrões	405
22 Ferramentas UML e UML como Documentação.	407
22.1 Engenharia avante, reversa e de ida e volta	408
22.2 O que é um relatório comum de características importantes?	408
22.3 O que procurar em uma ferramenta?	409
22.4 Se rascunhar UML, como atualizar os diagramas depois da codificação?	409
22.5 Leituras recomendadas	410
23 Rápida Atualização da Análise.	411
23.1 Estudo de caso: PDV proxGer	411
23.2 Estudo de caso: Banco Imobiliário	413
24 Iteração 2 – Requisitos	416
24.1 Da iteração 1 para 2	417
24.2 Requisitos e ênfase da iteração 2: projeto de objetos e padrões	418
25 GRASP: Mais Objetos com Responsabilidades.	421
25.1 Polimorfismo	422
25.2 Invenção pura	429
25.3 Indireção	433
25.4 Variações protegidas	435
26 Aplicação de Padrões de Projeto GoF.	443
26.1 Adaptador (GoF adapter)	444
26.2 Alguns princípios GRASP como uma generalização de outros padrões	446
26.3 Descobertas de “análise” durante o projeto: modelo de domínio	447
26.4 Fábrica	448
26.5 Objeto unitário (GoF singleton)	450
26.6 Conclusão dos serviços externos com problema de interfaces variadas	453

26.7	Estratégia (GoF strategy)	454
26.8	Padrão composto (GoF composite) e outros princípios de projeto	458
26.9	Fachada (GoF facade)	467
<u>26.10</u>	<u>Observador/publicação-assinatura/modelo de delegação de eventos (GoF observer/publish-subscribe/delegation event model)</u>	<u>470</u>
26.11	Conclusão	477
26.12	Leituras recomendadas	477
Parte V Elaboração: Iteração 3 – Tópicos Intermediários		479
27	Iteração 3 – Requisitos	481
27.1	PDV ProxGer	482
27.2	Banco Imobiliário	482
28	Diagramas de Atividades UML e Modelagem	483
28.1	Exemplo	483
28.2	Como aplicar os diagramas de atividade?	484
28.3	Mais notação sobre diagramas de atividade da UML	486
28.4	Diretrizes	487
28.5	Exemplo: diagrama de atividades para o ProxGer	488
28.6	Processo: diagramas de atividades no PU	488
28.7	Pano de fundo	489
29	Diagramas de Máquina de Estados em UML e Modelagem	490
29.1	Exemplo	491
29.2	Definições: eventos, estados e transições	491
29.3	Como aplicar diagramas de máquina de estados	492
29.4	Mais notação do diagrama de máquina de estados em UML	494
29.5	Exemplo: modelagem de navegação de IU com máquinas de estados	495
29.6	Exemplo: diagramas de máquina de estados para os casos de uso do ProxGer	495
29.7	Processo: diagramas de máquinas estados no PU	496
29.8	Leituras recomendadas	497
30	Como Relacionar Casos de Uso	498
30.1	O relacionamento incluir	499
30.2	Terminologia: casos de uso concreto, abstrato, de base e de adição	502
30.3	O relacionamento estender	502
30.4	O relacionamento generalizar	504
30.5	Diagramas de caso de uso	504
31	Mais sobre DSSs e Contratos	506
31.1	PDV ProxGer	506
32	Refinamento do Modelo de Domínio	511
32.1	Novos conceitos para o modelo de domínio ProxGer	512
32.2	Generalização	513
32.3	Definição de superclasses e subclasses conceituais	514
32.4	Quando definir uma subclasse conceitual?	517
32.5	Quando definir uma superclasse conceitual?	519

32.6	Hierarquias de classes conceituais no PDV_ProxGer	520
32.7	Classes conceituais abstratas	521
32.8	Modelagem de estados que sofrem modificações	524
32.9	Hierarquias de classes e herança no software	525
32.10	Classes associativas	525
32.11	Agregação e composição	527
32.12	Intervalos de tempo e preços de produto – correção de um “erro” da iteração 1	530
32.13	Nomes de papel em associações	531
32.14	Papéis como conceitos versus papéis em associações	532
32.15	Elementos derivados	533
32.16	Associações qualificadas	533
32.17	Associações reflexivas	534
32.18	Uso de pacotes para organizar o modelo de domínio	534
32.19	Exemplo: refinamentos no modelo de domínio de Banco Imobiliário	539
33	Análise Arquitetural	542
33.1	Processo: quando começamos a análise arquitetural?	543
33.2	Definição: pontos de variação e evolução	543
33.3	Análise arquitetural	543
33.4	Passos comuns na análise arquitetural	545
33.5	A Ciência: identificação e análise de fatores arquiteturais	545
33.6	Exemplo: Tabela de fatores arquiteturais parcial para a aplicação PDV ProxGer	549
33.7	A arte: resolução dos fatores arquiteturais	551
33.8	Resumo dos temas da análise arquitetural	557
33.9	Processo: arquitetura iterativa dentro do PU	558
33.10	Leituras recomendadas	559
34	Refinamento da Arquitetura Lógica	560
34.1	Exemplo: arquitetura lógica do ProxGer	561
34.2	Colaborações com o padrão camadas	565
34.3	Outros tópicos do padrão camadas	571
34.4	O princípio separação modelo-visão e comunicação “para cima”	576
34.4	Leituras recomendadas	577
35	Projeto de Pacotes	578
35.1	Diretrizes para a organização de pacotes	579
35.2	Leituras recomendadas	585
36	Mais Projeto de Objetos com Padrões GoF	586
36.1	Exemplo: PDV ProxGer	587
36.2	Uso de serviços locais em caso de falha; desempenho com cache local	587
36.3	Tratamento de falha	591
36.4	Substituição para serviços locais em caso de falha, com um procurador (GoF Proxy)	599
36.5	Projeto para requisitos não-funcionais ou de qualidade	601
36.6	Acesso a dispositivos físicos externos com adaptadores	601

36.7	Fábrica abstrata (GoF abstract factory) para famílias de objetos relacionados	603
36.8	Tratamento de pagamentos com polimorfismo e fazê-lo eu mesmo	606
<u>36.9</u>	<u>Exemplo: Banco Imobiliário</u>	<u>611</u>
36.10	Conclusão	615
37	Diagramas UML de Implantação e de Componentes.....	616
37.1	Diagramas de implantação	616
<u>37.2</u>	<u>Diagramas de componentes</u>	<u>618</u>
38	Projeto de um Framework de Persistência com Padrões	620
<u>38.1</u>	<u>O problema: objetos persistentes</u>	<u>621</u>
38.2	A solução: um serviço de persistência de um framework de persistência	622
38.3	Frameworks	622
38.4	Requisitos para o serviço e para o framework de persistência	623
38.5	Idéias-chave	623
38.6	Padrão: representação de objetos como tabelas	624
38.7	Perfil da modelagem de dados da UML	624
38.8	Padrão: identificador de objeto	625
<u>38.9</u>	<u>Como ter acesso a um serviço de persistência com uma fachada</u>	<u>626</u>
<u>38.10</u>	<u>Mapeamento de objetos: padrão mapeador de banco de dados ou intermediário do banco de dados</u>	<u>626</u>
38.11	Projeto de frameworks com o padrão Método Gabarito (Template Method)	629
38.12	Materialização com o padrão método gabarito	629
38.13	Configuração de mapeadores com uma FábricaDeMapeador	633
<u>38.14</u>	<u>Padrão: administração da cache</u>	<u>635</u>
38.15	Consolidação e ocultamento de instruções SQL em uma classe	635
<u>38.16</u>	<u>Estados transacionais e o padrão estado</u>	<u>636</u>
38.17	Projeto de uma transação com o padrão comando	639
38.18	Materialização sob demanda com um procurador virtual	641
38.19	Como representar relacionamentos em tabelas	644
38.20	Superclasse ObjetoPersistente e separação de interesses	645
38.21	Problemas que não foram resolvidos	645
Parte VI	Tópicos Especiais	647
39	Documentação da Arquitetura: UML e o Modelo das N+1 Visões	649
39.1	O DAS e sua visão arquitetural	650
39.2	Notação: a estrutura de um DAS	653
39.3	Exemplo: um DAS do PDV ProxGer	654
39.4	Exemplo: um DAS jakarta struts	659
39.5	Processo: documentação arquitetural iterativa	663
39.6	Leituras recomendadas	663
40	Mais sobre Desenvolvimento Iterativo e Gestão de Projetos Ágeis	664
40.1	Como planejar uma iteração?	665
40.2	Planejamento adaptativo versus planejamento preditivo	665

<u>40.3 Planos de fase e de iteração</u>	667
<u>40.4 Como planejar iterações com casos de uso e cenários</u>	668
<u>40.5 A (in)validade das estimativas iniciais</u>	669
<u>40.6 Organização dos artefatos de projeto</u>	670
<u>40.7 Você sabe que não entendeu o planejamento iterativo no PU quando...</u>	671
<u>40.8 Leituras recomendadas</u>	672
Bibliografia	673
Glossário	679
Índice	685

PARTE I INTRODUÇÃO

ANÁLISE E PROJETO ORIENTADOS A OBJETOS

O tempo é um grande professor, mas infelizmente ele mata todos os seus alunos.
– Hector Berlioz

Objetivos

- Descrever as metas e o escopo do livro.
- Definir análise e projeto orientados a objetos (A/POO).
- Ilustrar um breve exemplo de A/POO.
- Dar uma visão geral da UML e da modelagem visual ágil

1.1 O que você aprenderá? Será útil?

O que significa ter um bom projeto de objetos? Este livro é uma ferramenta para ajudar desenvolvedores e estudantes a aprenderem as habilidades básicas usadas na análise e no projeto orientados a objetos (A/POO). Essas habilidades são essenciais para a criação de um software bem-projetado, robusto e manutenível, usando tecnologias e linguagens orientadas a objetos, tais como Java ou C#.

O que vem a seguir?

Este capítulo introduz os objetivos do livro e A/POO. O próximo capítulo introduz o desenvolvimento iterativo e evolutivo que molda como a A/POO é apresentada neste livro. Os estudos de caso são evoluídos ao longo de três iterações.



O provérbio “possuir um martelo não torna alguém um arquiteto” é particularmente verdadeiro em relação à tecnologia de objetos. Conhecer uma linguagem orientada a objetos (como Java) é um primeiro passo necessário, mas insuficiente, para criar sistemas orientados a objetos. Saber “pensar em termos de objetos” é crucial.

Esta é uma introdução à A/POO aplicando a Linguagem de Modelagem Unificada (UML) e padrões. Também, ao desenvolvimento iterativo, usando uma abordagem ágil ao processo unificado como um exemplo de processo iterativo. Ele não pretende ser um texto avançado; antes, enfatiza o domínio dos fundamentos, como atribuir responsabilidades a objetos, notação UML freqüentemente usada e padrões de projeto comuns. Ao mesmo tempo, principalmente nos capítulos posteriores, o material avança para tópicos de nível intermediário, como projeto de um framework e análise arquitetural.

UML vs. Pensar em objetos

O livro não trata somente da UML. A UML é uma notação padrão de diagramação. Embora seja útil aprender a notação, há questões mais cruciais orientadas a objetos para aprender; especificamente, como pensar em objetos. A UML não é A/POO ou um método, é apenas uma notação de diagramação. Assim, não adianta aprender diagramação UML e, talvez, uma ferramenta CASE UML, e não ser capaz de criar um excelente projeto OO, ou avaliar e melhorar um existente. Esta é a habilidade mais difícil e de maior importância. Conseqüentemente, este livro é uma introdução ao projeto de objetos.

Ainda assim, é necessária uma linguagem para a A/POO e para as “plantas do software”, tanto como uma ferramenta de raciocínio quanto uma forma de comunicação. Portanto, este livro mostra como aplicar a UML na execução de A/POO, e sobre a notação UML freqüentemente usada.

POO: princípios e padrões

Como as **responsabilidades** devem ser atribuídas a classes de objetos? Como os objetos devem interagir? Quais classes devem fazer o quê? Estas são questões importantes no projeto de um sistema e este livro ensina a clássica metáfora de projeto OO: **projeto guiado por responsabilidades**. Também, certas soluções consagradas para os problemas de projeto podem ser (e têm sido) expressas na forma de princípios de melhores práticas, heurísticas ou **padrões** – fórmulas do tipo problema-solução, devidamente nomeadas, que codificam princípios exemplares de projeto. Este livro, ao ensinar como aplicar padrões ou princípios, permite um aprendizado mais rápido e o uso eficiente desses idiomas fundamentais do projeto de objetos.

Estudos de caso

Esta introdução à A/POO é ilustrada por meio de alguns estudos de caso em desenvolvimento que são discutidos ao longo de todo o livro, em cuidadosa e profunda abordagem da análise e do projeto, de modo que detalhes difíceis do que deve ser considerado e solucionado em um problema real são tratados e resolvidos.

Casos de uso

POO (e todo projeto de software) está fortemente relacionado à atividade pré-requisito de **análise de requisitos**, a qual inclui escrever **casos de uso**. Portanto, o estudo de caso começa com uma introdução a esses tópicos, embora eles não sejam especificamente orientados a objetos.

Desenvolvimento iterativo, modelagem ágil e um PU ágil

Considerando as muitas atividades possíveis, desde a análise de requisitos até a implementação, como deve proceder um desenvolvedor ou uma equipe de desenvolvimento? A análise de requisitos e a A/POO precisam ser apresentadas e praticadas no contexto de algum processo de desenvolvimento. Nesse caso, uma **abordagem ágil** (leve, flexível) para o bem conhecido **Processo Unificado** (PU) é usada como exemplo de **processo de desenvolvimento iterativo**, na qual estes tópicos são introduzidos. Entretanto, os tópicos de análise e projeto cobertos são comuns a muitas abordagens e seu aprendizado no contexto de um PU ágil não invalida sua aplicação a outros métodos, como Scrum, Feature-Driven Development, Lean Development, Crystal Methods, etc.

Concluindo, este livro ajuda um estudante ou um desenvolvedor a:

- Aplicar princípios e padrões para criar melhores projetos de objetos.
- Seguir iterativamente um conjunto de atividades comuns de análise e projeto, baseando-se em uma abordagem ágil para o PU como exemplo.
- Criar diagramas freqüentemente usados na notação UML.

Isso é ilustrado no contexto de estudos de caso a longo prazo, que evoluem por meio de diversas iterações.



Figura 1.1 Tópicos e habilidades abordadas.

Muitas outras habilidades são importantes!

Este não é um *Livro Completo de Software*; é, principalmente, uma introdução à A/POO, UML e desenvolvimento iterativo, tocando em assuntos correlatos. Construir um software demanda muitas habilidades e passos, por exemplo, engenharia de usabilidade, projeto da interface com o usuário e projeto de base de dados são decisivas para seu sucesso.

1.2 Qual é o objetivo mais importante do aprendizado?

Existem muitas atividades e artefatos possíveis na A/POO introdutória, além de uma vasta gama de princípios e diretrizes. Suponha que devamos escolher uma única habilidade prática dentre todos os tópicos discutidos aqui – uma habilidade a ser empregada em uma “ilha deserta”. Qual deveria ser ela?

Uma habilidade crucial no desenvolvimento OO é atribuir, habilmente, responsabilidades aos objetos de software.

Por quê? Porque essa é uma atividade que precisa ser executada – seja ao desenhar um diagrama UML ou ao programar – e ela influencia drasticamente a robustez, a facilidade de manutenção e a reusabilidade de componentes de software.

Naturalmente, existem outras habilidades importantes na A/POO, mas a *atribuição de responsabilidades* é enfatizada nesta introdução porque ela tende a ser uma habilidade difícil de ser dominada (com muitos “graus de liberdade” ou alternativas), e no entanto, é de vital importância. Em um projeto real, um desenvolvedor pode não ter a oportunidade de executar quaisquer outras atividades de modelagem – um processo de desenvolvimento do tipo “corrida para codificar”. Apesar disso, mesmo nessa situação, atribuir responsabilidades é inevitável.

Conseqüentemente, os passos de projeto neste livro enfatizam princípios de atribuição de responsabilidades.

São apresentados e aplicados nove princípios fundamentais para projeto de objetos e atribuição de responsabilidades. Eles são organizados em uma forma que ajuda o aprendizado, chamada **GRASP**, com princípios que possuem nomes, tais como *Especialista em Informação* e *Criador*.

1.3 O que são análise e projeto?

A **análise** enfatiza uma *investigação* do problema e dos requisitos, em vez de uma solução, por exemplo, se desejamos um novo sistema online de comercialização, como ele será usado? Quais são as suas funções?

"Análise" é um termo de significado amplo, melhor qualificado como *análise de requisitos* (uma investigação dos requisitos) ou *análise orientada a objetos* (uma investigação dos objetos do domínio).

O projeto enfatiza uma solução conceitual (em software ou hardware) que satisfaça os requisitos e não sua implementação. Uma descrição de um esquema de banco de dados e objetos de software é um bom exemplo. Idéias de projeto excluem freqüentemente detalhes de baixo nível ou "óbvios" – óbvios para os consumidores visados. Em última instância, projetos podem ser implementados e a implementação (como por exemplo, o código) expressa o verdadeiro e completo projeto realizado.

Da mesma forma que na análise, o termo projeto é melhor qualificado como *projeto de objetos* ou *projeto de banco de dados*.

A análise e o projeto úteis foram resumidos na frase *faça a coisa certa (análise) e faça certo a coisa (projeto)*.

1.4 O que são análise e projeto orientados a objetos?

Durante a **análise orientada a objetos**, há uma ênfase em encontrar e descrever os objetos – ou conceitos – no domínio do problema. Por exemplo, no caso de um sistema de informação de vôo, alguns dos conceitos incluem *avião*, *vôo* e *piloto*.

Durante o **projeto orientado a objetos** (ou simplesmente projeto de objetos), há uma ênfase na definição dos objetos de software e como eles colaboram para a satisfação dos requisitos. Por exemplo, um objeto de software *avião* pode ter um atributo *numDaCauda* e um método *obterHistoricoDoVoo* (ver Figura 1.2).

Finalmente, durante a implementação ou programação orientada a objetos, os objetos de projeto são implementados, como, por exemplo, uma classe *Avião* em Java.

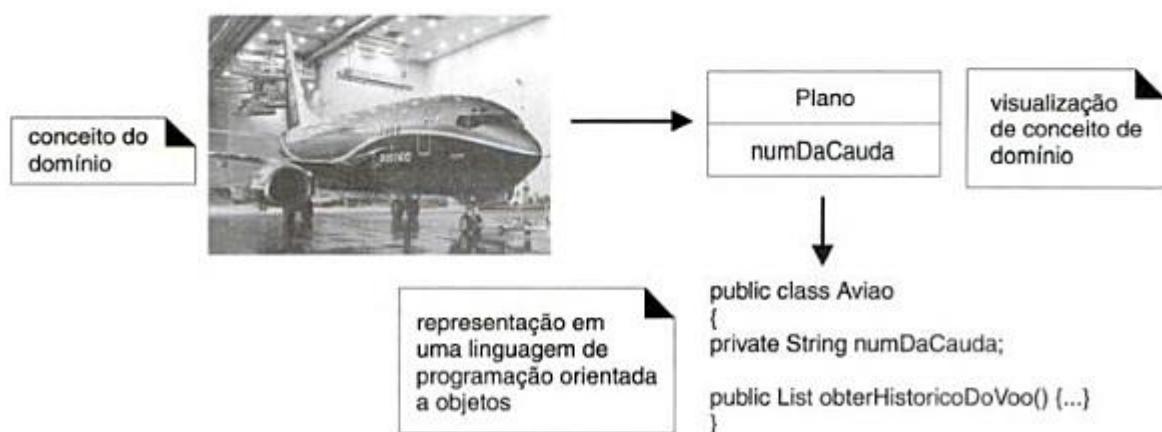


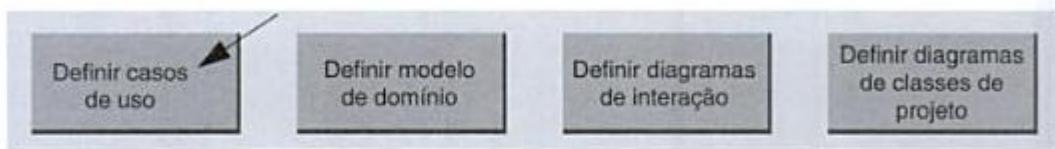
Figura 1.2 A orientação a objetos enfatiza a representação de objetos.

1.5 Um pequeno exemplo

Antes de aprofundar os detalhes do desenvolvimento iterativo UML, da análise de requisitos e da A/POO, esta seção apresenta uma visão geral de uns poucos passos-chave e diagramas, usando um exemplo simples – um jogo de dados no qual um jogador lança dois dados. Se o total for sete, ele vence; caso contrário, perde.



Definir casos de uso

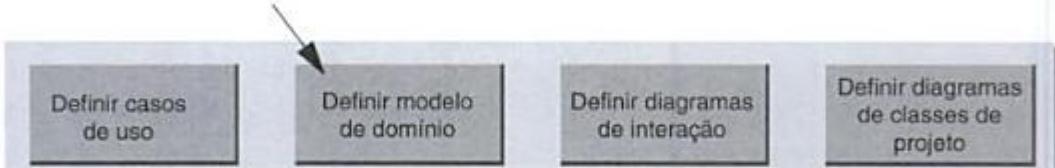


A análise de requisitos pode incluir narrativas ou cenários sobre como as pessoas usam a aplicação; estes podem ser escritos como **casos de uso**.

Casos de uso não são artefatos orientados a objetos – eles são simplesmente narrativas escritas. Contudo, são uma ferramenta popular para a utilização na análise de requisitos. Por exemplo, temos uma versão simplificada do caso de uso *Jogar um Jogo de Dados*:

Jogar um Jogo de Dados: um jogador pede que os dados sejam lançados. O sistema apresenta o resultado: se a soma do valor das faces dos dados totalizar sete, ele vence; caso contrário, perde.

Definir um modelo de domínio



A análise orientada a objetos se preocupa com a criação de uma descrição do domínio, a partir da perspectiva dos objetos. Há uma identificação dos conceitos, atributos e associações que são considerados de interesse.

O resultado pode ser expresso em um **modelo de domínio**, que mostra os conceitos ou objetos do domínio que são de interesse.

Por exemplo, um modelo parcial de domínio é mostrado na Figura 1.3.

Esse modelo ilustra os conceitos de interesse *Jogador*, *Dado* e *JogoDeDados*, com suas associações e atributos.

Note que um modelo de domínio não é uma descrição dos objetos de software; é uma visualização de conceitos ou modelos mentais do domínio do mundo real. Assim, tem também sido chamado de **modelo conceitual de objetos**.

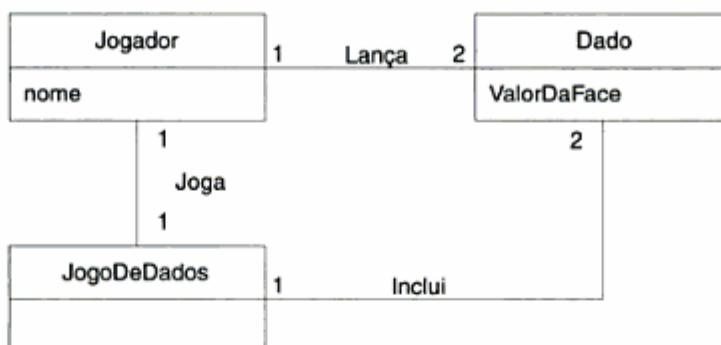
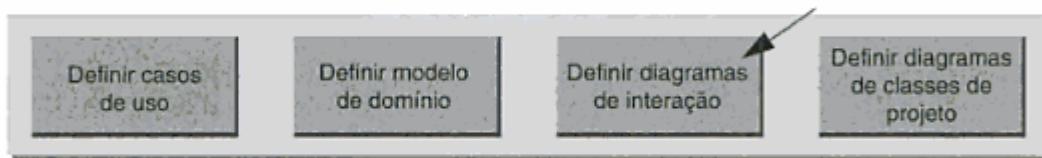


Figura 1.3 Modelo parcial de domínio do jogo de dados.

Atribuir responsabilidade aos objetos e desenhar os diagramas de interação



O projeto orientado a objetos se preocupa com a definição de objetos de software e suas responsabilidades e colaborações. Uma notação comum para ilustrar essas colaborações é o **diagrama de seqüência** (uma espécie de diagrama de interação da UML). Ele mostra o fluxo de mensagens entre os objetos de software e, assim, a invocação de métodos.

Por exemplo, o diagrama de seqüência da Figura 1.4 ilustra um projeto de software OO, em que são enviadas mensagens a instâncias das classes *JogoDeDados* e *Dado*. Note que isso ilustra o modo comum no mundo real pelo qual a UML é aplicada: rascunhando em um quadro branco

Observe que, embora no mundo real um jogador lance os dados, no projeto de software o objeto *JogoDeDados* “lança” os dados (isto é, envia mensagens para objetos *Dado*). Objetos do projeto de software e programas se inspiram, em parte, em domínios do mundo real, mas eles *não* são modelos diretos ou simulações do mundo real.

Definir diagramas de classes de projeto



Além de uma visão *dinâmica* de objetos colaborativos mostrada nos diagramas de interação, é útil criar uma visão *estática* das definições de classes com um **diagrama de classes de projeto**. Esse diagrama ilustra os atributos e métodos das classes.

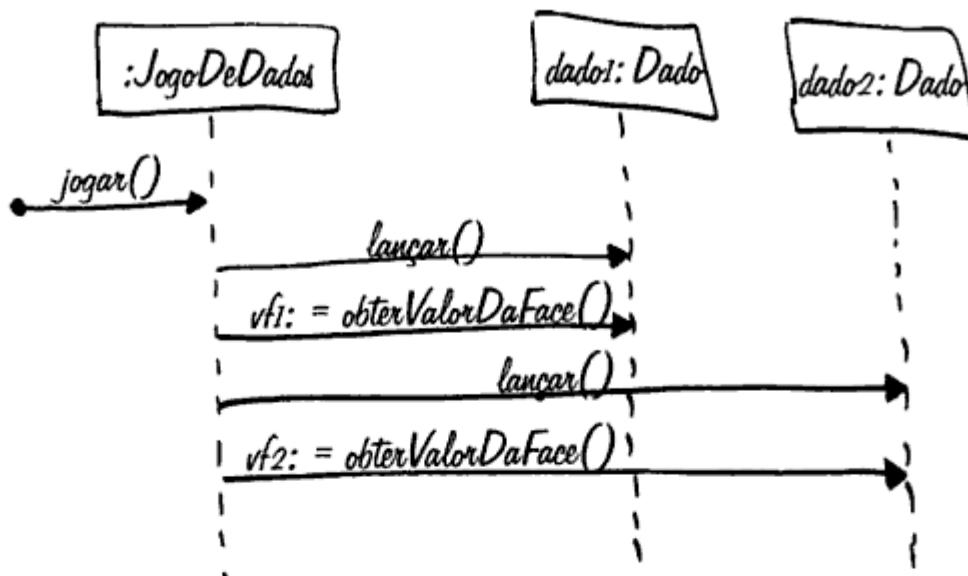


Figura 1.4 Diagrama de seqüência ilustrando mensagens entre os objetos de software.

Por exemplo, no jogo de dados, uma inspeção do diagrama de seqüência leva ao diagrama de classes de projeto parcial mostrado na Figura 1.5. Uma vez que uma mensagem `jogar` é enviada para um objeto `JogoDeDados`, a classe `JogoDeDados` requer um método `jogar` e a classe `Dado` requer um método `lançar` e um método `obterValorDaFace`.

Ao contrário do modelo de domínio, que mostra as classes do mundo real, esse diagrama mostra as classes de software.

Note que, apesar do diagrama de classes de projeto não ser o mesmo que o modelo de domínio, alguns nomes de classe e conteúdos são semelhantes. Assim, os projetos e linguagens OO podem favorecer um **baixo hiato representacional** entre os componentes de software e nossos modelos mentais de um domínio. Isso melhora a compreensão.

Resumo

O jogo de dados é um problema simples, apresentado com a intenção de focalizar alguns dos passos e artefatos na análise e projeto orientados a objetos. Visando manter

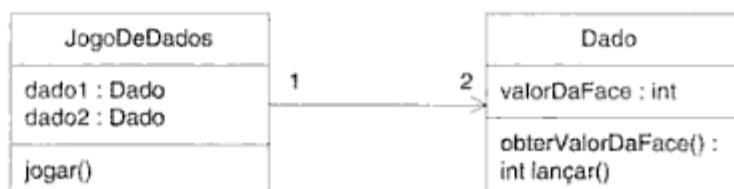


Figura 1.5 Diagrama de classes de projeto parcial.

a introdução simples, não foi explicada toda a notação UML ilustrada. Os próximos capítulos explorarão a análise, o projeto e esses artefatos em mais detalhes.

1.6 O que é UML?

Para citar:

A Linguagem de Modelagem Unificada (UML) é uma linguagem visual para especificar, construir e documentar os artefatos dos sistemas [OMG03a].

A palavra *visual* na definição é um ponto chave – a UML é a *notação diagramática* padrão, de fato, para desenhar ou apresentar figuras (com algum texto) relacionadas a software – principalmente software OO.

Este livro não cobre minuciosamente os aspectos da UML, que é uma notação volumosa. Ele enfoca os diagramas mais usados, os recursos mais comuns nesses diagramas e a notação básica, que é a que tem menor possibilidade de mudar nas futuras versões da UML.

A UML define vários **perfis UML** que especializam subconjuntos da notação para áreas de assunto comum, tais como diagramação de Enterprise JavaBeans (com o *perfil UML EJB*).

Em um nível mais baixo – de interesse principalmente para vendedores de ferramentas CASE para a **arquitetura guiada por modelos** (Model Driven Architecture – MDA) – subjacente à notação UML, está o meta-modelo da **UML**, que descreve a semântica dos elementos de modelagem. Não se trata de algo que o desenvolvedor precisa saber.

Três modos de aplicar UML

Em [Fowler03] três modos pelos quais as pessoas aplicam UML são apresentados:

- **UML como rascunho** – diagramas incompletos e informais (freqüentemente rascunhados à mão em quadros brancos) criados para explorar partes difíceis do problema ou espaço de soluções, explorando o poder das linguagens visuais.
- **UML como planta de software** – diagramas de projeto relativamente detalhados usados seja para: 1) engenharia reversa para visualizar e melhor entender o código existente em diagramas UML; seja para 2) geração de código (engenharia avante).
 - Em engenharia reversa uma ferramenta UML lê o código fonte ou o código binário e gera (tipicamente) diagramas UML de pacotes, de classes e de seqüência. Essas “plantas de software” podem ajudar o leitor a entender os elementos, estrutura e colaborações globais.
 - Antes da programação, alguns diagramas detalhados podem fornecer diretrizes para a geração de código (por exemplo, em Java), quer manualmente quer automaticamente, com uma ferramenta. É comum que os diagramas sejam usados para uma parte do código e outra parte seja preenchida por um desenvolvedor que esteja codificando (talvez também aplicando rascunhos UML).

UML e a idéia da “bala de prata”

Há um artigo bem conhecido de 1986, intitulado “Não é uma Bala de Prata”, escrito pelo Dr. Frederick Brooks, também publicado no seu livro clássico *Mythical Man Month* (edição de 20º aniversário). Leitura recomendada! Um ponto essencial é que é um erro fundamental (até agora repetido continuamente) acreditar que existe alguma ferramenta ou técnica especial em software que vá fazer uma dramática diferença em ordem de magnitude em produtividade, redução de defeitos, confiabilidade ou simplicidade. *E ferramentas não compensam a ignorância em projeto.*

No entanto, você ouvirá alegações – usualmente de vendedores de ferramentas – de que o desenho de diagramas UML vai tornar as coisas muito melhores, ou de que as ferramentas de arquitetura guiada por modelos (MDA) baseadas na UML vão ser a bala de prata que vai superar os limites.

Hora de cair na real. A UML é simplesmente uma notação padrão de diagramação – caixas, linhas, etc. A modelagem visual como uma notação comum pode ser uma grande ajuda, mas dificilmente é tão importante quanto saber como projetar e pensar em termos de objetos. Esse conhecimento de projeto é uma habilidade muito mais importante e diferente, e não é conseguida pelo aprendizado da notação UML ou pelo uso de uma ferramenta CASE ou MDA. Uma pessoa que não tenha boas habilidades de projeto e programação OO que desenha UML, está somente desenhando maus projetos. Eu sugiro o artigo *Death by UML Fever* [Bell04] (endossado pelo criador da UML Grady Booch) para mais informações neste assunto e também *What UML Is and Isn't* [Larman04].

Assim, este livro é uma introdução à A/POO e à aplicação da UML para apoiar um projeto orientado a objetos competente.

- **UML como linguagem de programação** – especificação executável completa de um sistema de software em UML. Código executável será automaticamente gerado, mas não é normalmente visto ou modificado por desenvolvedores; trabalha-se apenas na “linguagem de programação” UML. Esse uso da UML requer um modo prático de diagramar todo o comportamento ou a lógica (provavelmente usando diagramas de interação ou estado) e está ainda em desenvolvimento em termos de teoria, ferramentas robustas e usabilidade.

Modelagem ágil enfatiza a UML como rascunho; trata-se de um modo comum de aplicar a UML, freqüentemente com alto retorno no investimento de tempo (que é tipicamente curto). As ferramentas UML podem ser úteis, mas eu incentivo as pessoas a também considerar uma abordagem ágil de modelagem para aplicar a UML.

Três perspectivas para aplicar a UML

A UML descreve tipos de esboço de diagramas, tais como diagramas de classe e diagramas de seqüência. Ela não superpõe a eles uma perspectiva de modelagem. Por exemplo, a mesma notação UML de diagrama de classes pode ser usada para desenhar imagens de conceitos do mundo real ou de classes de software em Java.

Essa colocação foi enfatizada no método orientado a objetos Syntropy [CD94]. Isto é, a mesma notação pode ser usada para três perspectivas e tipos de modelos (Figura 1.6):

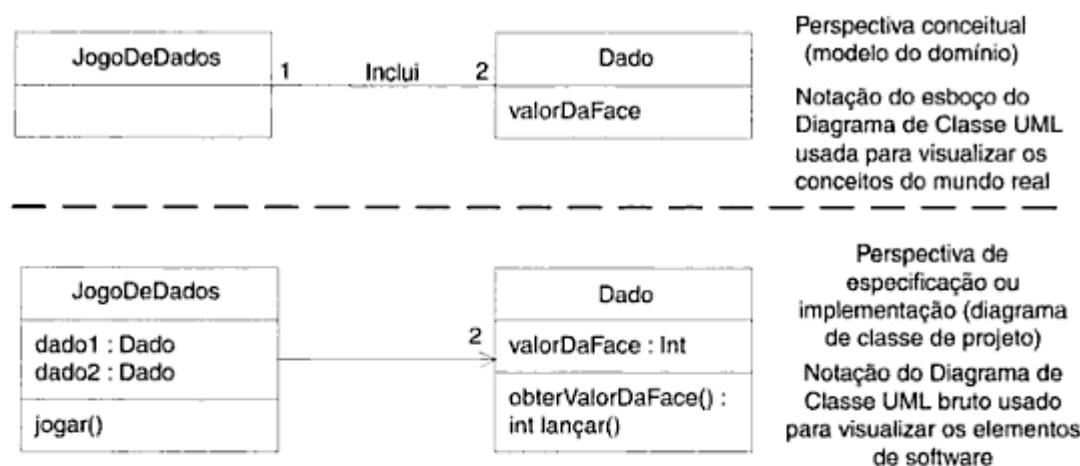


Figura 1.6 Diferentes perspectivas em UML.

1. **Perspectiva Conceitual** – os diagramas são interpretados como descrevendo coisas em uma situação do mundo real ou domínio de interesse.
2. **Perspectiva de Especificação (software)** – os diagramas (usando a mesma notação da perspectiva conceitual) descrevem abstrações de software ou componentes com especificações e interfaces, mas nenhum comprometimento com uma implementação particular (por exemplo, não especificamente uma classe em C# ou Java).
3. **Perspectiva de Implementação (software)** – os diagramas descrevem implementações de software em uma tecnologia particular (tal como Java).

Já vimos um exemplo disso nas Figuras 1.3 e 1.5, nas quais a mesma notação de diagrama de classes UML é usada para visualizar um modelo de domínio e um modelo de projeto.

Na prática, a perspectiva de especificação (adiando a definição da tecnologia alvo, tal como Java versus .NET) é raramente usada para projeto; a maior parte da diagramação em UML orientada a software considera uma perspectiva de implementação.

O significado de “classe” em diferentes perspectivas

Na UML pura, as caixas retangulares mostradas na Figura 1.6. são chamadas **classes**, mas esse termo engloba uma variedade de fenômenos – coisas físicas, conceitos abstratos, coisas de software, eventos, etc.¹

Um método sobrepõe a terminologia alternativa à UML pura. Por exemplo, no PU, quando as caixas UML são desenhadas no modelo de domínio elas são chamadas **conceitos de domínio** ou **classes conceituais**; o modelo de domínio mostra uma perspectiva conceitual. No PU, quando as caixas UML são desenhadas no modelo de

¹ Uma classe UML é um caso especial do elemento geral de modelo UML **classificador** – algo com características e/ou comportamento estrutural, inclusive classes, atores, interfaces e casos de uso.

projeto, elas são chamadas **classes de projeto**; o Modelo de Projeto mostra uma perspectiva de especificação ou implementação, conforme desejado pelo modelador.

Para manter as coisas claras, este livro irá usar termos relacionados à classe consistentes com a UML e o PU, como segue:

- **Classe Conceitual** – coisa ou conceito do mundo real. Uma perspectiva conceitual ou essencial. O Modelo de Domínio no PU contém classes conceituais.
- **Classe de Software** – classe que representa uma perspectiva de especificação ou implementação de um elemento de software, independente do processo ou método.
- **Classe de Implementação** – classe implementada em uma linguagem OO específica, como a Java.

UML 1 e UML 2

No final de 2004 emergiu uma importante nova versão da UML, a UML 2. Este texto é baseado na UML 2; a notação aqui usada foi cuidadosamente revista com membros-chave da equipe de especificação da UML 2.

Por que durante alguns capítulos não veremos muito da UML?

Este não é essencialmente um livro sobre a notação UML, mas sim um livro que explora o panorama da aplicação da UML, padrões e um processo iterativo no contexto da A/POO e análise de requisitos relacionada. A/POO é normalmente precedida pela análise de requisitos. Portanto, os capítulos iniciais apresentam uma introdução aos tópicos de casos de uso e análise de requisitos, seguidos por capítulos sobre a A/POO, além de mais detalhes da UML.

1.7 Modelagem visual é uma boa coisa

Correndo o risco de afirmar o óbvio, o desenho ou a leitura de UML implica que estamos trabalhando mais visualmente, explorando a nossa capacidade cerebral de rapidamente abranger símbolos, unidades e relacionamentos em (predominantemente) notações de caixas e linhas em 2D.

Essa idéia simples e antiga é freqüentemente perdida entre tantos detalhes e ferramentas UML. Não deveria ser! Diagramas nos ajudam a ver ou explorar mais do panorama e relacionamentos entre elementos de análise ou software, ao mesmo tempo em que nos permitem ignorar ou ocultar detalhes desinteressantes. Esse é o valor simples e essencial da UML ou de qualquer linguagem de diagramação.

1.8 Histórico

O histórico da A/POO tem muitas ramificações e este breve resumo não pode fazer justiça a todos os contribuintes. As décadas de 1960 e 1970 viram o surgimento de linguagens de programação OO, como Simula ou Smalltalk, com contribuintes-chave, tais como Kristen Nygaard e, especialmente, Alan Kay, o cientista de computação de

grande visão que criou o Smalltalk. Kay cunhou os termos *programação orientada a objetos* e *computação pessoal*, e ajudou a reunir as idéias do PC moderno enquanto estava na Xerox PARC.²

Mas a A/POO era informal ao longo daquele período e foi só em 1982 que o desenvolvimento OO emergiu como um tópico propriamente dito. Esse marco chegou quando Grady Booch (também um fundador da UML) escreveu o seu primeiro artigo intitulado *Object-Oriented Design*, provavelmente cunhando o termo [Booch82]. Muitos outros pioneiros bem conhecidos de A/POO desenvolveram suas idéias durante a década de 1980: Kent Beck, Peter Coad, Don Firesmith, Ivar Jacobson (um fundador da UML), Steve Mellor, Bertrand Meyer, Jim Rumbaugh (um fundador da UML) e Rebecca Wirfs-Brock, entre outros. Meyer publicou um dos primeiros livros influentes, *Object-Oriented Software Construction* em 1988. E Mellor e Schlaer publicaram *Object-Oriented Systems Analysis* cunhando o termo *análise orientada a objetos*, no mesmo ano. Peter Coad criou um método completo de A/POO no final da década de 1980 e publicou, em 1990 e 1991, os volumes *Object-Oriented Analysis* e *Object-Oriented Design*. Também em 1990 Wirfs-Brock e outros descreveram a abordagem de projeto guiada por responsabilidades para POO, no seu popular livro *Designing Object-Oriented Software*. Em 1991, dois livros muito populares de A/POO foram publicados. Um descrevia o método OMT, *Object Oriented Modeling and Design*, por Rumbaugh et al. O outro descrevia o método Booch, *Object Oriented Design with Applications*. Em 1992 Jacobson publicou o popular livro *Object Oriented Software Engineering* que promoveu não apenas a A/POO, mas casos de uso para requisitos.

A UML começou como um esforço de Booch e Rumbaugh, em 1994, não apenas com o intuito de criar uma notação comum, mas de combinar seus dois métodos – os métodos de Booch e OMT. Assim, o primeiro rascunho público do que hoje é a UML foi apresentado como o Método Unificado (Unified Method). Ivar Jacobson, o criador do método Objectory, se juntou a eles na Rational Corporation e, já formando um grupo, vieram a ser conhecidos como os *três amigos*. Foi nesse ponto que eles decidiram reduzir o escopo do seu esforço e enfocar em uma notação comum de diagramação – a UML – em vez de um método comum. Esse não foi apenas um esforço de diminuição do escopo; o Object Management Group (OMG, um grupo de padrões da indústria para tratar de padrões relacionados a OO) foi convencido por vários vendedores de ferramenta que um padrão aberto era necessário. Assim, o processo abriu-se e uma força tarefa da OMG chefiada por Mary Loomis and Jim Odell organizaram o esforço inicial que levou à UML 1.0 em 1997. Muitos outros contribuíram para a UML, e talvez a contribuição mais notável tenha sido a de Cris Kobryn, um dos líderes do seu contínuo refinamento.

A UML emergiu como notação de diagramação padrão de fato e de direito para a modelagem orientada a objetos e tem continuado a ser refinada em novas versões UML OMG disponíveis em: www.omg.org ou www.uml.org.

² Kay começou a trabalhar em OO e PC na década de 1960, quando era estudante de graduação. Em dezembro de 1979 – incitado pelo grande Jef Raskin da Apple (o líder da criação do Mac) – Steve Jobs, co-fundador e principal executivo da Apple, visitou Alan Kay e equipes de pesquisa (inclusive Dan Ingalls, o implementador da visão de Kay) na Xerox PARC para ver uma demonstração do computador pessoal Smalltalk. Impressionado pelo que viu – uma interface gráfica do usuário com janelas superpostas de mapas de bits, programação OO e PCs em rede – ele voltou à Apple com uma nova visão (aquele que Raskin desejava) e os computadores Lisa e Macintosh da Apple nasceram.

1.9 Leituras recomendadas

Vários textos sobre A/POO são recomendados nos capítulos posteriores para assuntos específicos, tal como projeto OO. Os livros mencionados na seção histórico são todos valiosos para estudo – e ainda aplicáveis no que diz respeito ao seu assunto principal.

Um resumo legível e popular da notação UML essencial, *UML Distilled*, por Martin Fowler, é altamente recomendado: Fowler tem escrito muitos livros úteis com uma atitude prática e “ágil”.

Para uma discussão detalhada da notação UML, o *The Unified Modeling Language Reference Manual*, por Rumbaugh, vale a pena. Note que esse texto não tem por objetivo o aprendizado de como fazer a modelagem de objetos ou a A/POO – ele é uma referência para a notação de diagramas da UML.

Para a descrição definitiva da versão em uso da UML, veja as páginas online UML Infrastructure Specification e UML Superstructure Specification em www.uml.org ou www.omg.org.

A modelagem visual UML, em um espírito de modelagem ágil, é descrita em *Agile Modeling* por Scott Ambler. Veja também www.agilemodeling.com.

Há uma grande coleção de links para métodos de A/POO em www.cetus-links.org e www.iturls.com (veja a grande subseção de *Software Engineering* em inglês em vez da seção chinesa).

Existem muitos livros sobre padrões de software, mas o clássico que inspirou a todos é *Design Patterns*⁴ de Gamma, Helm, Johnson e Vlissides. Este livro é uma leitura obrigatória para aqueles que estudam o projeto de objetos. Contudo, ele não é um texto introdutório e é melhor lê-lo após sentir-se confortável com os fundamentos do projeto e programação de objetos. Veja também www.hillside.net e www.iturls.com (a subseção *Software Engineering* em inglês) para links a muitos outros sites.

⁴ N. de R.T.: Publicado no Brasil pela Bookman Editora com o título *Padrões de Projeto*.

ITERATIVO, EVOLUTIVO E ÁGIL

Você deve usar o desenvolvimento iterativo apenas em projetos que você deseja que sejam bem sucedidos.
— Martin Fowler

Objetivos

- Fornecer motivação para o conteúdo e ordem dos capítulos subsequentes.
- Definir um processo iterativo e ágil.
- Definir conceitos fundamentais do Processo Unificado.

Introdução

O desenvolvimento iterativo representa a parte central sobre como a A/POO é melhor praticada, e é apresentado neste livro. As práticas ágeis, tal como a modelagem ágil, são chave para aplicar a UML de um modo efetivo. Este capítulo introduz esses assuntos e o Processo Unificado como um exemplo de método iterativo relativamente popular.

O que vem a seguir?

Apresentada a A/POO, este capítulo explora o desenvolvimento iterativo. O próximo introduz os estudos de caso que são evoluídos ao longo do livro, por meio de três iterações.



O desenvolvimento iterativo e evolutivo – em contraposição ao ciclo de vida sequencial ou “em cascata” – envolve a imediata programação e teste de um sistema parcial em ciclos repetidos. Também considera normalmente que o desenvolvimento começa antes que os requisitos tenham sido definidos em detalhe; a realimentação é usada para esclarecer e aperfeiçoar as especificações em evolução.

Nós confiamos em passos de desenvolvimento pequenos e rápidos, realimentação e adaptação para esclarecer os requisitos e o projeto. Em contraposição, os valores do modelo em cascata promovem passos grandes de requisitos e de projeto especulativos logo no início, antes da programação. Estudos de sucesso/falha mostram consistentemente que o ciclo em cascata é fortemente associado às maiores taxas de falha em projetos de software e foi promovido historicamente por causa de crenças ou boatos, ao invés de evidência estatisticamente significativa. A pesquisa demonstra que métodos iterativos são associados a maiores taxas de sucesso e produtividade e menores níveis de defeito.

2.1. O que é PU? Os outros métodos são complementares?

Um processo de desenvolvimento de software descreve uma abordagem para a construção, implantação e, possivelmente, a manutenção de software. O **Processo Unificado (PU)** [JBR99] surgiu como um processo *iterativo* popular para o desenvolvimento de software visando à construção de sistemas orientados a objetos. Em particular, o **Processo Unificado da Rational** ou **RUP** (de Rational Unified Process), [Kruchten00], um refinamento detalhado do PU, é muito adotado.

Como o Processo Unificado (PU) é um processo iterativo relativamente popular para projetos que usam a A/POO e como algum processo precisa ser usado para introduzir o assunto, o PU dá forma à estrutura do livro. Além disso, como o PU é comum e promove práticas amplamente reconhecidas como melhores, é útil para profissionais da indústria conhecê-lo e para que estudantes que estão entrando no mercado de trabalho o conheçam também.

Desenvolvimento guiado por teste e refatoração (pág. 395)

O PU é muito flexível e aberto e incentiva a inclusão de práticas interessantes de outros métodos iterativos tais como: **eXtreme Programming (XP)**, **Scrum** e assim por diante. Por exemplo, as práticas de XP de **desenvolvimento guiado por testes, refatoração e integração contínua** podem se encaixar em um projeto de PU. O mesmo se pode dizer da sala comum de projeto (war room) do Scrum e da prática de reunião diária do Scrum. A introdução do PU não visa diminuir o valor desses outros métodos – muito pelo contrário. Em meu trabalho de consultoria, encorajo os clientes a entender e adotar um misto de técnicas úteis de diversos métodos, em vez de uma mentalidade dogmática de “meu método é melhor do que o seu método”.

O PU combina as melhores práticas comumente aceitas, como ciclo de vida iterativo e desenvolvimento guiado por risco, em uma descrição de processo coesa e bem documentada.

Resumindo, este capítulo inclui uma introdução ao PU por três motivos:

1. O PU é um processo iterativo. O desenvolvimento iterativo influencia como este livro introduz a A/POO e como ela é melhor praticada.
2. As práticas do PU fornecem uma *estrutura-exemplo* sobre como realizar – e, portanto, como explicar – a A/POO. Essa estrutura molda a estrutura do livro.
3. O PU é flexível e pode ser aplicado em uma abordagem leve e *ágil* que inclui práticas de outros métodos ágeis (tais como XP e Scrum) – voltaremos a este ponto posteriormente.

Este livro apresenta uma introdução a uma abordagem ágil do PU, mas não uma cobertura completa. Ele destaca idéias e artefatos comuns, próprias de uma introdução à A/POO e à análise de requisitos.

E se o PU não me interessar?

O PU é usado como um processo-*exemplo* dentro do qual são exploradas a análise de requisitos e a A/POO iterativa e evolutiva, já que é necessário tratar do assunto no contexto de algum processo.

Mas as idéias centrais deste livro – como pensar e projetar em termos de objetos, aplicar UML, usar padrões de projeto, modelagem ágil, análise de requisitos evolutiva, escrever casos de uso, etc. – são independentes de qualquer processo particular e se aplicam a muitos métodos modernos iterativos, evolutivos e ágeis, como Scrum, Lean Development, DSDM, Feature-Driven Development, Adpative Software Development e outros.

2.2 O que é desenvolvimento iterativo e evolutivo?

Uma prática-chave tanto no PU quanto na maioria de outros métodos modernos é o **desenvolvimento iterativo**. Nesta abordagem do ciclo de vida, o desenvolvimento é organizado em uma série de miniprojetos curtos, de duração fixa (por exemplo, três semanas) chamados **iterações**; o produto de cada um é um sistema parcial, executável, testável e integrável. Cada iteração inclui suas próprias atividades de análise de requisitos, projeto, implementação e teste.

O ciclo de vida iterativo é baseado em refinamentos e incrementos sucessivos de um sistema por meio de múltiplas iterações, com realimentação (feedback) e adaptação cíclicas como principais propulsores para convergir para um sistema adequado. O sistema cresce incrementalmente ao longo do tempo, iteração por iteração, razão pela qual esta abordagem também é conhecida como **desenvolvimento iterativo e incremental** (ver Figura 2.1). Como a realimentação e adaptação fazem as especificações e o projeto evoluir, esse sistema também é conhecido como **desenvolvimento iterativo e evolutivo**.

Inicialmente, as idéias de processos iterativos eram conhecidas como desenvolvimento evolutivo e em espiral [Boehm88, Gilb88].

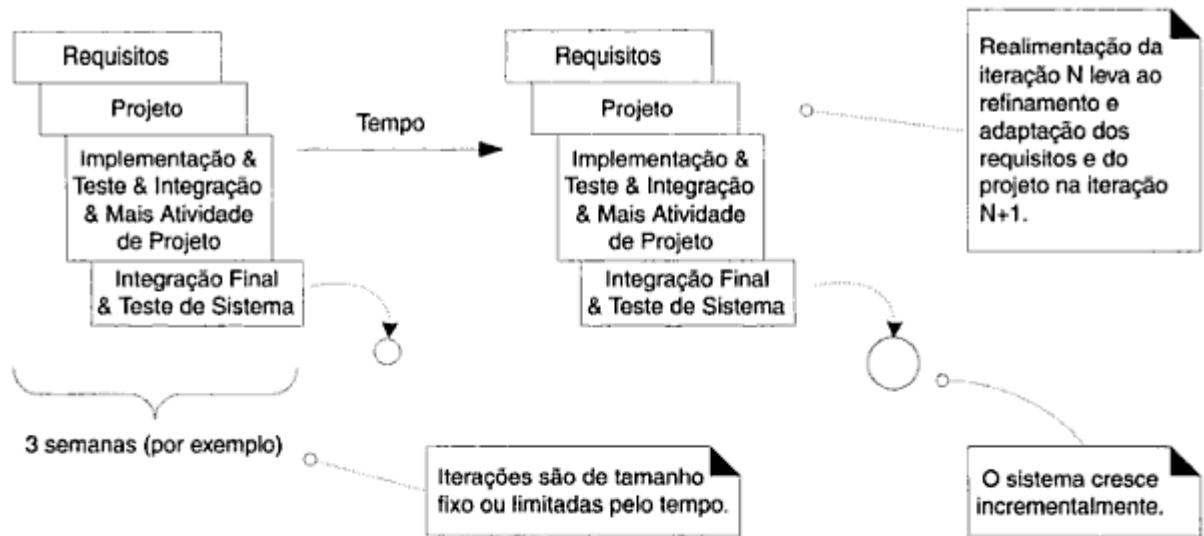


Figura 2.1 Desenvolvimento iterativo e evolutivo.

Exemplo

Como um exemplo (e não uma receita), em uma iteração de três semanas no início do projeto, talvez uma hora na segunda-feira de manhã seja gasta em uma reunião inicial com a equipe esclarecendo as tarefas e metas da iteração. Enquanto isso, uma pessoa faz a engenharia reversa do código da última iteração para diagramas UML (via uma ferramenta CASE) e imprime e exibe diagramas que sejam de interesse. A equipe gasta o resto da segunda-feira em quadros-brancos, trabalhando aos pares, fazendo modelagem ágil, rascunho de diagramas UML registrados em câmeras digitais e escrevendo pseudocódigo e notas de projeto. Os dias restantes são gastos em implementação, testes (unitários, de aceitação, de usabilidade,...), projeto adicional, integração, produção diária do sistema parcial. Outras atividades incluem demonstrações e avaliações para envolvidos no projeto, além do planejamento para a iteração seguinte.

Observe que, neste exemplo, não há uma corrida para codificar, nem um longo e demorado passo de projeto, que tenta aperfeiçoar todos os detalhes do projeto antes da programação. Algumas considerações “antecipadas” a respeito do projeto são feitas usando modelagem visual rápida e direta para esboço de diagramas UML, durante talvez meio ou um dia todo, por desenvolvedores trabalhando aos pares em quadros-brancos.

O resultado de cada iteração é um sistema executável, mas incompleto; ele não está pronto para ser colocado em produção. O sistema pode não estar preparado para instalação e produção senão após muitas iterações, como, por exemplo, 10 ou 15 iterações.

A saída de uma iteração *não* é um protótipo experimental ou descartável, assim como o desenvolvimento iterativo não é *prototipação*. Ao contrário, a saída é um subconjunto do sistema final com qualidade final de produção.

Como acolher mudanças em um projeto iterativo?

O subtítulo de um livro que discute o desenvolvimento iterativo é *Embrace Change* (Acolha a Mudança) [Beck00]. Esta frase evoca a atitude-chave do desenvolvimento iterativo: em vez de combater a inevitável mudança que ocorre no desenvolvimento de software tentando (geralmente sem sucesso) especificá-lo completa e corretamente e “fazer com que todos assinem embaixo” um conjunto de requisitos e projetos congelados antes da implementação (em um processo em “cascata”), o desenvolvimento iterativo e evolutivo é baseado em uma atitude de aceitar a mudança e a adaptação como fatores inevitáveis e, de fato, essenciais.

Isso não significa que o desenvolvimento iterativo e o PU encorajem um processo descontrolado e reativo “que se arrasta por ser excessivamente guiado por características (features)”. Os capítulos subsequentes mostram como o PU equilibra a necessidade – por um lado – de chegar a um acordo e estabilizar um conjunto de requisitos com – por outro lado – a realidade de mudança de requisitos, à medida que os interessados no projeto do sistema esclarecem sua visão ou ocorrem mudanças no mercado de atuação da organização.

Cada iteração exige a escolha de um pequeno subconjunto dos requisitos, além da sua rápida projeção, implementação e teste. Nas iterações iniciais, a escolha de requisitos e o projeto podem não ser exatamente o que é desejado em última instância. No entanto, o ato de executar rapidamente um pequeno passo antes de finalizar todos os requisitos ou, antes, que o projeto inteiro seja especulativamente definido, leva a uma realimentação rápida – obtida a partir de usuários, desenvolvedores e testes (como testes de carga e de facilidade de uso).

Essa realimentação precoce vale ouro; em vez de *especular* sobre os requisitos ou projeto corretos e completos, a equipe procura a realimentação a partir de uma construção e teste realístico de alguma coisa buscando uma percepção prática crucial, bem como uma oportunidade para modificar ou adaptar a compreensão dos requisitos ou do projeto. Os usuários finais têm a oportunidade de ver um sistema parcial e dizer: “sim, foi isso que eu pedi, mas agora que o experimentei, o que eu realmente quero é algo ligeiramente diferente!”¹. Esse processo de “sim... mas” não é um sinal de erro; na verdade, ciclos estruturados precoces e freqüentes de “sim... mas” são um modo hábil de progredir e descobrir o que é de real valor para os interessados no sistema. No entanto, isso não é um endosso do desenvolvimento caótico e reativo, no qual os desenvolvedores continuamente mudam de direção – é possível um meio-termo.

Além de esclarecer requisitos, atividades como teste de carga provarão se o projeto e a implementação parciais estão no caminho certo ou se, na iteração seguinte, será necessária uma mudança na arquitetura central do sistema. É melhor resolver e por à prova as decisões arriscadas e fundamentais de projeto precocemente do que tardivamente – e o desenvolvimento iterativo fornece o mecanismo para isso.

Conseqüentemente, o trabalho progride por meio de uma série de ciclos estruturados em construção-realimentação-adaptação. Não é surpreendente que nas iterações iniciais o desvio do “caminho verdadeiro” do sistema (em termos dos seus requisitos

¹ Ou, mais provável, “Você não compreendeu o que eu queria!”

Iterações iniciais estão longe do "verdadeiro caminho" do sistema. Por meio de realimentação e adaptação, o sistema converge para os requisitos e o projeto mais adequados.

Nas últimas iterações, é rara uma mudança significativa de requisitos, mas ela pode ocorrer. Essas mudanças tardias podem dar à organização uma vantagem competitiva nos negócios.



Figura 2.2 A realimentação e a evolução iterativas levam ao sistema desejado. A instabilidade dos requisitos e do projeto diminue com o tempo.

e projeto finais) seja maior do que nas últimas iterações. Com o tempo, o sistema converge para esse caminho, como ilustrado na Figura 2.2.

Existem benefícios no desenvolvimento iterativo?

Sim. Os benefícios incluem:

- Menos erros de projeto, maior produtividade e menor taxa de defeitos; mostrados pela pesquisa nos métodos iterativos e evolutivos
- Mitigação precoce, em vez de tardia, de altos riscos (técnicos, requisitos, objetivos, usabilidade, etc.)
- Progresso visível desde o início
- Realimentação, envolvimento do usuário e adaptação imediatos, levando a um sistema refinado que atenda, de forma mais adequada, às reais necessidades dos interessados no projeto
- A complexidade é administrada; a equipe não é sobrecarregada pela "paralisia da análise" ou por passos muito longos e complexos
- O aprendizado obtido em uma iteração pode ser metódicamente usado para melhorar o próprio processo de desenvolvimento, iteração por iteração

Quão longa deve ser uma iteração? O que são limites temporais?

A maioria dos métodos iterativos recomenda que a duração de uma iteração seja entre duas e seis semanas. Usar pequenos passos, obter realimentação rápida e fazer adaptações são idéias centrais no desenvolvimento iterativo; iterações longas subvertem a motivação central para o desenvolvimento iterativo e aumentam o risco do projeto. Em apenas uma semana freqüentemente é difícil completar trabalho suficiente para obter resultado e realimentação significativos; mais do que seis semanas fazem com que a complexidade seja avassaladora e a realimentação seja retardada.

Uma iteração muito longa perde o objetivo do desenvolvimento iterativo. Um período curto é melhor.

Uma idéia-chave é que as iterações têm **limites temporais** (ocupam “janelas de tempo” de duração fixa no cronograma). Por exemplo, se é decidido que a iteração seguinte deve ter três semanas de duração, então o sistema parcial *deve* estar integrado, testado e estabilizado dentro da data programada – o não cumprimento dos prazos é ilegal. Se acharmos que será difícil cumprir o prazo final, é recomendável remover tarefas ou requisitos da iteração e incluí-las em uma iteração futura, em vez de não cumprir o prazo.

2.3 O que dizer do ciclo de vida em cascata?

Em um processo de ciclo de vida em **cascata** (ou seqüencial) há uma tentativa de definir (em detalhe) todos ou a maioria dos requisitos antes da programação. Freqüentemente, também de criar um projeto abrangente (ou conjunto de modelos) antes da programação. Igualmente, há uma tentativa de definir um plano ou cronograma “confiável” logo no começo – não que será assim.

Atenção: superposição do “em Cascata” com o iterativo

Se você estiver em um projeto “iterativo” em que a maioria dos requisitos é escrita antes que o desenvolvimento comece ou em que há tentativa de criar especificações muito abrangentes e detalhadas ou modelos e projetos UML antes da programação, saiba que o raciocínio em cascata infelizmente invadiu o projeto. Não é um projeto saudavelmente iterativo ou PU, independentemente das alegações.

Pesquisas (coletadas de muitas fontes e resumidas em [Larman03] e [LB03]) mostram agora, conclusivamente, que a recomendação na era de 1960 e 1970 de aplicar o ciclo em cascata era – ironicamente – uma má prática para a maioria dos projetos de software, em vez de ser uma abordagem racional. Ela está fortemente associada a altas taxas de falhas, menor produtividade e maiores taxas de defeitos (do que projetos iterativos). Na média, 45% das características nos requisitos em cascata nunca são usadas e os cronogramas e orçamentos iniciais, em cascata, variam até 400% do que ocorre na realidade.

Em retrospecto, sabe-se agora que a recomendação em cascata era baseada em *especulação e boato*, em vez de em práticas baseadas em evidência. Em contraposição, práticas iterativas e evolutivas são apoiadas por evidência – estudos mostram que elas são menos propensas a falhas e associadas com melhor produtividade e menor taxa de defeitos.

Diretriz: não deixe o raciocínio em cascata invadir um projeto iterativo ou PU

É necessário enfatizar que o “racioncínio em cascata” freqüentemente ainda invade, incorretamente, um projeto chamado de iterativo ou PU. Idéias tais como “vamos es-

“crever todos os casos de uso antes de começar a programar” ou “vamos fazer muitos modelos OO detalhados em UML antes de começar a programar” são exemplos de raciocínio em cascata doentio, incorretamente superposto no PU. Os criadores do PU citam esse mal entendido – vasta análise e modelagem antecipada – como uma razão-chave para a sua adoção falha [KL01].

Por que o ciclo em cascata é tão propenso à falha?

Não há uma resposta simples para o porquê do ciclo em cascata ser tão propenso à falha, mas isso está fortemente relacionado à falsa suposição-chave subjacente a muitos projetos falhos de software – de que as especificações são previsíveis e estáveis e podem ser corretamente definidas no início, com baixas taxas de modificação. Isso na verdade está longe de ser preciso e é um mal-entendido dispendioso. Um estudo feito por Boehm e Papaccio mostrou que um típico projeto de software sofre 25% de modificações nos requisitos [BP88]. Essa tendência foi corroborada em outro importante estudo de milhares de projetos de software com taxas de modificações que vão ainda mais alto – 35% a 50% para grandes projetos – como ilustrado na Figura 2.3 [Jones97].

Essas são taxas de modificação *extremamente* altas. O que esses dados mostram – como qualquer desenvolvedor ou gerente experiente está lamentavelmente ciente – é que o desenvolvimento de software (em média) é um domínio de alta modificação e instabilidade – também conhecido como o domínio de **desenvolvimento de produto novo**. Software não é usualmente um domínio de fabricação previsível ou em massa – áreas de baixa modificação em que é possível e eficiente definir todas as especificações estáveis e planos confiáveis perto do início.

Assim, qualquer prática de análise, modelagem, desenvolvimento ou gestão baseada na suposição de que as coisas são estáveis a longo prazo (em cascata) é fundamentalmente falha. *Modificação* é uma constante em projetos de software. Métodos iterativos e evolutivos assumem e acolhem modificação e adaptação de especificações, modelos e planos *parciais* e *evolutivos* baseados em realimentação.

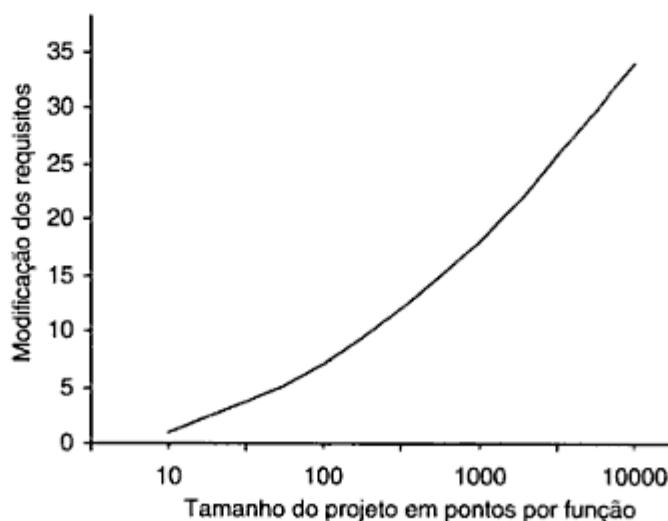


Figura 2.3 Percentagem de modificação em projetos de software de diferentes tamanhos.

A necessidade de realimentação e adaptação

Em sistemas complexos e mutáveis (tais como a maioria dos projetos de software), realimentação e adaptação são ingredientes-chave para o sucesso.

- Realimentação oriunda do início do desenvolvimento, dos programadores tentando ler especificações e das demonstrações ao cliente para refinar os requisitos.
- Realimentação oriunda dos testes e dos desenvolvedores para refinar o projeto ou modelos.
- Realimentação oriunda do progresso da equipe em detectar as características iniciais para refinar o cronograma e orçamento.
- Realimentação oriunda do cliente e do mercado para re-priorizar as características a atacar na iteração seguinte.

2.4. Como fazer análise e projeto iterativos e evolutivos?

Esta introdução pode ter dado a impressão de que de nada vale a análise e projeto antes da programação, mas esse é um mal-entendido tão extremo como pensar que uma análise antecipada “completa” é racional. Há um meio termo. Segue um pequeno *exemplo* (não uma receita) de como isso pode funcionar em um projeto PU bem conduzido. Considera-se que haverá, em última instância, 20 iterações no projeto antes da entrega:

1. Antes da iteração 1, realizar a primeira reunião sobre os requisitos com tempo limitado, por exemplo, dois dias. O pessoal de negócio e de desenvolvimento (incluindo o arquiteto chefe) está presente.
 - Na manhã do primeiro dia, fazer a análise de requisitos de alto nível como, por exemplo, identificar apenas os nomes dos casos de usos e características e requisitos-chave não funcionais. A análise não ficará perfeita.
 - Pedir ao arquiteto chefe ou ao pessoal de negócio para escolher 10% dessa lista de alto nível (por exemplo, 10% dos 30 nomes de casos de uso) que tenha uma mistura de três das seguintes qualidades: 1) arquiteturalmente significativo (se implementado, fica-se forçado a projetar, construir e testar essa arquitetura central), 2) alto valor de negócio (características com as quais o negócio realmente se preocupa) e 3) alto risco (tal como “poder manipular 500 transações concorrentes”). Talvez três casos de uso sejam assim identificados: CDU2, CDU11 e CDU14.
 - Para os restantes 1,5 dias, fazer análise detalhada intensiva dos requisitos funcionais e não funcionais desses três casos de uso. Ao final, 10% estão profundamente analisados e 90% apenas em alto nível.
2. Antes da iteração 1, realizar uma reunião de planejamento da iteração na qual um subconjunto de CDU2, CDU11 e CDU14 é escolhido para projetar, construir e testar dentro de um tempo especificado (por exemplo, iteração com tempo limitado de quatro semanas). Note que esses três casos de uso não precisam ser

todos construídos na iteração 1, pois eles vão dar muito trabalho. Depois de escolher o subconjunto específico de metas, divida-as em um conjunto de tarefas de iteração mais detalhadas, com ajuda da equipe de desenvolvimento.

3. Realizar a iteração 1 ao longo de três ou quatro semanas (escolher a limitação de tempo e respeitá-la).
 - Nos primeiros dois dias, desenvolvedores e demais pessoas fazem trabalho de modelagem e projeto aos pares, rascunhando diagramas tipo UML em vários quadros-brancos (ao mesmo tempo em que se rascunham outros tipos de modelos) em uma sala comum, treinados e dirigidos pelo arquiteto chefe.
 - Depois os desenvolvedores tiram seus “chapéus de modelagem” e colocam seus “chapéus de programação”. Eles começam a programar, testar e integrar seu trabalho continuamente ao longo das semanas restantes, usando os rascunhos de modelagem como ponto de partida da inspiração, sabendo que os modelos são parciais e freqüentemente vagos.
 - Vários testes ocorrem: de unidade, de aceitação, de carga, de usabilidade, etc.
 - Uma semana antes do fim, pergunta-se à equipe se as metas originais da iteração podem ser alcançadas; em caso negativo, diminuir o escopo da iteração, colocando as metas secundárias de volta na lista “a fazer”.
 - Na terça-feira da última semana há um congelamento do código; todo o código precisa ser entregue, integrado e testado para criar a linha de referência da iteração.
 - Na manhã de quarta-feira demonstrar o sistema parcial aos interessados externos para mostrar o progresso visível inicial. Realimentação é solicitada.
4. Realizar a segunda reunião de requisitos perto do fim da iteração 1 como, por exemplo, na última quarta e quinta-feira. Rever e refinar todo o material da reunião anterior. Escolher então outros 10 ou 15% dos casos de uso que são arquiteturalmente significativos e de alto valor de negócio e analisá-los em detalhe durante um ou dois dias. Ao final, talvez 25% dos casos de uso e dos requisitos não funcionais vão estar escritos em detalhe. Eles não vão estar perfeitos.
5. Na manhã de sexta-feira realizar outra reunião de planejamento de iteração para a iteração seguinte.
6. Realizar a iteração 2; passos semelhantes.
7. Repetir para quatro iterações e cinco reuniões de requisitos de modo que, ao final da iteração 4, talvez 80 ou 90% dos requisitos estejam escritos em detalhe, mas apenas 10% do sistema tenha sido implementado.
 - Note que esse grande conjunto de requisitos detalhados é baseado em realimentação e evolução e é, assim, de muito mais alta qualidade que especificações em cascata puramente especulativas.
8. Estamos talvez avançados em apenas 20% na duração do projeto global. Em termos de PU, esse é o fim da **fase de elaboração**. Nesse ponto, estimar em detalhe o esforço e tempo para requisitos refinados de alta qualidade. Em razão da in-

vestigação realística significativa, da realimentação e da programação e testes antecipados, as estimativas do que pode ser feito e quanto tempo vai levar são muito mais confiáveis.

9. Além desse ponto, reuniões de trabalho são improváveis; os requisitos estão estabilizados – apesar de nunca completamente congelados. Continuar em uma série de iterações de três semanas, escolhendo o próximo passo de trabalho, adaptativamente em cada reunião de planejamento de iteração na sexta-feira final, repetindo a questão a cada iteração, “considerando o que sabemos hoje, quais são as características mais críticas em termos técnicos e do negócio que nós deveríamos atender nas próximas três semanas?”

A Figura 2.4 ilustra a abordagem para um projeto de vinte iterações.

Desse modo, depois de algumas iterações do desenvolvimento exploratório inicial, chega-se a um ponto em que a equipe pode responder mais confiavelmente “o que, quanto, quando”.

2.5 O que é planejamento iterativo guiado por risco e guiado pelo cliente?

O PU (e a maioria dos novos métodos) encoraja uma combinação de planejamento iterativo **guiado por risco** e **guiado por cliente**. Isso significa que as metas das iterações iniciais são escolhidas para: 1) identificar e controlar os maiores riscos e 2) construir características visíveis com as quais o cliente mais se preocupa.

Desenvolvimento iterativo guiado por risco inclui, mais especificamente, a prática de desenvolvimento iterativo **centrado na arquitetura**, significando que as iterações iniciais focalizam a construção, o teste e a estabilização da arquitetura central. Por que? Porque não dispor de uma arquitetura sólida é um risco alto e comum.

Iterações do livro vs. Iterações do projeto real

A iteração 1 dos estudos de caso deste livro é guiada por metas de aprendizado em vez de metas reais de projeto. Assim, a iteração 1 não é centrada na arquitetura ou guiada por riscos. Em um projeto real, iríamos detectar as coisas difíceis e arriscadas primeiro. Mas, no contexto de um livro ajudando as pessoas a entenderem fundamentos de A/POO e UML, isso não é prático – precisamos começar com problemas que ilustram princípios básicos e não tópicos e problemas mais difíceis.

2.6 O que são métodos e atitudes ágeis?

Métodos de **desenvolvimento ágil** usualmente aplicam desenvolvimento iterativo e evolutivo de tempo limitado, empregam planejamento adaptativo, promovem entrega incremental e incluem outros valores e práticas que encorajam *agilidade* – resposta rápida e flexível à modificação.

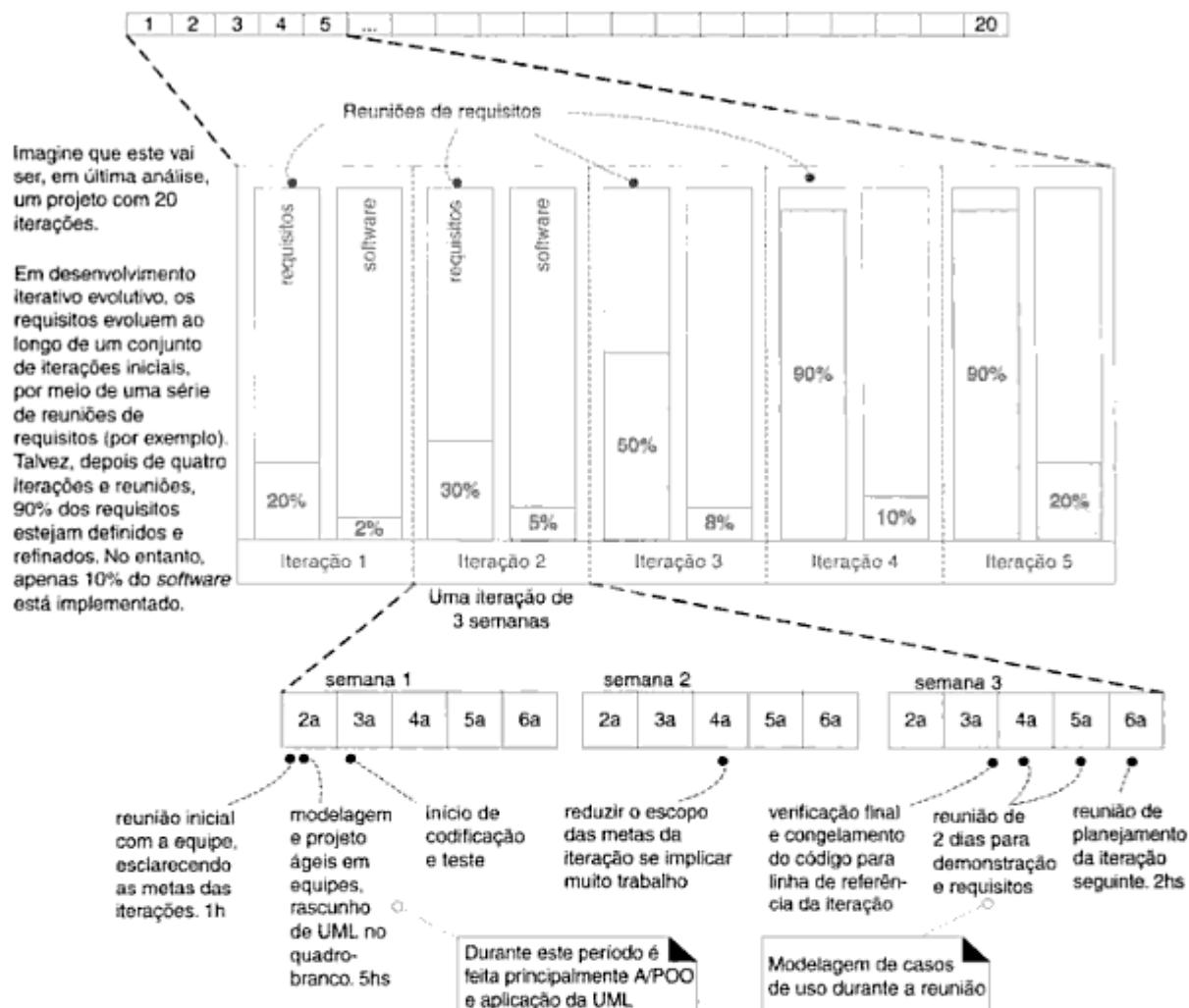


Figura 2.4 Análise e projeto evolutivos – a maioria nas primeiras iterações.

Não é possível definir exatamente **métodos ágeis**, pois as práticas específicas variam muito. No entanto, iterações curtas de tempo limitado com refinamento evolutivo de planos, requisitos e projeto é uma prática básica que os métodos compartilham. Além disso, eles promovem práticas e princípios que refletem uma sensibilidade ágil de simplicidade, leveza, comunicação, equipes auto-organizadas, entre outras.

Exemplos de práticas do método ágil Scrum incluem *uma sala de trabalho de projeto comum* e *equipes auto-organizadas* que se coordenam por meio de uma reunião diária, em pé, com quatro questões especiais que cada membro responde. Exemplos de práticas do método Extreme Programming (XP) incluem *programação em pares* e *desenvolvimento guiado por teste*.

DGT
(pág. 395)

Qualquer método iterativo, inclusive o PU, pode ser aplicado com espírito ágil. O próprio PU é flexível, encorajando uma atitude de “o que quer que funcione” para incluir práticas do Scrum, XP e outros métodos.

O manifesto ágil e os princípios ágeis

O Manifesto Ágil

<i>Indivíduos e iterações</i>	<i>vem antes de processos e ferramentas</i>
<i>Software funcionando</i>	<i>vem antes de documentação abrangente</i>
<i>Colaboração do cliente</i>	<i>vem antes de negociação de contrato</i>
<i>Resposta à modificação</i>	<i>vem antes de um plano em andamento</i>

Os Princípios Ágeis

1. Nossa prioridade mais alta é satisfazer o cliente por meio de entrega pronta e contínua de software de valor.
2. Acolher modificação de requisitos, mesmo no final do desenvolvimento. Processos ágeis valorizam a modificação para vantagem competitiva do cliente.
3. Entregar software funcionando com freqüência (de várias semanas a vários meses), preferencialmente usando uma escala de tempo menor.
4. O pessoal do negócio e os desenvolvedores devem trabalhar juntos diariamente ao longo do projeto.
5. Construir projetos em volta de indivíduos motivados. Dê a eles o ambiente e o apoio que necessitam e confie que eles vão fazer o serviço.
6. O método mais eficiente e efetivo para levar informação de e para uma equipe de desenvolvimento é a conversa face a face.
7. Software funcionando é a principal medida de progresso.
8. Processos ágeis promovem desenvolvimento sustentável.
9. Os patrocinadores, desenvolvedores e usuários devem poder manter um ritmo constante indefinidamente.
10. Atenção contínua para a excelência técnica e para um bom projeto aumenta a agilidade.
11. Simplicidade – a arte de maximizar a quantidade de trabalho não realizada – é essencial.
12. As melhores arquiteturas, requisitos e projetos surgem de equipes auto-organizadas.
13. Em intervalos regulares, a equipe reflete sobre como se tornar mais efetiva, depois sintoniza e ajusta o seu comportamento de acordo com isso.

Em 2001 um grupo interessado em métodos iterativos e ágeis (cunhando o termo) reuniu-se para descobrir pontos comuns. Daí surgiu a Aliança Ágil (Agile Alliance) (www.agilealliance.com) com um manifesto e uma declaração de princípios para captar o espírito dos métodos ágeis.

2.7 O que é modelagem ágil?

Mais sobre modelagem ágil (págs. 235-236) Analistas e modeladores experientes conhecem o *segredo da modelagem*:

A finalidade da modelagem (rascunhos em UML,...) é principalmente *entender*, não documentar.

Isto é, o próprio ato de modelar pode e deve fornecer um modo de melhor entender o problema ou o espaço de solução. Desse ponto de vista, a verdadeira finalidade de “fazer UML” (que deveria na realidade significar “fazer A/POO”) *não* é de um projetista criar muitos diagramas UML detalhados que são entregues a um programador

(um raciocínio não muito ágil e orientado em cascata), mas em vez disso rapidamente explorar (mais rapidamente do que com código) alternativas e o caminho para um bom projeto OO.

Essa visão consistente com métodos ágeis tem sido chamada **modelagem ágil** no livro (surpreendentemente chamado) *Modelagem Ágil* [Ambler02]. Isso implica uma quantidade de prática e valores, incluindo:

- A adoção de um método ágil não significa evitar a modelagem; esse é um mal-entendido. Muitos métodos ágeis, como desenvolvimento guiado por características (Feature Driven Development), DSDM e Scrum, normalmente incluem significativas sessões de modelagem. Mesmo os criadores do XP, talvez o mais conhecido método ágil com menor ênfase em modelagem, endossam a modelagem ágil tal como descrita por Ambler – e praticada por muitos modeladores ao longo dos anos.
- A finalidade da modelagem e dos modelos é, principalmente, apoiar o entendimento e a comunicação, não a documentação.
- Não modele ou aplique UML a todo ou à maioria do projeto de software. Adie problemas simples ou diretos de projeto até a programação – resolva-os durante a programação e testes. Modele e aplique UML nas partes incomuns, difíceis e cheias de armadilhas do espaço de projeto, que representam uma percentagem menor em relação ao projeto.
- Use a ferramenta mais simples possível. Prefira ferramentas simples “de baixa energia” que incentivam a criatividade e apóiam a compreensão e a modificação rápidas. Escolha também ferramentas que apóiam amplos espaços visuais. Por exemplo, prefira rascunhar UML em quadros-branco e capte os diagramas com uma câmera digital.²
 - Isso não significa que ferramentas CASE UML ou processadores de texto não possam ser usados ou não tenham valor, mas especialmente para o trabalho criativo de descoberta, o rascunho em quadro-branco favorece um fluxo criativo e de modificação rápida. A regra-chave é facilidade e agilidade, qualquer que seja a tecnologia.
- Não modele sozinho, modele aos pares (ou trios) no quadro-branco, na certeza de que a finalidade da modelagem é descobrir, entender e compartilhar esse entendimento. Faça com que a caneta rode entre os membros de modo que todos participem.
- Crie modelos em paralelo. Por exemplo, em um quadro-branco comece a rascunhar uma visão dinâmica de um diagrama de interação UML e em outro comece a desenhar a visão estática complementar de um diagrama de classes UML. Desenvolva os dois modelos (duas visões) simultaneamente, comutando entre eles.

² Duas sugestões para rascunho em quadro-branco: 1) se você não tem quadros-brancos suficientes (e você deveria ter muitos, e grandes) uma alternativa é a folha de plástico aderente “quadro-branco”, que adere a paredes (com carga estática) para criar quadros-brancos. O principal produto na América do Norte é o Avery Write-On Cling Sheets; o principal produto na Europa é o LegaMaster Magic-Chart. 2) fotos digitais de imagens em quadro-branco são frequentemente de má qualidade (por causa do reflexo), por isso não use flash – use uma aplicação de software para limpeza de imagem em quadro-branco (“whiteboard image clean up”) para melhorar a imagem, se você precisar limpá-las (como fiz para este livro).

- Use uma notação simples e “suficientemente boa” para rascunhar a caneta em quadro-branco. Detalhes exatos de UML não são importantes, desde que os modeladores entendam-se entre si. Restrinja-se a elementos UML simples, freqüentemente usados.
- Saiba que todos os modelos vão ficar imprecisos e que o código ou projeto final ficará diferente – algumas vezes bastante diferente – do modelo. Apenas o código testado demonstra o verdadeiro projeto; todos os diagramas anteriores são indícios incompletos, que podem ser mais bem pensados como explorações descartáveis.
- Os próprios desenvolvedores deveriam fazer a modelagem do projeto OO para si próprios, não para criar diagramas que sejam dados a outros programadores para implementar – um exemplo de práticas não ágeis orientadas em cascata.

Modelagem ágil neste livro: por que fotografar os rascunhos UML?

A modelagem de rascunhos UML em quadros-brancos é uma prática que eu – e muitos desenvolvedores – temos recomendado e praticado entusiasticamente durante anos. No entanto, a maioria dos diagramas UML deste livro dá a impressão de que eu não trabalho desse modo, porque eles foram desenhados elegantemente com uma ferramenta, para maior legibilidade. Para contrabalançar essa impressão, o livro ocasionalmente inclui fotografias instantâneas digitais de rascunhos UML em quadro-branco. Isso sacrifica a legibilidade, mas lembra que a modelagem é útil e é a prática real usada nos estudos de caso.

Por exemplo, a Figura 2.5 é um rascunho UML não editado criado em um projeto que eu estava orientando. Levou 20 minutos para ser desenhado por mim com quatro desenvolvedores em volta. Precisamos entender a colaboração inter-sistemas. O ato de desenhá-lo juntos forneceu um contexto para contribuir com conhecimentos profundos de cada um e alcançar entendimento compartilhado. Isso capta o sentimento de como modeladores ágeis aplicam a UML.

2.8 O que é PU ágil?

O PU não era destinado por seus criadores a ser pesado ou não ágil, apesar de seu grande conjunto *opcional* de atividades e artefatos ter comprehensivelmente levado alguns a ter essa impressão. Ao contrário, estava destinado a ser adotado e aplicado no espírito de adaptabilidade e leveza – um PU ágil. Alguns exemplos de como isso se aplica são:

- Personalizar o PU (págs. 64-65)*
- Prefira um conjunto *pequeno* de atividades e artefatos do PU. Alguns projetos vão se beneficiar mais do que outros, mas em geral conserve a simplicidade. Lembre-se de que todos os artefatos do PU são opcionais e evite criá-los, a menos que eles adicionem valor. Dê ênfase à programação precoce e não à documentação precoce.
 - Como o PU é iterativo e evolutivo, os requisitos e projetos não são completados antes da implementação. Eles emergem adaptativamente por meio de uma série de iterações, com base em realimentação.
- A&P evolutivos (pág. 53)*

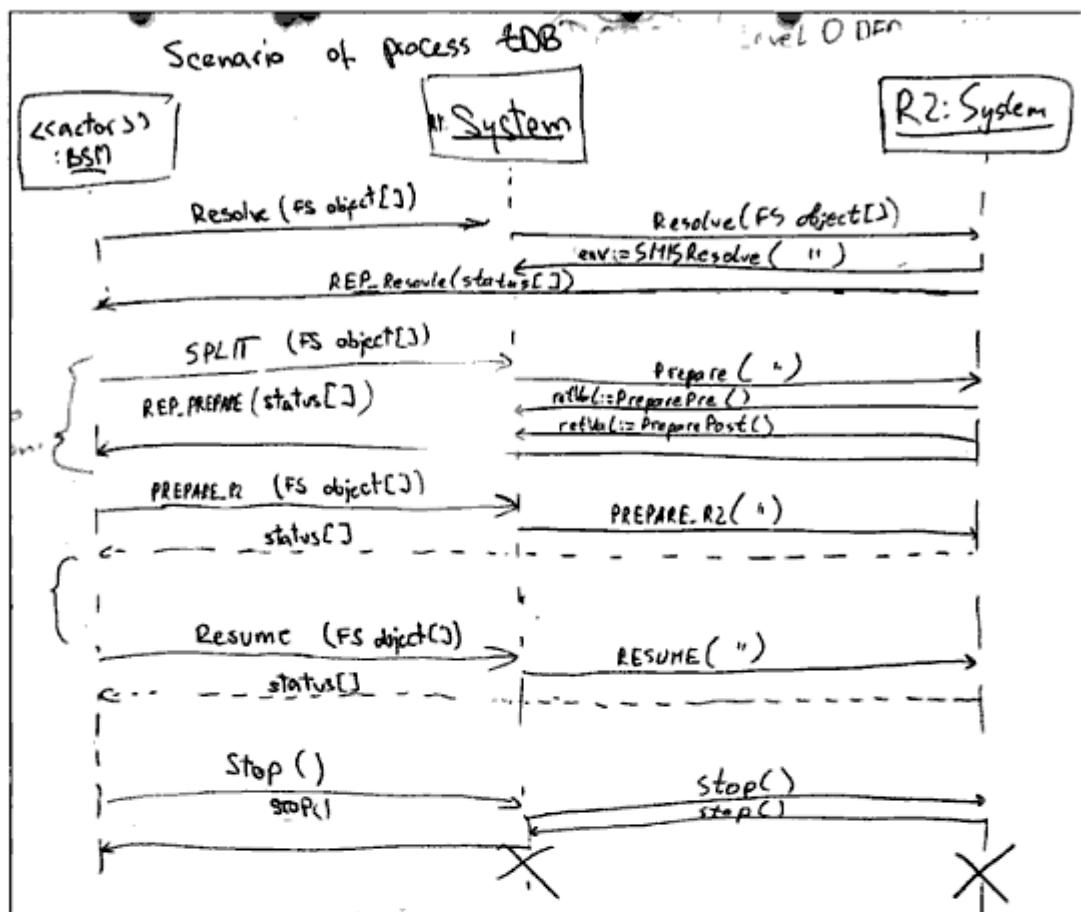


Figura 2.5 Um rascunho UML de um diagrama de seqüência para um projeto.

Modelos
ágeis
(págs. 57-58)

GP ágil
(págs. 665-
666)

- Aplique a UML com práticas de modelagem ágil.
- Não há um plano *detalhado* para o todo o projeto. Há um plano de alto nível (chamado **Plano de Fase**) que estima a data final do projeto e outros marcos importantes, mas não dá os passos detalhados para esses marcos. Um plano detalhado (chamado **Plano de Iteração**) planeja com mais detalhes apenas uma iteração futura. O planejamento detalhado é feito adaptativamente, iteração a iteração.

Os estudos de caso enfatizam um número de artefatos relativamente pequeno e o desenvolvimento iterativo, no espírito de um PU ágil.

2.9 Há outras práticas importantes de PU?

A idéia central a ser apreciada e praticada no PU é a de usar iterações curtas, com duração fixa, em um processo de desenvolvimento iterativo, evolutivo e adaptativo. Algumas das melhores práticas e conceitos-chave adicionais do PU incluem:

- enfrentar os problemas que envolvem altos riscos e alto valor já nas iterações iniciais;
- envolver continuamente os usuários na avaliação, na realimentação e nos requisitos;
- construir uma arquitetura central coesa nas iterações iniciais;
- verificar continuamente a qualidade; fazer testes logo de início, com freqüência e em situações realísticas;
- aplicar casos de uso quando adequado;
- modelar visualmente o software (com a UML);
- gerenciar requisitos cuidadosamente;
- por em prática o gerenciamento de solicitações de mudança e o gerenciamento de configuração.

2.10 Quais são as fases do PU?

Um projeto PU organiza o trabalho e as iterações em quatro fases principais:

1. **Concepção** – visão aproximada, casos de negócio, escopo e estimativas vagas.
2. **Elaboração** – visão refinada, implementação iterativa da arquitetura central, resolução dos altos riscos, identificação da maioria dos requisitos e do escopo e estimativas mais realistas.
3. **Construção** – implementação iterativa dos elementos restantes de menor risco e mais fáceis e preparação para a implantação.
4. **Transição** – testes beta e implantação.

Essas fases são mais bem definidas nos capítulos subsequentes.

Esse *não* é o antigo ciclo de vida em “cascata” ou seqüencial, no qual primeiro temos que definir todos os requisitos para, então, fazer todo ou a maioria do projeto.

A concepção não é uma fase de requisitos; na verdade, é uma espécie de fase de estudo de viabilidade, na qual se executa um volume suficiente de investigação para fundamentar a decisão de continuar ou parar.

Da mesma forma, a elaboração não é a fase de requisitos ou projeto; é uma fase na qual a arquitetura central é iterativamente implementada e os problemas de alto risco são mitigados.

A Figura 2.6 ilustra termos comuns relativos ao cronograma de um PU. Note que um ciclo de desenvolvimento (que termina com a entrega de uma versão do sistema para ser posta em produção) é composto de muitas iterações.

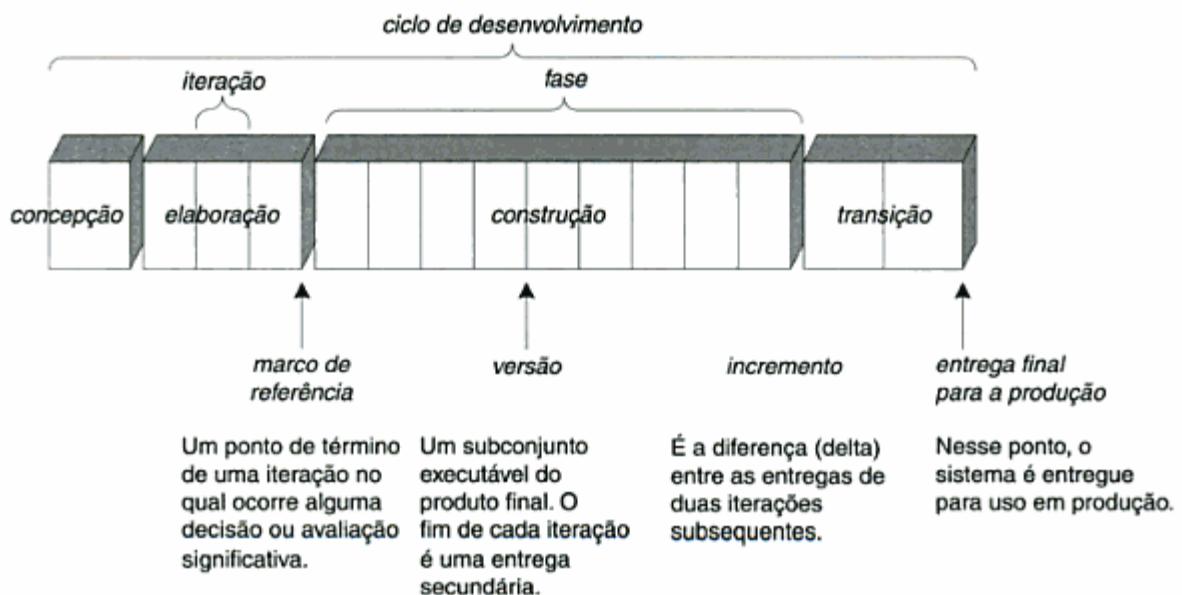


Figura 2.6 Termos relativos ao cronograma de um PU.

2.11 Quais são as disciplinas do PU?

O PU descreve atividades de trabalho, como redigir um caso de uso, dentro de **disciplinas** – conjunto de atividades (e artefatos relacionados) em uma área alvo, como atividades dentro da análise de requisitos. No PU, um **artefato** é o termo usado para qualquer produto de trabalho: código, gráficos para a Web, esquemas de bancos de dados, documentos em texto, diagramas, modelos e assim por diante. Existem várias disciplinas no PU; este livro focaliza alguns artefatos nas três disciplinas seguintes:

- **Modelagem de Negócio** – O artefato Modelo do Domínio, para visualizar conceitos importantes no domínio de aplicação.
- **Requisitos** – Os artefatos Modelo de Casos de Uso e Especificação Suplementar, para captar requisitos funcionais e não funcionais.
- **Projeto** – o artefato Modelo de Projeto para projetar os objetos de software.

Uma lista maior de disciplinas do PU é mostrada na Figura 2.7.

No PU, **Implementação** significa programação e construção do sistema, não a implantação. A disciplina **Ambiente** se refere ao estabelecimento do instrumental e à personalização do processo para um projeto – ou seja, a preparação das ferramentas e o ambiente do processo.

Qual é a relação entre disciplinas e fases?

Como ilustrado na Figura 2.7, durante uma iteração o trabalho prossegue na maioria ou em todas as disciplinas. Contudo, o esforço relativo no decorrer destas disciplinas muda ao longo do tempo. As iterações iniciais naturalmente tendem a dar uma ênfase maior aos requisitos e ao projeto, enquanto que as últimas disciplinas dão a esses itens uma ênfase menor, à medida que os requisitos e o projeto central se estabilizam por meio de um processo de realimentação e adaptação.

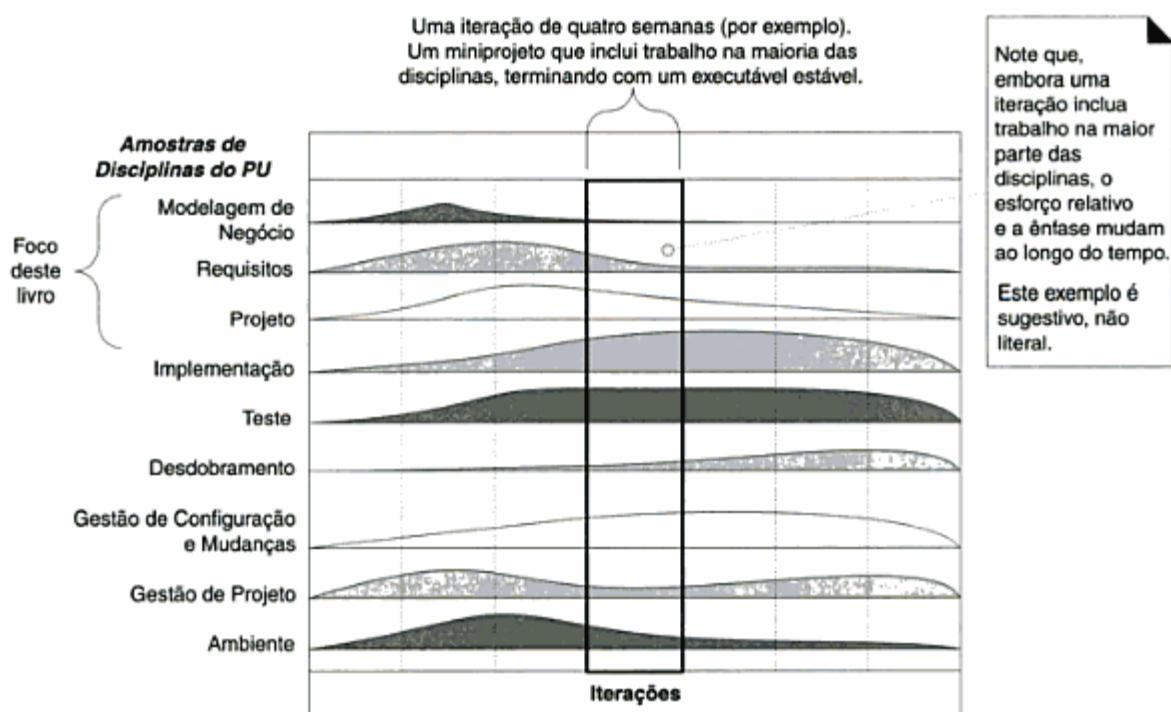


Figura 2.7 Disciplinas do PU.

Relacionando isso com as fases do PU (concepção, elaboração, etc.), a Figura 2.8 ilustra a mudança do esforço relativo em relação às fases; note que isso é uma sugestão e não deve ser tomada "ao pé da letra" como uma recomendação. Por exemplo, na elaboração, as iterações tendem a ter uma carga de análise de requisitos e trabalho de projeto relativamente altos, embora também tenham algum esforço de implementação. Durante a construção, a ênfase é mais pesada na implementação e mais leve na análise de requisitos.

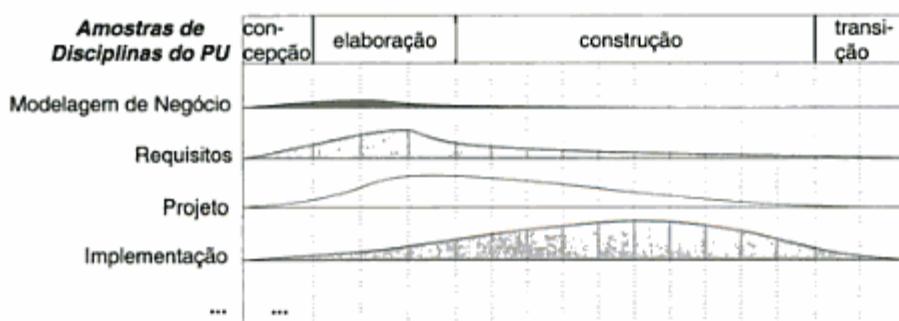


Figura 2.8 Disciplinas e fases.

Como a estrutura deste livro é influenciada pelas fases e disciplinas do PU?

Em relação a fases e disciplinas, qual é o foco dos estudos de caso?

Os estudos de caso enfatizam as fases de concepção e elaboração. Eles enfocam alguns artefatos nas disciplinas de Modelagem de Negócio, Requisitos e Projeto, já que é aqui que a análise de requisitos, a A/POO, os padrões e a UML são principalmente aplicados.

Os capítulos iniciais apresentam as atividades de concepção; os capítulos seguintes exploram várias iterações da elaboração. A lista a seguir e a Figura 2.9 descrevem a organização em relação às fases do PU.

1. Os capítulos relativos à fase de concepção apresentam os fundamentos da análise de requisitos.
2. A iteração 1 introduz os fundamentos da A/POO e como atribuir responsabilidades a objetos.
3. A iteração 2 enfoca o projeto de objetos, especialmente a introdução de alguns “padrões de projeto” altamente utilizados.
4. A iteração 3 introduz diversos assuntos, tais como a análise de arquitetura e o projeto de frameworks.

O Livro

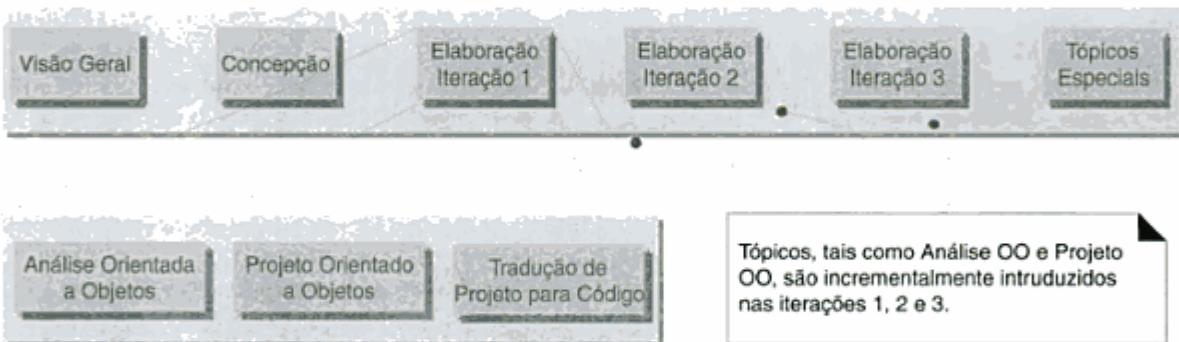


Figura 2.9 A organização do livro está relacionada às fases e iterações do PU.

2.12 Como personalizar o processo? A pasta de desenvolvimento PU

Há artefatos ou práticas opcionais no PU?

Sim! Quase tudo é opcional. Dito isso, algumas das práticas e dos princípios do PU são invariáveis, como o desenvolvimento iterativo e orientado ao controle dos riscos, bem como a verificação contínua da qualidade.

Entretanto, um aspecto-chave do PU é que todas as atividades e os artefatos (modelos, diagramas, documentos, etc.) são *opcionais* – bem, talvez não o código!

Analogia

O conjunto de possíveis artefatos descritos no PU deve ser visto como um conjunto de medicamentos em uma farmácia. Da mesma forma que alguém não toma medicamentos de forma indiscriminada, mas ajusta a escolha de acordo com os sintomas, em um projeto de PU a equipe deve selecionar um pequeno subconjunto de artefatos que atenda a seus problemas e necessidades. Em geral, a equipe deve se concentrar em um *pequeno* conjunto de artefatos que demonstrem possuir um alto valor prático.

Definição: o que é a pasta de desenvolvimento PU?

A escolha de práticas e artefatos do PU para um projeto pode ser escrita em um curto documento denominado **Pasta de Desenvolvimento** (um artefato na disciplina Ambiente). Por exemplo, a Tabela 2.1 poderia ser a Pasta de Desenvolvimento que descreve os artefatos para o “Projeto ProxGer”, o estudo de caso explorado neste livro.

Tabela 2.1 Amostra dos artefatos de uma Pasta de Desenvolvimento do PU

Disciplina	Prática	Artefato Iteração →	Concepção C1	Elaboração El...En	Construção Ct1...Ctn	Transição T1...Tn
Modelagem de Negócio	Modelagem ágil Reunião de requisitos	Modelo de domínio		i		
Requisitos	Reunião de requisitos	Modelo de casos de uso	i	r		
	Exercitar a caixa de visão	Visão	i	r		
	Votação do ponto	Especificações suplementares	i	r		
		Glossário	i	r		
Projeto	Modelagem ágil	Modelo de projeto		i	r	
	Desenvolvimento guiado por teste	Documento de arquitetura de software		i		
		Modelo de dados		i	r	
Implementação	Desenvolvimento guiado por teste Programação aos pares Integração contínua	...				
Padronização de código	Gestão de projeto ágil Reunião diária de Scrum	...				
...						

Os próximos capítulos descrevem a concepção de alguns desses artefatos, incluindo o Modelo de Domínio, o Modelo de Casos de Uso e o Modelo de Projeto.

Os exemplos de práticas e artefatos apresentados nesse estudo de caso não são, de forma alguma, suficientes ou adequados para todos os projetos. Por exemplo, em um sistema de controle de máquinas, pode ser benéfico fazer muitos diagramas de estado. Um sistema de comércio eletrônico baseado na Web pode exigir um foco em protótipos da interface com o usuário. O desenvolvimento de um projeto em um campo totalmente novo tem necessidade de artefatos de projeto bem diferentes daqueles de um projeto de integração de sistemas.

2.13 Você fica sabendo que não compreendeu o desenvolvimento iterativo ou o PU quando...

Aqui são apresentados alguns sinais que indicam quando você não compreendeu o que significa adotar o desenvolvimento iterativo e o PU.

- Você tenta definir a maior parte dos requisitos antes de começar o projeto ou a implementação. Analogamente, você tenta definir a maior parte do projeto antes de começar a implementação; tenta defini-lo completamente e se comprometer com uma arquitetura antes da programação e de testes iterativos.
- Você gasta dias ou semanas em modelagem UML, antes de programar, ou você pensa que as atividades de diagramação e projeto UML são um tempo para definir completa e precisamente projetos e modelos com grande detalhe, e vê a programação como uma tradução simples e mecânica daqueles para código.
- Você pensa que concepção = requisitos, elaboração = projeto e construção = implementação (isto é, sobrepõe um ciclo de vida em cascata no PU).
- Você pensa que a finalidade da elaboração é definir completa e cuidadosamente os modelos, os quais são traduzidos em código durante a construção.
- Você acredita que a duração da iteração adequada é de três meses, em vez de três semanas.
- Você pensa que adotar o PU significa fazer muitas das atividades possíveis e criar muitos documentos, e pensa ou imagina o PU como um processo formal, meticoloso demais, com muitos passos a serem seguidos.
- Você tenta planejar um projeto em detalhes do começo ao fim; tenta especulativamente prever todas as iterações e o que deveria acontecer em cada uma.

2.14 Histórico

Para a história e as citações completas veja *Iterative and Incremental Development: A Brief History* (IEEE Computer, Junho 2003, Larman e Basili) e também [Larman03]. Métodos iterativos são mais antigos do que muitos pensam. No fim da década de 1950 o desenvolvimento evolutivo, iterativo e incremental (DII), em vez do ciclo em cascata, foi aplicado no projeto espacial Mercury, e no início da década de 1960 no projeto do submarino Trident, além de muitos outros sistemas grandes. O primeiro trabalho publi-

cado promovendo o desenvolvimento iterativo em vez do em cascata foi lançado em 1968 no IBM T. J. Watson Research Center.

O DII foi usado em muitos projetos grandes de defesa aeroespaciais na década de 1970, incluindo o software de controle de vôo do *USA Space Shuttle* (construído em 17 iterações com quatro semanas em média cada uma.) Um importante líder do pensamento em engenharia de software da década de 1970, Harlan Mills, escreveu naquele época sobre a falha do método em cascata para projetos de software e sobre a necessidade do DII. Tom Gilb, um consultor privado, criou e publicou o método DII Evo na década de 1970, comprovadamente o primeiro método iterativo completamente formado. O Departamento de Defesa dos Estados Unidos tinha adotado um padrão em cascata no fim da década de 1970 e no começo da de 1980 (DoD-2167); ao final da década de 1980 eles estavam sofrendo significativas falhas (orçamentos de pelo menos 50% dos projetos de software cancelados ou não usáveis) e, assim, ele foi abandonado, posteriormente (começando em 1987) substituído pelos padrões do método DII – apesar de que o legado da influência do método em cascata ainda confunde alguns projetos do Departamento de Defesa.

Também na década de 1980 o Dr. Frederick Brooks (famoso pelo *Mythical Man-Month*), um importante líder do pensamento em engenharia de software daquele período, escreveu e falou sobre as deficiências do método em cascata e a necessidade de, em vez dele, usar métodos DII. Outro marco da década de 1980 foi a publicação do método DII com modelo espiral guiado por risco pelo Dr. Barry Boehm, citando o alto risco de falha quando o método cascata era aplicado.

No início da década de 1990, o DII foi amplamente reconhecido como sucessor do método em cascata e houve um florescimento de métodos iterativos e evolutivos: PU, DSDM, Scrum, XP, e muitos outros.

2.15 Leituras recomendadas

Uma introdução de fácil leitura ao PU e seu refinamento, o RUP, é *The Rational Unified Process – An Introduction*, de autoria de Philippe Kruchten. Também é excelente *The Rational Unified Process Made Easy*, por Kruchten e Kroll.

Agile and Iterative Development: A Manager's Guide [Larman03] discute práticas iterativas e ágeis, quatro métodos iterativos (XP, PU, Scrum e Evo), a evidência e a história relativa a eles e a evidência de falha do método em cascata.

Para saber mais sobre outros métodos iterativos e ágeis, recomendamos a série de livros **Extreme Programming** (XP) [Beck00, BF00, JAH00], tal como *Extreme Programming Explained*. Algumas práticas do XP são encorajadas nos capítulos posteriores deste livro. A maior parte das práticas do XP (tais como programação guiada por teste, integração contínua e desenvolvimento iterativo) é compatível – ou idêntica – com as práticas do PU, sendo encorajada sua adoção em um projeto de PU.

O método Scrum é outra abordagem iterativa popular que aplica iterações de tempo limitado de 30 dias e com uma reunião diária, em pé, com três questões especiais respondidas por cada membro da equipe. *Agile Development with Scrum* é uma leitura recomendada.

Modelagem ágil é descrita em *Agile Modeling*, por Scott Ambler.

A IBM vende um produto de documentação online do RUP baseada na Web, que fornece leitura abrangente sobre os artefatos e atividades do RUP, com gabaritos para a maioria dos artefatos. Uma organização pode desenvolver um projeto de PU usando apenas mentores e livros como recursos de aprendizado, mas alguns acham o produto RUP um processo de ajuda e aprendizado útil.

Para recursos na Web:

- www.agilealliance.com – coleta muitos artigos especificamente relacionados a métodos ágeis e iterativos, além de links.
- www.agilemodeling.com – artigos sobre modelagem ágil.
- www.cetus-links.org – O site Cetus Links tem se especializado durante anos em tecnologia de objetos (TO). Sob o título *OO Project Management – OOA/D Methods* (Gestão de Projetos OO – Métodos A/POO) há vários *links* para métodos iterativos e ágeis, apesar deles não estarem diretamente relacionados à TO.
- www.bradapp.net – Bradd Appleton mantém uma grande coleção de links sobre engenharia de software, inclusive métodos iterativos.
- www.iturls.com – a página inicial em chinês tem links para uma versão em inglês, com um motor de busca referenciando artigos iterativos e ágeis.

ESTUDOS DE CASO

Poucas coisas são mais difíceis de tolerar do que um bom exemplo.
— Mark Twain

Introdução

Esses problemas de estudos de caso (começando nas págs. 71-72) foram escolhidos porque são bem conhecidos pela maioria das pessoas, ao mesmo tempo em que são ricos em complexidade e interessantes problemas de projeto. Isso nos permite concentrar no aprendizado básico da A/POO, análise de requisitos, UML e padrões, em vez de explicar os problemas.

O que vem a seguir?

Apresentado o desenvolvimento iterativo, este capítulo resume os estudos de caso e o nosso enfoque na aplicação da camada lógica. O capítulo seguinte introduz a fase de concepção dos estudos de caso, enfatizando que a concepção *não* é a fase do ciclo em cascata em que se faz a análise antecipada de todos os requisitos.



3.1 O que é e não é abordado nos estudos de caso?

Geralmente, as aplicações incluem elementos de Interface com o Usuário (IU), lógica central da aplicação, acesso ao banco de dados e colaboração com componentes externos de software ou hardware.

Apesar da tecnologia OO poder ser aplicada em todos os níveis, esta introdução à A/POO enfoca a *camada da lógica central da aplicação*, com alguma discussão secundária sobre as outras camadas.

*Definição
dessas ca-
madas
(pág. 221)*

A exploração do projeto das outras camadas (como por exemplo, a camada de IU) enfocará apenas o projeto da sua interface com a camada lógica da aplicação.

Por que enfocar a A/POO da camada de lógica central da aplicação?

- Outras camadas são geralmente muito dependentes de tecnologia/plataforma. Por exemplo, para explorar o projeto OO de uma IU Web ou uma camada IU em Java de cliente “gordo”, precisaríamos aprender em detalhe um framework, como o Struts ou o Swing. Mas para .NET ou Python, a escolha e os detalhes são muito diferentes.
- Em contraposição, o projeto OO da camada lógica central é semelhante em diferentes tecnologias.
- As habilidades de projeto OO essenciais aprendidas no contexto da camada de lógica da aplicação são aplicáveis a todas as outras camadas ou componentes.
- A abordagem de projeto e os padrões de projeto para as outras camadas tendem a mudar rapidamente à medida que outros frameworks ou tecnologias surgem. Por exemplo, no meio da década de 1990 os desenvolvedores iriam provavelmente construir em casa sua própria camada de acesso ao banco de dados objeto-relacional. Alguns anos depois eles provavelmente usariam uma solução livre, de fonte aberta, como, por exemplo, o Hibernate (em tecnologia Java).



Figura 3.1 Exemplo de camadas e objetos em um sistema orientado a objetos e o enfoque dos estudos de caso.

3.2 Estratégia dos estudos de caso: desenvolvimento iterativo + aprendizado iterativo

Este livro é organizado para mostrar uma estratégia de desenvolvimento iterativo. A A/POO é aplicada aos estudos de caso em múltiplas iterações; a primeira trata de algumas funções centrais. As iterações posteriores expandem a funcionalidade do sistema (ver Figura 3.2).

Em conjunto com o desenvolvimento iterativo, a *apresentação* dos tópicos de análise e projeto e a notação UML, os padrões são apresentados iterativa e incrementalmente. Na primeira iteração, é abordado um conjunto central de tópicos da análise, de projeto e da notação UML. A segunda iteração introduz novas idéias, notações UML e novos padrões. E, assim, de modo análogo, a terceira iteração.

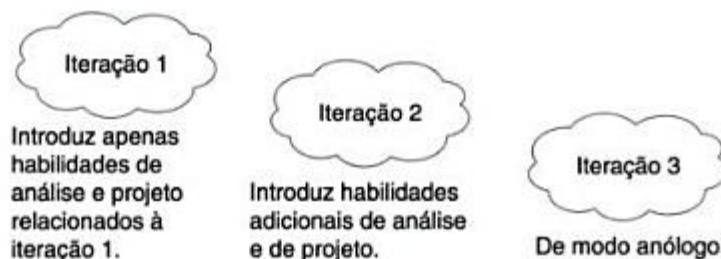


Figura 3.2 A trajetória de aprendizado segue as iterações.

3.3 Caso um: o sistema PDV ProxGer

O primeiro estudo de caso é a próxima geração do Sistema de Pontos-de-Venda (PDV ProxGer). Neste domínio de problema aparentemente simples, veremos que existem problemas muito interessantes de requisitos e de projeto a serem resolvidos. Além disso, é um problema realista; os grupos realmente desenvolvem sistemas PDV usando tecnologias de objetos.

Um sistema PDV é uma aplicação computadorizada usada (em parte) para registrar vendas e cuidar de pagamentos; é tipicamente utilizado por lojas de varejo. Inclui componentes de hardware, como um computador e um leitor de código de barras, e um software para rodar o sistema. Tem interfaces com várias aplicações de serviço, como, por exemplo, uma aplicação de cálculo de impostos e uma aplicação de controle de estoque. Esses sistemas devem ser relativamente tolerantes a falhas; ou seja, mesmo que os serviços remotos fiquem temporariamente não disponíveis (como por exemplo o sistema de controle de estoque), eles devem ser capazes de capturar as vendas e tratar pelo menos os pagamentos em dinheiro (para que o negócio não seja muito afetado).



Um sistema PDV deve dar suporte de forma incremental a múltiplos e variados terminais e interfaces no lado do cliente. Estes incluem um terminal magro baseado em navegador da Web, um computador pessoal comum com uma interface de usuário gráfica, como Java Swing, entrada de informações por telas sensíveis ao toque, PDAs sem fio, etc.

Além disso, vamos criar um sistema PDV comercial que venderemos a diferentes clientes, com necessidades diferentes em termos de processamento de regras de negócios. Cada cliente vai desejar um conjunto especial de lógica para ser executado em pontos previsíveis no cenário de utilização do sistema, como quando uma nova venda é iniciada ou quando uma nova linha é acrescentada. Portanto, necessitaremos de um mecanismo que forneça esta capacidade de flexibilidade e personalização.

Usando uma estratégia de desenvolvimento iterativo, executaremos a análise de requisitos, a análise orientada a objetos, o projeto e a implementação.

3.4 Caso dois: o sistema do jogo Banco Imobiliário

Para mostrar que as mesmas práticas de A/POO podem ser aplicadas a problemas *muito* diferentes, escolhi uma versão em software do jogo Banco Imobiliário® como outro estudo de caso. Apesar do domínio e dos requisitos não serem em nada parecidos aos de um sistema de negócios como o PDV ProxGer, veremos que a modelagem de domínio, o projeto dos objetos com padrões e a aplicação da UML continuam relevantes e úteis. Do mesmo modo que um PDV, as versões de software do Banco Imobiliário são realmente desenvolvidas e vendidas, tanto com IU Web quanto com IU cliente rico.



Não vou repetir as regras do Banco Imobiliário; parece que quase todas as pessoas em todos os países jogaram esse jogo quando criança ou jovem. Se você tiver dúvidas, as regras estão disponíveis em muitos sites na Web.

A versão em software do jogo será executada como uma simulação. Uma pessoa começará o jogo e indicará o número de jogadores simulados e depois observará enquanto o jogo se desenrola, apresentando um rastro da atividade durante a vez dos jogadores simulados.

PARTE II CONCEPÇÃO

CONCEPÇÃO NÃO É A FASE DE REQUISITOS

O ótimo é inimigo do bom.
— Voltaire

Objetivos

- Definir o passo de concepção.
- Motivar os capítulos seguintes desta seção.

Introdução

A concepção é um passo inicial curto, para estabelecer uma visão comum e o escopo básico do projeto. Incluirá a análise de talvez 10% dos casos de uso, análise dos requisitos não funcionais críticos, criação de um caso de negócio e preparação do ambiente de desenvolvimento, para que a programação possa começar na fase seguinte de elaboração.

O que vem a seguir?

Introduzidos os estudos de caso, este capítulo explora a fase de concepção — que não é a fase de requisitos do ciclo em cascata. O capítulo seguinte examina esse ponto mais profundamente: requisitos evolutivos x em cascata.



4.1 O que é concepção?

A maior parte dos projetos exige um passo inicial curto, no qual os seguintes tipos de questões são explorados:

- Qual é a visão e o caso de negócio para este projeto?
- Ele é viável?
- Devemos construir ou comprar?
- Estimativa aproximada de custo: é da ordem de \$ 10 a 100 mil ou de milhões?
- Devemos continuar ou parar?

Definir a visão e obter uma estimativa da ordem de grandeza (não confiável) exige alguma exploração dos requisitos. Contudo, a **finalidade da fase de concepção não é definir todos os requisitos** ou gerar uma estimativa ou plano de projeto plausíveis.

Definição

Esse é um ponto crucial e repetidamente mal-entendido em projetos PU quando as pessoas sobrepõem o velho raciocínio “em cascata”. O PU não é o ciclo em cascata e a primeira fase, concepção, não é a época de eliciar todos os requisitos ou criar planos e estimativas confiáveis. Isso acontece durante a elaboração.

Sob o risco de simplificarmos demasiadamente, a idéia é fazer uma investigação suficiente para formar uma opinião racional e justificável da finalidade geral e da viabilidade do novo sistema em potencial, para então decidir se vale a pena investir em uma exploração mais profunda (a finalidade da fase de elaboração).

A maior parte da análise de requisitos ocorre durante a fase de elaboração, em paralelo com o início da programação com qualidade de produção e com os testes.

Assim, a fase de concepção deve ser relativamente curta para a maioria dos projetos, com uma ou poucas semanas de duração. De fato, em muitos projetos, se ela tiver mais de uma semana de duração, o sentido dela não foi compreendido: sua finalidade é decidir se o projeto merece uma investigação séria (durante a elaboração) e não executar tal investigação.

A concepção, em uma frase:

Conceber o escopo do produto, a visão e o caso de negócio.

O problema principal a ser resolvido, em uma frase:

Os interessados no projeto do sistema têm um consenso básico sobre a visão do projeto e vale a pena investir em uma investigação séria?

A analogia ajuda?

No negócio de petróleo, quando um novo campo é avaliado, alguns dos passos dessa avaliação incluem:

1. Decidir se há evidência suficiente ou um caso de negócios para justificar uma prospecção exploratória.
2. Efetuar, se for o caso, medições e prospecções exploratórias.
3. Fornecer informações sobre o escopo e o orçamento.
4. Realizar passos adicionais...

A fase de concepção é como o passo um nessa analogia. No passo um, as pessoas não prevêem quanto petróleo há ou o custo ou esforço para extraí-lo. Embora fosse desejável poder responder às perguntas “quanto” e “quando” sem o custo e o esforço da exploração, no negócio de petróleo não seria realista.

Em termos do PU, o passo de exploração realista é a fase de elaboração. A fase precedente de concepção é semelhante a um estudo de viabilidade para decidir se vale a pena investir em “perfurações” exploratórias. Somente depois da exploração (elaboração) é que teremos os dados e a percepção para elaborar estimativas e planos com um pouco mais de credibilidade. Portanto, no desenvolvimento iterativo e no PU, planos e estimativas não devem ser considerados confiáveis na fase de concepção. Eles meramente fornecem um sentimento da ordem de grandeza do nível de esforço envolvido, para nos auxiliar em uma decisão de prosseguir ou não.

4.2 Quanto deve durar a concepção?

O propósito da concepção é estabelecer uma visão inicial comum para os objetivos do projeto, determinar se o mesmo é viável e decidir se ele realmente deve passar por uma investigação mais séria na fase de elaboração. Se foi definido de antemão que o projeto será feito e ele for claramente viável (talvez porque a equipe já tenha executado projetos como este), a fase de concepção será especialmente breve. Ela pode incluir os primeiros seminários sobre requisitos, o planejamento para a primeira iteração, passando rapidamente para a elaboração.

4.3 Quais artefatos podem ser iniciados na concepção?

A Tabela 4.1 lista artefatos comuns da concepção (ou de uma elaboração precoce) e indica os aspectos que eles devem abordar. Os capítulos subsequentes examinarão alguns desses aspectos em detalhes, especialmente o Modelo de Casos de Uso. Uma questão-chave para a compreensão do desenvolvimento iterativo é perceber que esses artefatos são apenas parcialmente completados nesta fase, sendo refinados em iterações posteriores, não devendo nem mesmo ser criados, a menos que se perceba que eles acrescentarão um real valor prático. Além disso, uma vez que estamos na concepção, o conteúdo da investigação e dos artefatos deve ser leve.

Tabela 4.1 Amostras de artefatos para a concepção

Artefato*	Comentário
Visão e Caso de Negócio	Descrevem os objetivos e as restrições de alto nível, o caso de negócio, além de um resumo para executivos.
Modelo de Casos de Uso	Descreve os requisitos funcionais. Durante a concepção, os nomes da maior parte dos casos de uso vão ser identificados e talvez 10% dos casos de uso serão analisados em detalhes.
Especificações Suplementares	Descrevem outros requisitos, a maior parte não funcional. Durante a concepção é útil ter alguma idéia dos requisitos-chave não funcionais que deverão ter impacto importante na arquitetura.
Glossário	Contém a terminologia-chave do domínio e dicionário de dados.
Lista de Riscos e Plano de Gestão de Riscos	Descrevem os riscos (de negócio, técnicos, de recursos e de cronograma), bem como as idéias para sua minimização ou solução.
Protótipos e provas de conceitos	Visam a esclarecer a visão e validar as idéias técnicas.
Plano da Iteração	Descreve o que fazer na primeira iteração da elaboração.
Plano da Fase e Plano de Desenvolvimento de Software	Estimativa de baixa precisão para a duração e esforço da fase de elaboração. Ferramentas, pessoal, treinamento e outros recursos.
Pasta de Desenvolvimento	Descrição dos passos e artefatos do PU personalizados para esse projeto. O PU é sempre personalizado para um projeto.

* Esses artefatos são apenas parcialmente completados nesta fase. Eles serão iterativamente refinados nas iterações subsequentes. Um nome grafado em letra maiúscula indica que ele é um artefato oficialmente nomeado pelo PU.

Por exemplo, o Modelo de Casos de Uso pode listar os *nomes* de casos de uso e atores mais esperados, mas talvez descreva somente 10% dos casos de uso em detalhe – o que é feito durante o desenvolvimento da visão estimativa de alto nível do escopo do sistema, sua finalidade e riscos envolvidos.

Note que pode ocorrer algum trabalho de programação na concepção, com o objetivo de criar protótipos para “prova de conceitos”, visando a esclarecer alguns requisitos via (tipicamente) protótipos voltados para a IU (Interface do Usuário), bem como a execução de algumas experiências de programação relativas a questões técnicas-chave que podem impedir que prossigamos.

Não é documentação demais?

Lembre-se de que os artefatos devem ser considerados opcionais. Crie somente aqueles que realmente acrescentem valor ao projeto, eliminando aqueles cujo valor não estiver comprovado. Além disso, como se trata de um desenvolvimento evolutivo, o ponto não é criar especificações completas durante esta fase, mas um esboço de documentos iniciais, que são refinados durante as iterações de elaboração, em resposta à valiosa realimentação oriunda da programação e testes antecipados.

Também, o objetivo de criar artefatos ou modelos não é o documento ou diagrama em si, mas o raciocínio, a análise e a presteza proativa. Essa é uma perspectiva da modelagem ágil: que o maior valor da modelagem é melhorar o entendimento, em vez de documentar especificações confiáveis. Como disse o general Eisenhower: "Na preparação para uma batalha, eu sempre percebo que os planos são inúteis, mas o planejamento é indispensável" [Nixon90, BF00].

Note também que artefatos do PU de projetos anteriores podem ser parcialmente reutilizados em projetos posteriores. É comum que existam muitas similaridades no que diz respeito aos riscos, à gerência de projeto, aos testes e aos artefatos relacionados ao ambiente entre vários projetos. Todos os projetos PU deveriam organizar os artefatos do mesmo modo, com os mesmos nomes (Lista de Riscos, Pasta de Desenvolvimento, etc.). Isso torna mais fácil a busca por artefatos reutilizáveis de projetos anteriores em novos esforços que adotem o PU.

4.4 Você sabe que não compreendeu a concepção quando...

- Ela gasta mais do que "poucas" semanas para a maioria dos projetos.
- Há uma tentativa de definir a maioria dos requisitos.
- Espera-se que as estimativas ou planos sejam confiáveis.
- Você define a arquitetura (isso deveria ser feito iterativamente na elaboração).
- Você acredita que a seqüência correta de trabalho deveria ser: 1) definir os requisitos; 2) projetar a arquitetura; 3) implementar.
- Não há um artefato Caso de Negócio ou Visão.
- Todos os casos de uso foram definidos em detalhes.
- Nenhum dos casos de uso foi definido em detalhes; de preferência, 10% a 20% deveriam ser definidos em detalhes para se ganhar uma percepção realista do escopo do problema.

4.5 Quanto de UML durante a concepção?

A finalidade da concepção é coletar apenas informação suficiente para estabelecer uma visão comum, decidir se avançar é exequível e se o projeto merece séria investigação na fase de elaboração. Assim, a menos talvez de simples diagramas de casos de uso UML, não seja necessária muita diagramação. Há maior enfoque durante a concepção no entendimento do escopo básico e de 10% dos requisitos, expressos na sua maior parte em forma de texto. Na prática e também nesta apresentação, a maior parte da diagramação UML ocorrerá na fase seguinte – a elaboração.

Capítulo

5

REQUISITOS EVOLUTIVOS

Em nosso mundo as pessoas não sabem o que querem, mas estão dispostas a ir até o inferno para consegui-lo.
– Don Marquis

Objetivos

- Motivar a adoção de requisitos evolutivos.
- Definir o modelo FURPS+.
- Definir os artefatos de requisitos do PU.

Introdução

Outras práticas do PU (págs. 60-61) Este capítulo introduz brevemente os requisitos iterativos e evolutivos e descreve artefatos de requisitos específicos do PU, a fim de fornecer contexto para os capítulos seguintes, que são orientados a requisitos.

Também explora alguma evidência que ilustra a futilidade e falta de habilidade das abordagens de análise de requisitos orientados em cascata, nas quais há uma tentativa de definir as conhecidas especificações “completas” antes de começar o desenvolvimento.

O que vem a seguir?

Introduzida a concepção, este capítulo introduz requisitos e seu refinamento evolutivo. O capítulo seguinte aborda os casos de uso, que são uma prática para coletar os requisitos iniciais utilizada no PU e em muitos métodos modernos.



5.1 Definição: requisitos

Requisitos são capacidades e condições às quais o sistema – e em termos mais amplos, o projeto – deve atender [JBR99].

O PU promove um conjunto de melhores práticas, uma das quais é *gerenciar requisitos*. Isso não se refere à atitude dos que adotam o ciclo de desenvolvimento em cascata, de tentar definir e estabilizar por completo os requisitos na primeira fase de um projeto antes da programação, mas sim – no contexto das inevitáveis mutações e dos desejos não muito claros dos interessados no projeto – “uma abordagem sistemática para encontrar, documentar, organizar e rastrear as *mudanças* de requisitos de um sistema” [RUP].

Em resumo, fazendo-o de forma iterativa, competente e não descuidada.

Um desafio básico da análise de requisitos é encontrar, comunicar e lembrar (o que geralmente quer dizer registrar) o que é realmente necessário, expressando isso de forma clara para o cliente e os membros da equipe de desenvolvimento.

5.2 Requisitos evolutivos vs. em cascata

Note a palavra *mudança* na definição do que significa gerenciar requisitos. O PU encara a mudança nos requisitos como um componente fundamental nos projetos. Isso é incrivelmente importante e constitui a parte central do raciocínio envolvendo o ciclo em cascata versus o ciclo iterativo e evolutivo.

No PU e em outros métodos evolutivos (Scrum, XP, FDD e outros), iniciamos a programação e teste com qualidade de produção muito antes da maior parte dos requisitos terem sido analisados ou especificados – talvez quando apenas 10 ou 20% dos requisitos mais significativos arquiteturalmente, de maior risco e de mais alto valor de negócios, tenham sido especificados.

Quais são os detalhes do processo? Como fazer a análise parcial e evolutiva de requisitos, combinada com projeto e programação antecipados, em iterações? Ver “Como Fazer Análise e Projeto Iterativos e Evolutivos?” na pág. 53, que fornece uma breve descrição e uma figura para ajudar a explicar o processo. Ver “Processo: Como Trabalhar com Casos de Uso e Métodos Iterativos?” na pág. 121, que apresenta uma discussão mais detalhada.

Cuidado!

Se você estiver em um projeto aclamado como PU ou iterativo, mas que tenta especificar a maior parte ou todos os requisitos (casos de uso, etc.) antes de começar a programar e testar, existe um profundo mal entendido – não é um projeto iterativo ou um PU saudável.

Nas décadas de 1960 e 1970 (quando comecei a trabalhar como desenvolvedor) havia ainda uma crença especulativa comum sobre a eficácia da análise antecipada e completa de requisitos para projetos de software (isto é, em cascata). Na década de 1980 começou a surgir evidências de que isso não era correto e levava a muitas falhas; a crença antiga estava enraizada no paradigma errado de considerar um projeto de software semelhante a uma previsível fabricação em massa, com baixas taxas de modificação. Mas o software permanece no domínio de desenvolvimento de novos produtos com altos intervalos de modificação e altos graus de novidade e descoberta.

Mudanças na pesquisa (pág. 52)

Lembre a estatística-chave de que, em média, 25% dos requisitos são modificados nos projetos de software. Assim, qualquer método que tenta congelar ou definir totalmente os requisitos no começo é fundamentalmente falho, baseado em uma suposição falsa, e combate ou nega as inevitáveis mudanças.

Reforçando esse ponto, por exemplo, estava um estudo sobre fatores de falhas em 1027 projetos de software [Thomas01]. O que foi encontrado? Tentativas de práticas em cascata (inclusive requisitos detalhados no início) eram o fator individual que mais contribuía para a falha, sendo indicado em 82% dos projetos como o problema número um. Citando a conclusão:

... a abordagem de definição completa de requisitos, seguida de um longo intervalo antes daqueles requisitos serem entregues, não é mais adequada.

O alto grau de requisitos de negócios mutáveis sugere que qualquer suposição de que vai haver pouca mudança significativa nos requisitos, após eles terem sido documentados, é fundamentalmente errada e que gastar tempo e esforço significativos definindo-os no nível máximo é inadequado.

Outro resultado relevante de pesquisa responde a seguinte questão: quando se tenta fazer a análise de requisitos em cascata, quantas das características prematuras especificadas inicialmente são realmente úteis no produto de software final? Em um estudo [Johnson02] de milhares de projetos, os resultados são bastante reveladores – 45% de tais características nunca foram usadas e, além disso, 19% foram “raramente” usadas (ver Figura 5.1). Quase 65% das características especificadas em cascata foram de pequeno ou de nenhum valor!

Esses resultados não implicam que a solução seja começar a mexer no código próximo ao primeiro dia do projeto e esquecer a análise de requisitos ou de registrar os requisitos. Há um meio termo: análise iterativa e evolutiva de requisitos, combinada com desenvolvimento iterativo de tempo limitado e freqüente participação dos interessados na avaliação e realimentação dos resultados parciais.

5.3 Quais são os meios racionais de eliciar requisitos?

Revisando a boa prática do PU de gerenciar requisitos:

... uma abordagem sistemática de encontrar, documentar, organizar e rastrear as mudanças de requisitos de um sistema. [RUP]

Além de *mudar*, a palavra *encontrar* é importante; isto é, o PU encoraja a elicitação racional por meio de técnicas como: escrever os casos de uso com os clientes, promover

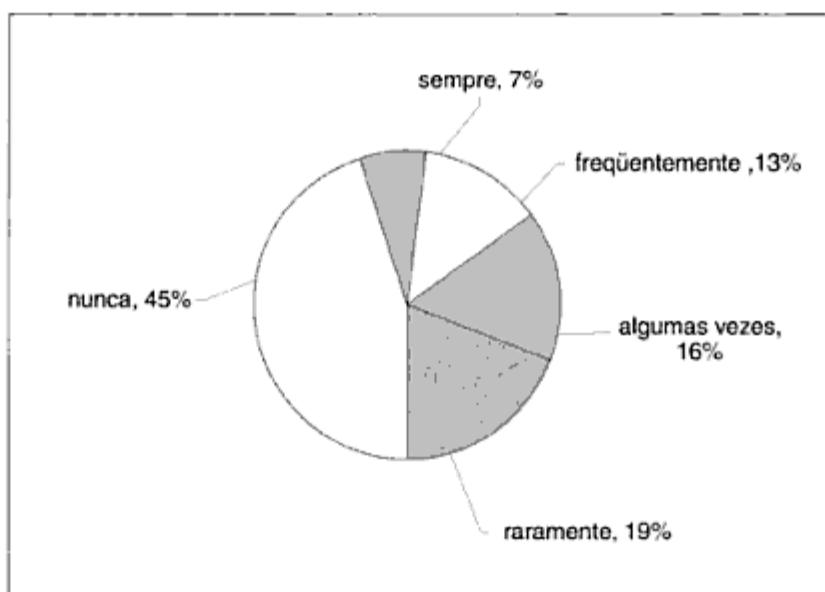


Figura 5.1 Uso efetivo das características especificadas no modelo em cascata.

seminários de requisitos que incluem tanto desenvolvedores quanto clientes, enfocar grupos com pseudoclientes e uma demonstração dos resultados de cada iteração para os clientes, para solicitar realimentação.

O PU acolhe qualquer método de elicitação de requisitos que possa adicionar valor e aumentar a participação do usuário. Mesmo a simples prática do XP de “cartão de estória” (story card) é aceitável em um projeto PU, se ela puder ser colocada em ação efetivamente (isso exige a presença de um especialista em clientes em tempo integral na sala do projeto – uma prática excelente, mas freqüentemente difícil de conseguir).

5.4 Quais são os tipos e categorias de requisitos?

No PU, os requisitos são categorizados de acordo com o modelo FURPS+ [Grady92], um mnemônico útil com o seguinte significado¹:

- **Funcional** – características, capacidade, segurança.
- **Usabilidade** – fatores humanos, recursos de ajuda, documentação.
- **Confiabilidade** – freqüência de falhas, capacidade de recuperação, previsibilidade.
- **Desempenho** – tempos de resposta, fluxo de vazão (throughput), precisão, disponibilidade, uso de recursos.

¹ Existem vários sistemas de categorização de requisitos e atributos de qualidade publicados em livros e por organizações de padronização, como a ISO 9126 (a qual é similar à lista FURPS+), e várias do Software Engineering Institute (SEI); qualquer uma delas pode ser usada em um projeto de PU.

- **Facilidade de Suporte** – facilidade de adaptação e de manutenção, internacionalização, configurabilidade.

O “+” em FURPS+ indica aspectos auxiliares e subfatores, tais como:

- **Implementação** – limitações de recursos, linguagens e ferramentas, hardware, etc.
- **Interface** – restrições impostas pelas interfaces com sistemas externos.
- **Operações** – gerenciamento do sistema no ambiente operacional.
- **Empacotamento** – por exemplo, uma caixa física.
- **Questões legais** – licenças de uso, etc.

É de grande ajuda o uso das categorias FURPS+ (ou algum outro esquema de categorização) como uma lista de verificação para a cobertura dos requisitos, a fim de reduzir o risco de não levar em conta alguma faceta importante do sistema.

Alguns desses requisitos são coletivamente chamados de **atributos de qualidade**, **requisitos de qualidade** ou as “-idades” de um sistema. Estas incluem a usabilidade, a confiabilidade, o desempenho e a facilidade de suporte. No uso comum, requisitos são categorizados como **funcionais** (comportamentais) ou **não funcionais** (todos os outros); alguns desaprovam essa generalização ampla [BCK98], mas ela é bastante usada.

Análise arquitetural
(pág. 542)

Como veremos quando explorarmos a análise arquitetural, os requisitos de qualidade têm uma forte influência na arquitetura de um sistema. Por exemplo, um requisito de alto desempenho e alta confiabilidade influenciará a escolha dos componentes de software e hardware, bem como a sua configuração.

5.5 Como os requisitos são organizados em artefatos PU?

O PU oferece vários artefatos de requisitos. Como todos os artefatos do PU, eles são opcionais. Os artefatos-chave incluem:

- **Modelo de Casos de Uso** – um conjunto de cenários típicos do uso de um sistema. Eles são principalmente para requisitos funcionais (comportamentais).
- **Especificação Suplementar** – basicamente tudo o que não está nos casos de uso. Esse artefato é principalmente para requisitos não funcionais, como desempenho ou autenticação. É também o lugar para registrar características funcionais não expressas (ou expressíveis) como casos de uso; por exemplo, a geração de um relatório.
- **Glossário** – em sua forma mais simples, o glossário define termos importantes. Também abrange o conceito de **dicionário de dados**, que registra requisitos relativos aos dados, tais como: regras de validação, valores aceitáveis, etc. O glossário pode detalhar qualquer elemento: um atributo de um objeto, um parâmetro de chamada de uma operação, um leiaute de relatório, etc.
- **Visão** – resume requisitos de alto nível que são detalhados no modelo de casos de uso e especificação suplementar, e resume o caso de negócios do projeto. É um documento curto contendo uma visão geral para entendimento rápido das grandes idéias do projeto.

- **Regras de Negócio** – regras de negócio (também chamadas de regras de domínio) descrevem tipicamente requisitos ou políticas que transcendem um projeto de software – elas são necessárias no domínio ou no negócio e muitas aplicações podem precisar respeitá-las. Um excelente exemplo são as leis sobre impostos governamentais. Os detalhes das regras de domínio *podem* ser registrados na especificação suplementar, mas como elas são usualmente mais duradouras e aplicáveis do que para um projeto de software, colocá-las em um artefato central de Regras de Negócio (compartilhado por todos os analistas da empresa) leva ao melhor reúso do esforço de análise.

Qual é o formato correto desses artefatos?

No PU, todos os artefatos são abstrações de informação; eles poderiam ser armazenados em páginas da Web (como em uma Wiki Web), murais ou qualquer variante imaginável. O produto da documentação RUP online contém gabaritos para os artefatos, mas são uma ajuda opcional e podem ser ignorados.

5.6 O livro contém exemplos desses artefatos?

Sim! Este livro está mais voltado à uma introdução à A/POO em um processo iterativo do que à análise de requisitos, mas explorar a A/POO sem algum exemplo ou contexto de requisitos dá um panorama incompleto – ignora a influência dos requisitos na A/POO. E é útil ter um exemplo maior de artefatos-chave do PU relativos a requisitos. Onde encontrar os exemplos:

Artefato de requisitos	Onde?	Comentário
Modelo de Casos de Uso	Introdução pág. 87 Intermediário págs. 403-404	Casos de uso são comuns no PU e uma entrada para a A/POO e, assim, são descritos em detalhe nos capítulos iniciais.
Especificação Suplementar, Glossário, Visão, Regras de Negócio	Exemplos de estudos de caso pág. 127	São fornecidos para dar consistência, mas podem ser pulados – não se trata de um tópico de A/POO.

5.7 Leituras recomendadas

Em um capítulo subsequente, cobriremos referências bibliográficas relacionadas aos requisitos com casos de uso. Textos sobre requisitos orientados a casos de uso, como *Writing Effective Use Cases* [Cockburn01], são o ponto de partida recomendado para o estudo de requisitos, em vez de textos mais genéricos (e, usualmente, tradicionais) sobre requisitos.

Há, atualmente, um amplo esforço para discutir os requisitos – e uma grande variedade de tópicos de engenharia de software – sob a sombra do Corpo de Conhecimento de Engenharia de Software (Software Engineering Body of Knowledge – SWEBOK), disponível em www.swebok.org.

O SEI (www.sei.cmu.edu) tem várias propostas relacionadas a requisitos de qualidade. A ISO 9126, a IEEE Std 830 e a IEEE Std 1061 são normas (padronizações) relacionadas a requisitos e atributos de qualidade e estão disponíveis na Web em diversos sítios.

Algumas precauções devem ser tomadas no que diz respeito a livros sobre requisitos gerais, mesmo em relação àqueles que se propõe a cobrir casos de uso, desenvolvimento iterativo ou mesmo requisitos no PU:

A maioria deles está escrita com uma distorção do modelo em cascata, propondo uma definição significativa ou “completa” de requisitos logo no início, antes da execução de qualquer atividade relacionada ao projeto e à implementação. Esses livros que também mencionam o desenvolvimento iterativo podem fazê-lo, digamos, superficialmente, talvez sob a forma de material “iterativo” adicionado recentemente ao texto para satisfazer às tendências modernas. Eles podem conter boas indicações quanto à elicitação e organização de requisitos, mas não oferecem uma visão precisa da análise evolutiva e iterativa.

Qualquer variante de conselho sobre o processo que implique “tentar definir a maioria dos requisitos e, então, partir para o processo de projeto e implementação” não é consistente com o desenvolvimento iterativo evolutivo e o PU.

CASOS DE USO

O primeiro e indispensável passo para obter as coisas que você deseja da vida é decidir o que você quer.

— Ben Stein

Objetivos

- Identificar e escrever casos de uso.
- Usar os formatos resumido, informal e completo, em um estilo essencial.
- Aplicar testes para identificar casos de uso adequados.
- Relacionar a análise de um caso de uso ao desenvolvimento iterativo.

Introdução

Tópicos de
casos de uso
intermediá-
rios (págs.
498-499)

Casos de uso são narrativas em texto, amplamente utilizadas para descobrir e registrar requisitos. Eles influenciam muitos aspectos de um projeto, inclusive a A/POO, e servem de entrada para vários artefatos subseqüentes nos estudos de caso. Este capítulo explora conceitos básicos, inclusive como redigir casos de uso e desenhar um diagrama de casos de uso UML. Este capítulo também mostra o maior valor da habilidade de análise em comparação com o conhecimento sobre a notação UML; o diagrama de caso de uso UML é trivial para aprender, mas as várias diretrizes para identificar e redigir bons casos de uso levam semanas – ou mais – para dominar totalmente.

O que vem a seguir?

Abordados os requisitos, este capítulo explora casos de uso para requisitos funcionais. O capítulo seguinte cobre outros requisitos no PU, inclusive a Especificação Suplementar para requisitos não funcionais.



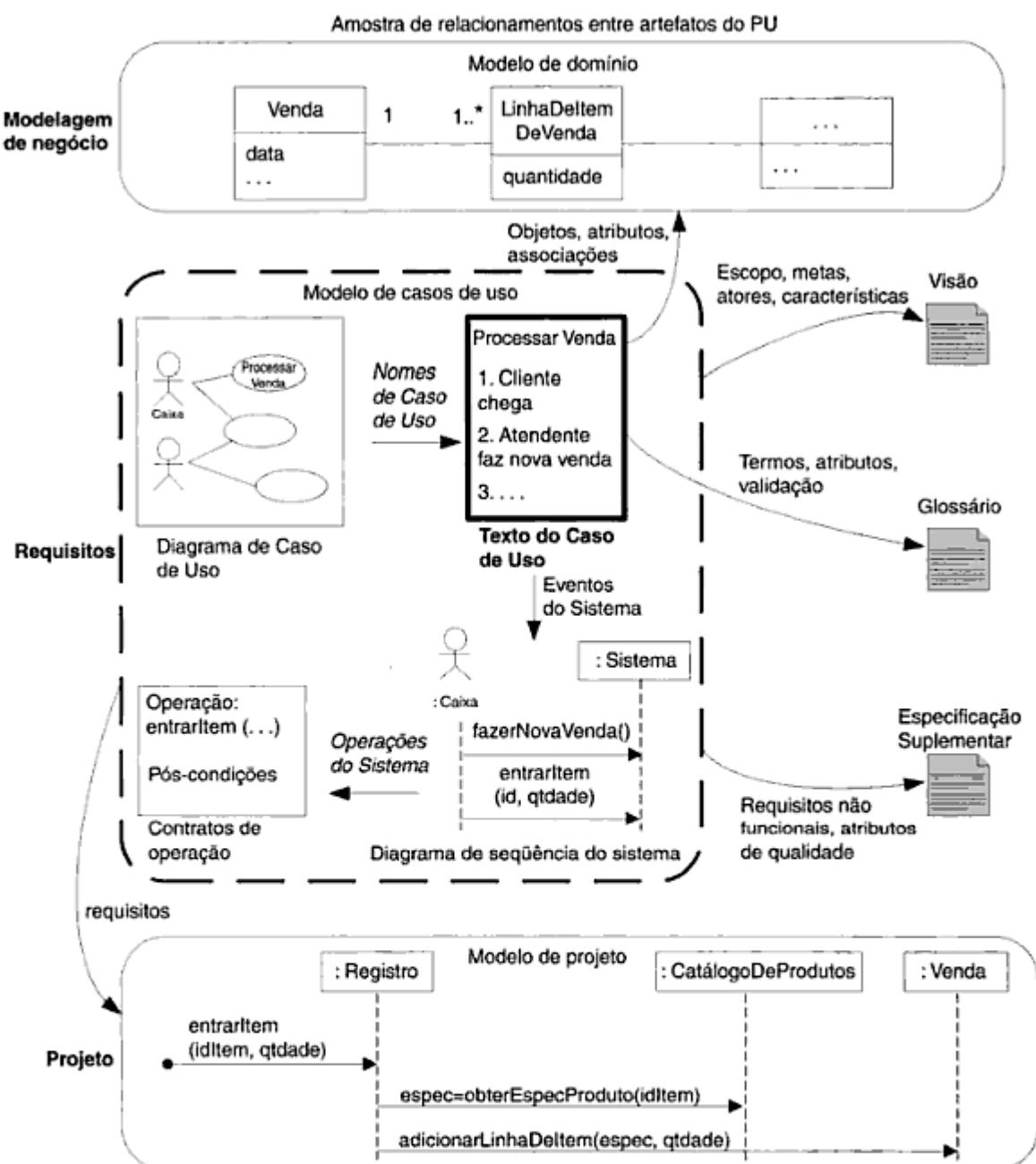


Figura 6.1 Amostra da influência entre artefatos do PU.

A influência entre os artefatos do PU, com ênfase no texto dos casos de uso, é mostrada na Figura 6.1. Metas de alto nível e diagramas de casos de uso são utilizadas como entrada para a criação do texto do caso de uso. Os casos de uso, por sua vez, influenciam muitos outros artefatos de análise, projeto, implementação, gestão de projeto e teste.

6.1 Exemplo

Informalmente, casos de uso são narrativas em texto de algum ator usando um sistema para atingir objetivos. Segue um exemplo de caso de uso no *formato resumido*:

Processar Venda: um cliente chega em um ponto de pagamento com itens que deseja adquirir. O caixa usa o sistema PDV para registrar cada item comprado. O sistema apresenta um total parcial e detalhes de linha de item. O cliente entra os dados sobre o pagamento, que são validados e, em seguida, registrados pelo sistema. O sistema atualiza o estoque. O cliente recebe um recibo do sistema e sai com os itens comprados.

Os diagramas de caso de uso UML (págs. 115-116) Note que **casos de uso não são diagramas, são textos**. Enfocar os diagramas de caso de uso UML, de valor secundário, em vez do importante texto do caso de uso, é um erro comum para novatos.

Casos de uso freqüentemente necessitam ser mais detalhados ou estruturados do que no exemplo, mas o essencial é descobrir e registrar os requisitos funcionais, escrevendo narrativas de uso de um sistema para satisfazer as metas do usuário, ou seja, *casos de uso*¹. Não se trata de uma idéia difícil, embora possa ser difícil descobrir o que é necessário e escrevê-lo de forma coerente.

6.2 Definição: o que são atores, cenários e casos de uso?

Primeiro, daremos algumas definições informais: um **ator** é algo com comportamento, tal como uma pessoa (identificada por seu papel), um sistema de computador ou uma organização; por exemplo, um caixa.

Um **cenário** é uma seqüência específica de ações e interações entre atores e o sistema; é também chamado de **instância de caso de uso**. É uma história particular de uso de um sistema ou um caminho através do caso de uso; por exemplo, o cenário de efetuar com sucesso a compra de itens em dinheiro, ou o cenário de não consumar a compra de itens por causa da recusa de uma autorização de crédito.

Em termos informais, então, um **caso de uso** é uma coleção de cenários relacionados de sucesso e fracasso, que descrevem um ator usando um sistema como meio para atingir um objetivo. Por exemplo, temos a seguir um caso de uso em um *formato informal* que inclui alguns cenários alternativos:

Tratar Devoluções

Cenário de sucesso principal: um cliente chega a um posto de pagamento com itens a serem devolvidos. O caixa usa o sistema PDV para registrar cada item devolvido...

¹ O termo original em sueco seria traduzido literalmente como “caso de utilização”.

Cenários alternativos:

Se o cliente pagou a crédito e a transação de reembolso para estorno em sua conta de crédito é rejeitada, informe o cliente e o reembolse com dinheiro.

Se o identificador do item não for encontrado no sistema, este notifica o caixa e sugere que entre manualmente o código do produto (talvez ele esteja corrompido).

Se o sistema detecta uma falha para se comunicar com o sistema externo de contabilidade...

Agora que cenários (instâncias de casos de uso) estão definidos, uma definição alternativa, porém similar, de caso de uso fornecida pelo RUP vai fazer mais sentido:

Um conjunto de instâncias de casos de uso, no qual cada instância é uma seqüência de ações que um sistema executa e que fornece um resultado observável com valor para um determinado ator [RUP].

6.3 Casos de uso e o modelo de casos de uso

O PU define o **Modelo de Casos de Uso** dentro da disciplina Requisitos. Essencialmente, trata-se do conjunto de todos os casos de uso escritos; é um modelo da funcionalidade e ambiente do sistema.

**Casos de uso são documentos de texto e não diagramas.
Portanto, a modelagem de caso de uso é essencialmente uma ação de redigir texto e não de desenhar diagramas.**

Outros requisitos do PU (pág. 127) O Modelo de Casos de Uso não é o único artefato de requisitos no PU. Existem também: Especificação Suplementar, Glossário, Visão e Regras de Negócio. São todos úteis para a análise de requisitos, mas secundários neste ponto.

Diagramas de Casos de Uso UML (págs. 115-116) O Modelo de Casos de Uso pode opcionalmente incluir um diagrama de caso de uso UML para mostrar os nomes de casos de uso, atores e seus relacionamentos. Isso dá um belo **diagrama de contexto** de um sistema e do seu ambiente. Fornece também um modo rápido de listar os casos de uso por nome.

Não há nada orientado a objetos em casos de uso; não estamos fazendo análise OO quando os escrevemos. Isso não é problema – casos de uso são amplamente aplicáveis, o que aumenta a sua utilidade. Dito isso, casos de uso são considerados uma entrada de requisitos chave para a A/POO clássica.

6.4 Motivação: por que casos de uso?

Nós temos objetivos e desejamos computadores para ajudar a atingi-los, que vão desde o registro de vendas até o uso de jogos para estimar o fluxo de petróleo de fu-

turos poços. Analistas experientes inventaram *muitos* modos de descobrir objetivos, mas os melhores são simples e familiares. Por que? Isso torna mais fácil – especialmente para clientes – contribuir para sua definição e revisão. Isso diminui o risco de perder a referência. Este pode parecer um comentário inoportuno, mas é importante. Pesquisadores apresentaram métodos complexos de análise que eles entendem, mas que levam uma pessoa de negócios a entrar em coma! Falta de envolvimento do usuário em projetos de software está perto do topo da lista de razões para fracasso de projetos [Larman03], assim, qualquer coisa que ajude a mantê-los envolvidos é realmente desejável.

- Mais motivação (pág. 118)* Casos de uso são uma boa maneira de manter a coisa simples e de tornar possível a especialistas no domínio ou fornecedores de requisitos escrever eles mesmos (ou participar da escrita de) casos de uso.

Outro valor dos casos de uso é que *eles enfatizam os objetivos e perspectivas do usuário*; formulamos a questão “quem está usando o sistema, quais são seus típicos cenários de uso e quais são os seus objetivos?” Essa é uma ênfase mais centrada no usuário, comparada a simplesmente solicitar uma lista de características do sistema.

Muito tem sido escrito sobre casos de uso e, apesar de valer a pena, pessoas criativas freqüentemente obscurecem uma idéia simples com camadas de sofisticação ou supercomplicação. Geralmente, é possível encontrar um modelador de casos de uso novato (ou um analista sério tipo A) que se preocupa em excesso com problemas secundários, como diagramas de casos de uso, relacionamento entre casos de uso, pacotes de casos de uso e etc., em vez de se concentrar no trabalho árduo de simplesmente *escrever as narrativas em texto*.

Dito isso, um dos pontos fortes do mecanismo de casos de uso é a sua capacidade de escalabilidade para cima ou para baixo em termos de sofisticação e formalidade.

6.5 Definição: os casos de uso são requisitos funcionais?

- FURPS+ (págs. 82-83)* Casos de uso *são* requisitos; primeiramente, eles são requisitos funcionais ou comportamentais que indicam o que o sistema fará. Em termos dos tipos de requisitos FURPS+, eles enfatizam o “F” (funcional ou comportamental), mas também podem ser usados para definir outros tipos de requisitos, especialmente aqueles outros tipos fortemente relacionados com um caso de uso. No PU – e em muitos métodos modernos – casos de uso são o mecanismo central recomendado para a descoberta e definição dos requisitos.

Um ponto de vista relacionado é que um caso de uso define um *contrato* de como um sistema se comportará [Cockburn01].

Sendo claro: casos de uso são realmente requisitos (embora não sejam todos os requisitos). Alguns pensam em requisitos somente como “o sistema deverá fazer...” listas de funções ou características. Não é bem assim, e uma idéia chave por trás dos casos de uso é (geralmente) reduzir a importância ou uso das antigas listas detalhadas de características e, em vez disso, redigir casos de uso para os requisitos funcionais. Em uma seção posterior falaremos mais sobre esse ponto.

6.6 Definição: quais são os três tipos de atores?

Um **ator** é qualquer coisa com um comportamento, inclusive o próprio sistema em discussão (SeD) quando invoca os serviços de outros sistemas². Atores principais e de suporte aparecerão nos passos de ação do texto do caso de uso. Atores não são somente papéis desempenhados por pessoas, mas também por organizações, softwares e máquinas. Há três tipos de atores externos em relação ao SeD:

- **Autor principal** – tem objetivos de usuário satisfeitos por meio do uso dos serviços do SeD. Por exemplo, o caixa.
 - Por que identificar? Para encontrar objetivos de usuário, que guiam os casos de uso.
- **Autor de suporte** – fornece um serviço (por exemplo, informações) para o SeD. O serviço automatizado de autorização de pagamento é um exemplo. Frequentemente é um sistema de computador, mas pode ser uma organização ou pessoa.
 - Por que identificar? Para esclarecer interfaces externas e protocolos.
- **Autor de bastidor** – tem interesse no comportamento do caso de uso, mas não é um autor principal ou de suporte; por exemplo, um órgão governamental responsável por impostos.
 - Por que identificar? Para garantir que *todos* os interesses necessários estejam identificados e satisfeitos. Os interesses de autores de bastidor são, às vezes, sutis ou de fácil esquecimento, a menos que esses autores sejam explicitamente nomeados.

6.7 Notação: quais são os três formatos comuns de casos de uso?

Casos de uso podem ser escritos em diferentes formatos e níveis de formalidade:

*Exemplo
(pág. 89)*

- **resumido** – resumo sucinto de um parágrafo, geralmente o cenário de sucesso principal. O exemplo precedente *Processar Venda* foi resumido.
 - Quando? Durante a análise de requisitos inicial, para obter uma rápida idéia do assunto e escopo. Pode levar apenas alguns minutos para criar.
- **informal** – formato informal de parágrafos. Múltiplos parágrafos que cobrem vários cenários. O exemplo precedente *Tratar Devoluções* foi informal.
 - Quando? Como acima.
- **completo** – Todos os passos e variantes são escritos em detalhe e há seções de suporte, como pré-condições e garantias de sucesso.

*Exemplo
(pág. 89)*

² Isso é um refinamento e uma melhoria em relação às definições alternativas de atores, inclusive àquelas das versões iniciais da UML e do PU [Cockburn97]. As versões mais antigas excluíam de forma inconsistente o SeD como ator, mesmo quando invocava serviços de outros sistemas. Todas as entidades podem desempenhar múltiplos papéis, inclusive o SeD.

Mais sobre a ocasião para escrever casos de uso (pág. 121)

- Quando? Depois que muitos casos tiverem sido identificados e escritos em formato resumido, então durante o primeiro workshop de requisitos, alguns (como por exemplo, 10%) dos casos de uso arquiteturalmente significativos e de alto valor, são escritos em detalhe.

O exemplo seguinte é um caso completo para o nosso estudo de caso ProxGer.

6.8 Exemplo: processar venda em estilo completo

Casos de uso completos mostram mais detalhes e são estruturados; eles vão mais fundo.

Na análise de requisitos iterativa e evolutiva do PU, 10% dos casos de uso críticos seriam escritos desse modo durante o primeiro workshop de requisitos. O projeto e a programação começam nos casos de uso ou cenários mais significativos arquiteturalmente desse conjunto de 10%.

Vários gabaritos de formatos estão disponíveis para casos de uso detalhados. Provavelmente, o formato mais amplamente usado e compartilhado, desde o início da década de 1990, seja o gabarito disponível na Web em alistair.cockburn.us, criado por Alistair Cockburn, o autor do livro e abordagem mais popular para modelagem de casos de uso. O exemplo a seguir ilustra este estilo.

Primeiramente, aqui está o gabarito:

Seção do Caso de Uso	Comentário
Nome do Caso de Uso	Começar com um verbo.
Escopo	O sistema em projeto.
Nível	“Objetivo do usuário” ou “sub-função”.
Ator principal	Chama o sistema para fornecer os serviços.
Interessados e interesses	Quem se importa com este caso de uso e o que eles desejam?
Pré-condições	O que precisa ser verdade de início e vale a pena dizer ao leitor?
Garantia de sucesso	O que precisa ser verdade quando da finalização bem sucedida e se vale a pena dizer ao leitor.
Cenário de sucesso principal	Um caminho típico, incondicional e otimista do cenário de sucesso.
Extensões	Cenários alternativos de sucesso ou fracasso.
Requisitos especiais	Requisitos não funcionais relacionados.
Lista de variantes tecnológicas e de dados	Métodos de entrada e saída e formatos de dados variáveis.
Freqüência de ocorrência	Influencia a investigação, teste e oportunidade da implementação.
Diversos	Como, por exemplo, pontos em aberto.

Segue um exemplo baseado no gabarito.

Observe, por favor, que este é o exemplo de um caso de uso detalhado do principal estudo de caso do livro; mostra muitos elementos e pontos comuns.

Ele mostra, provavelmente, muito mais do que você jamais desejou saber sobre um sistema PDV! Mas ele é para um PDV real e mostra a habilidade dos casos de uso de captar requisitos complexos do mundo real e cenários profundamente ramificados.

Caso de uso CDU1: processar venda

Escopo: Aplicação PDV ProxGer

Nível: objetivo do usuário

Autor Principal: Caixa

Interessados e Interesses:

- Caixa: deseja entrada de pagamento rápida, precisa e sem erros, pois a falta de dinheiro na gaveta do caixa será deduzida do seu salário.
- Vendedor: deseja comissões sobre vendas atualizadas.
- Cliente: deseja comprar, receber um serviço rápido e com o mínimo esforço. Deseja a exibição, facilmente visível, dos itens e preços inseridos. Deseja um comprovante da compra, necessário no caso de devoluções de mercadorias.
- Empresa: deseja registrar precisamente as transações e satisfazer aos interesses do cliente. Quer garantir que os pagamentos a receber do Serviço de Autorização de Pagamentos sejam registrados. Deseja algum tipo de proteção contra falhas para permitir que as vendas sejam capturadas mesmo se os componentes do servidor (por exemplo, validação remota de crédito) se encontrarem indisponíveis. Deseja uma atualização automática e rápida da contabilidade e do estoque.
- Gerente: deseja poder realizar rapidamente operações de correção e facilmente corrigir os problemas do Caixa.
- Órgãos fiscais governamentais: desejam cobrar os impostos de cada venda. Podem estar envolvidos vários órgãos, como, por exemplo, federais, estaduais e municipais.
- Serviço de autorização de pagamentos: deseja receber solicitações de autorização digital no formato e protocolo corretos. Deseja contabilizar com precisão seus débitos a pagar para a loja.

Pré-Condições: Caixa está identificado e autenticado.

Garantia de Sucesso (ou Pós-Condições): Venda foi salva. Impostos foram corretamente calculados. Contabilidade e Estoque foram atualizados. Comissões foram registradas. Recibo foi gerado. Autorizações de pagamento foram registradas.

Cenário de Sucesso Principal (ou Fluxo Básico):

1. Cliente chega à saída do PDV com bens ou serviços para adquirir.
2. Caixa começa uma nova venda.
3. Caixa insere o identificador do item.
4. Sistema registra a linha de item da venda e apresenta uma descrição do item, seu preço e total parcial da venda. Preço calculado segundo um conjunto de regras de preços.

Caixa repete os passos 3 e 4 até que indique ter terminado.

5. Sistema apresenta o total com impostos calculados.

6. Caixa informa total ao Cliente e solicita pagamento.
7. Cliente paga e Sistema trata pagamento.
8. Sistema registra venda completada e envia informações de venda e pagamento para Sistema externo de contabilidade (para contabilidade e comissões) e para Sistema de Estoque (para atualizar o estoque).
9. Sistema apresenta recibo.
10. Cliente vai embora com recibo e mercadorias (se houver).

Extensões (ou Fluxos Alternativos):

*a. A qualquer momento, Gerente solicita uma operação de correção:

1. Sistema entra no modo Autorizado pelo Gerente.
2. Gerente ou Caixa realiza uma das operações do modo Gerente, por exemplo, modificação do saldo em dinheiro, retoma uma venda suspensa em outro registrador, anula uma venda, etc.
3. Sistema reverte para o modo Autorizado pelo Caixa.

*b. A qualquer momento, Sistema falha:

Para fornecer suporte à recuperação e à correta contabilidade, garanta que todos os estados e os eventos sensíveis das transações possam ser recuperados a partir de qualquer passo do cenário.

1. Caixa reinicia Sistema, registra-se e solicita a recuperação do estado anterior.
2. Sistema restaura estado anterior.

2a. Sistema detecta anomalias que impedem a restauração:

1. Sistema avisa Caixa sobre erro, registra o erro e, então, entra em um novo estado consistente.
2. Caixa começa uma nova venda.

1a. Cliente ou Gerente indica a retomada de uma venda suspensa.

1. Caixa realiza a operação retomada e insere a Identidade para recuperar a venda.
2. Sistema mostra o estado da venda retomada, com subtotal.

2a. Venda não encontrada

1. Sistema avisa Caixa sobre o erro.
2. Caixa, provavelmente, começa nova venda e re-insere todos os itens.

3. Caixa continua a venda (provavelmente inserindo mais itens ou tratando o pagamento).

2-4a. Cliente diz ao Caixa que tem uma condição de isenção de imposto (por exemplo, idoso, cidadão local)

1. Caixa verifica e depois insere o código de condição de isenção de imposto.
2. Sistema registra a condição (que vai usar durante os cálculos de imposto).

3a. ID do item inválido (não encontrado no sistema):

1. Sistema avisa o erro e rejeita a entrada.
2. Caixa responde ao erro:

2a. Existe um ID do item legível a uma pessoa humana (por exemplo, CUP numérico):

1. Caixa insere manualmente ID do item.
2. Sistema mostra descrição e preço.

2a. ID do item inválido: sistema avisa erro. Caixa tenta método alternativo.

2b. Não existe identificador de item, mas existe um preço na etiqueta:

1. Caixa solicita ao Gerente executar operação de correção.
2. Gerente realiza correção.

3. Caixa indica entrada manual de preço, insere preço e solicita imposto padrão para essa quantia (como não existe informação de produto, o calculador de imposto não pode inferir como calculá-lo).
 - 2c. Caixa invoca Procurar Ajuda de Produto a fim de obter o ID e preço reais do item.
 - 2d. Caso contrário, Caixa pergunta a um empregado da empresa o preço e identificador reais do item e executa a introdução manual do identificador ou a introdução manual do preço (ver acima).
 - 3b. Existem vários itens do mesmo tipo e rastrear um item físico individual não é importante (por exemplo, 5 pacotes de sanduíches naturais):
 1. Caixa pode inserir o identificador do tipo do item e quantidade.
 - 3c. Item exige a introdução manual do tipo e preço (como por exemplo, flores ou cartões com etiqueta de preço)
 1. Caixa insere código de tipo manual especial, mais preço.
- 3-6a: Cliente pede ao Caixa para remover (isto é, anular) um item da compra: isso só é possível se o valor do item é menor que o limite de anulação do Caixa, caso contrário é necessária correção do Gerente.
1. Caixa insere o identificador do item a ser removido da venda.
 2. Sistema remove o item e exibe o total parcial atualizado.
 - 2a. Preço do item excede o limite de anulação do Caixa:
 1. Sistema avisa que houve erro e sugere correção do Gerente.
 2. Caixa solicita correção do Gerente e, após obtê-la, repete a operação.
- 3-6b. Cliente diz ao Caixa para cancelar a venda:
1. Caixa cancela venda no Sistema.
- 3-6c. Caixa suspende a venda:
1. Sistema registra a venda de forma que ela fique disponível para acesso a partir de qualquer terminal PDV.
 2. Sistema apresenta "recibo de suspensão" que inclui as linhas de item e uma identificação da venda usada para recuperar e restaurar a venda.
- 4a. Preço do item gerado pelo Sistema não é desejado (por exemplo, Cliente se queixa de que algo está sendo oferecido a um preço mais baixo):
1. Caixa solicita a aprovação do Gerente.
 2. Gerente realiza operação de correção.
 3. Caixa insere a correção manual de preço.
 4. Sistema apresenta novo preço.
- 5a. Sistema detecta uma falha na comunicação com o serviço externo de cálculo de impostos:
1. Sistema reinicia esse serviço no nó do PDV e continua.
 - 1a. Sistema detecta que o serviço não reinicia.
 1. Sistema avisa o erro.
 2. Caixa pode calcular e inserir manualmente o imposto ou cancelar a venda.
- 5b. Cliente diz que tem direito a um desconto (por exemplo, empregado ou cliente preferencial):
1. Caixa avisa sobre uma solicitação de desconto.
 2. Caixa insere a identificação do Cliente.
 3. Sistema apresenta o total do desconto, com base nas regras para descontos.
- 5c. Cliente diz que tem um crédito na sua conta que pode ser usado para pagar a compra:
1. Caixa avisa sobre uma solicitação de crédito.
 2. Caixa insere a identificação do Cliente.

- 3. Sistema aplica o crédito até que preço=0 e reduz o crédito remanescente.
- 6a. Cliente diz que pretendia pagar com dinheiro, mas não tem dinheiro suficiente:
 - 1. Caixa solicita um método alternativo de pagamento.
 - 1a. Cliente diz ao Caixa para cancelar a venda. Caixa cancela a venda no Sistema.
- 7a. Pagamento em dinheiro:
 - 1. Caixa insere quantia de dinheiro fornecida.
 - 2. Sistema apresenta valor do troco e libera gaveta de dinheiro.
 - 3. Caixa deposita dinheiro fornecido e entrega troco para Cliente.
 - 4. Sistema registra pagamento em dinheiro.
- 7b. Pagamento a crédito:
 - 1. Cliente insere as informações de sua conta de crédito.
 - 2. Sistema mostra seu pagamento para verificação.
 - 3. Caixa confirma.
 - 3a. Caixa cancela o passo de pagamento:
 - 1. Sistema reverte para o modo “introdução de item”.
 - 4. Sistema envia a solicitação de autorização de pagamento para um sistema externo de serviço de Autorização de Pagamento e solicita sua aprovação.
 - 4a. Sistema detecta uma falha ao tentar colaborar com o sistema externo:
 - 1. Sistema avisa erro ao Caixa.
 - 2. Caixa pede ao Cliente uma forma de pagamento alternativa.
 - 5. Sistema recebe aprovação do pagamento, Sistema avisa ao Caixa da aprovação e libera a gaveta de dinheiro (para a inserção do recibo de pagamento a crédito assinado).
 - 5a. Sistema recebe rejeição do pagamento:
 - 1. Sistema avisa rejeição ao Caixa.
 - 2. Caixa solicita ao Cliente uma forma alternativa de pagamento.
 - 5b. Esgotado o tempo de espera da resposta.
 - 1. Sistema avisa ao Caixa o esgotamento.
 - 2. Caixa pode tentar novamente ou solicitar ao cliente uma forma de pagamento alternativa.
 - 6. Sistema registra o pagamento a crédito, que inclui a aprovação do pagamento.
 - 7. Sistema apresenta o mecanismo para entrada de assinatura do pagamento a crédito.
 - 8. Caixa solicita ao Cliente uma assinatura para pagamento a crédito. Cliente fornece a assinatura.
 - 9. Se assinatura em recibo de papel, Caixa coloca recibo na gaveta de dinheiro e a fecha.
- 7c. Pagamento em cheque...
- 7d. Pagamento com débito em conta...
- 7e. Caixa cancela o passo de pagamento:
 - 1. Sistema reverte ao modo “introdução de itens”.
- 7f. Cliente apresenta cupons:
 - 1. Antes de tratar o pagamento, Caixa registra cada cupom e Sistema reduz o preço conforme estabelecido. Sistema registra os cupons usados por razões contábeis.
 - 1a. Cupom inserido não serve para quaisquer dos itens comprados:
 - 1. Sistema avisa erro ao Caixa.
- 9a. Existem descontos de preços específicos para certos produtos:
 - 1. Sistema apresenta os formulários de descontos e os recibos de descontos para cada item ao qual se aplica um desconto.

- 9b. Cliente solicita o recibo especial para presente (os campos de preços não ficam visíveis):
1. Caixa solicita o recibo para presente e Sistema o apresenta.
- 9c. Impressora não tem papel:
1. Se Sistema detecta a falha, avisará sobre o problema.
2. Caixa repõe papel.
3. Caixa solicita outro recibo.

Requisitos Especiais:

- Interface de Usuário (IU) por tela sensível ao toque em um monitor de tela plana grande. O texto deve ser visível à distância de um metro.
- Resposta de autorização de crédito dentro de 30 segundos em 90% do tempo.
- De alguma forma, queremos uma recuperação robusta quando o acesso aos serviços remotos, tal como o sistema de estoque, estiver falhando.
- Internacionalização de linguagem no texto exibido.
- Regras de negócio "plugáveis" que podem ser inseridas nos passos 3 e 7.
- ...

Lista de Variantes Tecnológicas e de Dados:

- *a. Correção de Gerente inserida pela passagem de um cartão de correção numa leitora de cartão ou inserindo um código de autorização através do teclado.
- 3a. Identificador de item inserido por leitora a laser de código de barras (quando o item tiver código de barras) ou pelo teclado.
- 3b. Identificador de item pode ser qualquer um dos seguintes esquemas de codificação: CUP (UCP), EAN, JAN ou SKU.
- 7a. Informação sobre a conta de crédito inserida por leitora de cartão ou pelo teclado.
- 7b. Assinatura de pagamento a crédito captada em recibo de papel. No entanto, prevenimos que dentro de dois anos muitos clientes vão desejar a captação digital de assinatura.

Freqüência de Ocorrência: Poderia ser quase contínuo.

Problemas em Aberto:

- Quais são as variações das leis de impostos?
- Deve-se explorar o problema de recuperação do serviço remoto.
- Qual personalização é necessária para diferentes negócios?
- Um Caixa deve levar seu porta-notas da gaveta de dinheiro quando ele sai do sistema?
- O Cliente pode usar diretamente o leitor de cartão ou é o Caixa que deve fazê-lo?

Este caso de uso é ilustrativo e não pretende ser exaustivo (embora seja baseado nos requisitos de um sistema PDV real – desenvolvido com projeto OO em Java). No entanto, é suficientemente detalhado e complexo para oferecer um senso de realismo, de modo que um caso de uso completo possa registrar muitos detalhes de requisitos. Este exemplo servirá de modelo para muitos problemas de casos de uso.

6.9 Qual é o significado das seções?

Elementos do prefácio

Escopo

O escopo limita o sistema (ou sistemas) em projeto. Tipicamente, um caso de uso descreve o uso de um sistema de software (ou de hardware e software); nesse caso, ele é conhecido como um **caso de uso de sistema**. Num escopo mais amplo, os casos de uso podem também descrever como um negócio é usado por seus clientes e sócios. Tal descrição de processo no nível de empresa é chamado de **caso de uso de negócio** e é um bom exemplo da ampla aplicabilidade de casos de uso, mas eles não são abordados neste livro introdutório.

Nível

PEN (págs. 113-114) No sistema de Cockburn, os casos de uso são classificados como estando no nível de objetivo do usuário ou no nível de subfunção, entre outros. Um caso de uso no **nível de objetivo do usuário** é o tipo comum que descreve os cenários para atingir os objetivos de um ator principal para conseguir que um trabalho seja feito; corresponde aproximadamente a um **processo elementar de negócio** (PEN) na engenharia de processos de negócio. Um caso de uso no **nível de subfunção** descreve subpassos necessários para apoiar um objetivo de usuário e é geralmente criado para individualizar subpassos duplicados compartilhados por diversos casos de uso regulares (para evitar duplicar texto comum); um exemplo é o caso de uso de subfunção *Pagar a Crédito*, que pode ser compartilhado por muitos casos de uso regulares.

Ver o relacionamento “inclui” entre casos de uso para mais informação sobre casos de uso de sub-função (págs. 499-500)

Autor principal

O autor principal que procura os serviços do sistema para atingir um objetivo.

Lista de interessados e interesses – Importante!

Essa lista é mais importante e prática do que pode, a princípio, parecer. Ela sugere e limita o que o sistema deve fazer. Citando um exemplo:

O [sistema] opera um contrato entre os interessados, com os casos de uso detalhando as partes comportamentais desse contrato... O caso de uso, como contrato para o comportamento, capta *todos e somente* os comportamentos relacionados à satisfação dos interesses dos interessados⁴ [Cockburn01].

Isso responde a questão: o que deve constar no caso de uso? A resposta: o que satisfaz a todos os interessados. Além disso, começando com os interessados e seus interesses antes de redigir o resto do caso de uso, temos um método para nos lembrar quais deveriam ser as responsabilidades mais detalhadas do sistema. Por exemplo, eu teria identificado uma responsabilidade para tratar a comissão de um vendedor se não tivesse primeiro listado o interessado vendedor e seus interesses? Espero que

⁴ N. de R.T.: Os interessados em um projeto (stakeholders) são as pessoas da organização que têm algum tipo de envolvimento direto ou indireto com o sistema, como usuários, gerentes, clientes, patronos e financiadores.

sim, mas talvez não tivesse notado isso durante a primeira sessão de análise. O ponto de vista do interessado fornece um procedimento abrangente e prático para descobrir e registrar todos os comportamentos requeridos.

Interessados e Interesses:

- Caixa: deseja entrada rápida, precisa e sem erros de pagamento, tal como falta de dinheiro na gaveta do caixa, pois esta será deduzida do seu salário.
- Vendedor: deseja comissões atualizadas sobre as vendas.
- ...

Pré-condições e garantias de sucesso (pós-condições)

Primeiro, não se importe com uma pré-condição ou garantia de sucesso, a menos que você esteja declarando algo importante e não óbvio para ajudar o leitor a ganhar esclarecimento. Não adicione ruído inútil ao documento de requisitos.

Pré-condições declaram o que *deve sempre* ser verdadeiro antes de iniciar um cenário do caso de uso. As pré-condições *não* são testadas dentro do caso de uso; ao contrário, são condições assumidas como sendo verdadeiras. Em geral, uma pré-condição implica o cenário de um outro caso de uso, por exemplo, registrar-se no sistema, que tenha sido completado com sucesso. Note que há condições que devem ser verdadeiras, mas que não há valor prático em escrevê-las, tal como “o sistema está ligado”. As pré-condições comunicam pressupostos importantes que o autor do caso de uso julga que os leitores devem conhecer.

Garantias de sucesso (ou pós-condições) declaram o que deve ser verdadeiro quando da bem-sucedida conclusão do caso de uso – seja o cenário de sucesso principal ou algum outro caminho alternativo. A garantia deve atender às necessidades de todos os interessados.

Pré-condições: o Caixa está identificado e autenticado.

Garantia de sucesso (pós-condições): a Venda foi salva. Os Impostos foram corretamente calculados. A Contabilidade e o Estoque foram atualizados. As Comissões foram registradas. O Recibo foi gerado.

Cenário e passos de sucesso principal (ou Fluxo Básico)

Este cenário também é chamado de cenário do “caminho correto” ou, mais prosaicamente, “Fluxo Básico” ou “Fluxo Típico”. Descreve um caminho típico de sucesso que satisfaz os interesses dos interessados. Observe que ele, em geral, *não* contém nenhuma condição ou desvio. Embora não seja errado ou ilegal, é discutivelmente mais compreensível e extensível ser bastante consistente e deixar todo o tratamento condicional para a seção de Extensões.

Diretriz

Adie todas as instruções condicionais e de desvio para a seção de Extensões.

O cenário registra os passos, dos quais temos três tipos:

1. Uma interação entre atores³.
2. Uma validação (geralmente feita pelo sistema).
3. Uma mudança de estado pelo sistema (por exemplo, registrar ou modificar algo).

O passo 1 de um caso de uso nem sempre se enquadra nessa classificação, mas ele indica o evento “gatilho” que dá início ao cenário.

É um estilo comum sempre começar os nomes dos atores com maiúsculas, para facilidade de identificação. Observe também o estilo que é usado para indicar repetição.

Cenário de Sucesso Principal:

1. Cliente chega à saída de um PDV com itens a adquirir.
 2. Caixa começa uma nova venda.
 3. Caixa insere o identificador de item.
 4. ...
- Caixa repete os passos 3 e 4 até que indique ter terminado.
5. ...

Extensões (ou fluxos alternativos)

Extensões são importantes e normalmente abrangem a maioria do texto. Elas indicam todos os outros cenários ou ramos, tanto de sucesso como de fracasso. Observe, no exemplo completo, que a seção Extensões era consideravelmente mais longa e complexa do que a seção do Cenário de Sucesso Principal; isso é comum.

Na redação completa de casos de uso, a combinação entre caminho correto e cenários de extensão deve satisfazer a “quase” todos os interesses dos interessados. Esse “quase” significa que alguns interesses podem ser melhor captados como requisitos não funcionais expressos nas Especificações Suplementares, em vez de nos casos de uso. Por exemplo, o interesse dos clientes em uma exibição visível das descrições e preços é um requisito de usabilidade.

Cenários de extensão são ramos do cenário de sucesso principal e, assim, podem ser anotados com relação a seus passos 1...N. Por exemplo, no Passo 3 do cenário de sucesso principal pode haver um identificador inválido, ou porque ele foi inserido incorretamente, ou porque é desconhecido do sistema. Uma extensão é rotulada como “3a”; ela primeiramente identifica a condição e, então, a resposta. Extensões alternativas ao Passo 3 são rotuladas como “3b” e assim por diante.

Extensões:

- 3a. Identificador inválido:
1. Sistema informa o erro e rejeita a entrada.
- ...

³ Note que o próprio sistema em discussão deve ser considerado um ator quando desempenha um papel de colaborador com outros sistemas.

- 3b. Existem vários itens do mesmo tipo e rastrear um item físico individual não é importante (por exemplo, 5 pacotes de sanduíches naturais):
1. Caixa pode inserir o identificador do tipo de item e a quantidade.
 - ...

Uma extensão tem duas partes: a condição e o tratamento.

Diretriz: Quando possível escreva a condição como algo que pode ser detectado pelo sistema ou por um ator. Compare:

- 5a. Sistema detecta uma falha ao se comunicar com o serviço do sistema externo de cálculo de imposto:
- 5a. Sistema externo de cálculo de imposto não está funcionando:

O primeiro estilo é preferível visto que o sistema pode detectar; o segundo é uma inferência.

O tratamento de extensão pode ser resumido em um passo, ou incluir uma sequência de passos, como no exemplo a seguir, o qual também ilustra a notação para indicar que uma condição pode ocorrer dentro de um intervalo de vários passos:

- 3-6a: Cliente pede ao Caixa para remover um item da compra:
1. Caixa insere o identificador do item a ser removido.
 2. Sistema exibe o total parcial atualizado.

Ao fim do tratamento de uma extensão, por default o cenário se insere de volta no cenário de sucesso principal, a menos que a extensão indique diferente (tal como indicar uma parada do sistema).

Às vezes, um ponto específico de extensão é muito complexo como, por exemplo, na extensão “pagamento a crédito”. Isso pode ser uma motivação para expressar a extensão como um caso de uso separado.

Este exemplo de extensão também mostra a notação para expressar fracassos dentro de extensões.

- 7b Pagamento a crédito:
1. Cliente insere as informações de sua conta de crédito.
 2. Sistema envia solicitação de autorização de pagamento a um Sistema externo de Serviço de Autorização de Pagamento e solicita a aprovação do pagamento.
 - 2a. Sistema detecta uma falha ao tentar colaborar com o sistema externo:
 1. Sistema avisa ao Caixa que houve erro.
 2. Caixa pede ao Cliente uma forma de pagamento alternativa.

Se for desejável descrever uma condição de extensão como possível durante qualquer passo (ou pelo menos na maioria deles), podem ser usados os rótulos *a, *b, etc.

***a. A qualquer momento, Sistema falha:**

Para fornecer suporte à recuperação e à correta contabilidade, garanta que todos os estados e eventos sensíveis das transações possam ser recuperados a partir de qualquer passo do cenário.

1. Caixa reinicia o Sistema, registra-se e solicita a recuperação do estado anterior.
2. Sistema restaura o estado anterior.

Realização de outro cenário de caso de uso

Algumas vezes, um caso de uso desvia para percorrer outro cenário de caso de uso. Por exemplo, a estória *Procurar Ajuda sobre o Produto* (para mostrar detalhes de produto tais como: descrição, preço, uma figura ou vídeo, etc.) é um caso de uso distinto algumas vezes invocado quando se está dentro de *Processar Venda* (usualmente, quando o ID do item não pode ser encontrado.). Na notação de Cockburn, a invocação desse segundo caso de uso é indicada sublinhando o seu nome, como mostra o exemplo a seguir:

3a. ID do item inválido (não encontrado no sistema):

1. Sistema avisa o erro e rejeita a entrada.

2. Caixa responde ao erro:

2a. ...

- 2c** Caixa invoca Procurar Ajuda de Produto a fim de obter o ID e preço reais do item.

Considerando, como é usual, que casos de uso são escritos com uma ferramenta com hiperligações (hiperlinks), clicar nesse nome sublinhado de caso de uso irá mostrar o seu texto.

Requisitos especiais

Se um requisito não funcional, um atributo de qualidade ou uma restrição se relaciona especificamente com um caso de uso, registe-o com esse caso de uso. Entre esses atributos, temos requisitos de qualidade, tais como desempenho, confiabilidade, usabilidade e restrições de projeto (freqüentemente relativas a dispositivos de E/S) que foram impostas ou consideradas prováveis.

Requisitos especiais:

- Interface de Usuário (IU) por tela sensível ao toque em um monitor de painel plano grande. O texto deve ser visível à distância de um metro.
- Resposta de autorização de crédito dentro de 30 segundos em 90% do tempo.
- Internacionalização de linguagem no texto exibido.
- Regras de negócios “plugáveis” que podem ser inseridas nos passos 3 e 7.

Registrar esses requisitos com os casos de uso é um conselho clássico do PU e essa é uma localização razoável quando da redação do caso de uso pela *primeira vez*. Contudo, muitos profissionais acham útil, no fim, consolidar todos os requisitos não funcionais nas Especificações Suplementares, para efeito de gestão de conteúdo, abran-

gência e legibilidade, porque esses requisitos, geralmente, devem ser considerados como um todo durante a análise arquitetural.

Listas de variantes tecnológicas e de dados

Freqüentemente, existem variantes técnicas sobre *como* algo deve ser feito, mas não sobre o que deve ser feito, e vale a pena registrar isso no caso de uso. Um exemplo comum é uma restrição técnica imposta por um interessado no projeto com respeito às tecnologias de entrada ou saída. Por exemplo, um interessado no projeto pode dizer que “o sistema PDV deve apoiar a entrada da conta de crédito usando um leitor de cartões e o teclado”. Note que esses são exemplos de decisões de projeto ou de restrições precoces; em geral, é melhor evitar decisões prematuras de projeto, mas, às vezes, elas são óbvias ou inevitáveis, especialmente no que diz respeito às tecnologias de entrada e saída.

É também necessário compreender as variações em esquemas de dados, tais como usar CUPs (UPCs) ou EANs para identificadores de itens, codificados em simbologia de código de barras[†].

Esta lista é o lugar para registrar tais variações. Também é útil registrar as variações nos dados que podem ser captados em um determinado passo.

Parabéns: os casos de uso foram escritos e estão errados!

A equipe do PDV ProxGer está escrevendo alguns casos de uso em vários *workshops* curtos de requisitos, em paralelo com uma série de iterações de desenvolvimento curtas e limitadas em tempo, que envolvem programação e teste com qualidade de produção. A equipe está incrementalmente adicionando ao conjunto de casos de uso e refinando e adaptando com base em realimentação propiciada por programação, testes e demos preliminares. Especialistas na área, caixas e desenvolvedores participam ativamente na análise dos requisitos.

Esse é um bom processo de análise evolutiva – em vez do processo em cascata –, mas uma dose de “realismo nos requisitos” ainda é necessária. Especificações escritas e outros modelos dão a *ilusão* de correção; mas modelos mentem (não intencionalmente). Apenas código e teste revelam de fato o que se deseja e o que funciona.

Os casos de uso, diagramas UML e etc. não serão perfeitos – isso é garantido. Eles não vão conter informação importante, mas sim afirmações erradas. A solução não é a atitude em cascata de tentar registrar especificações quase perfeitas e completas no início – apesar de sem dúvida fazermos o melhor que podemos no tempo disponível e devermos aprender e aplicar grandes práticas de requisitos. Entretanto, isso nunca será suficiente.

Isso não é um apelo para a corrida em direção à codificação sem qualquer análise ou modelagem. Existe um meio termo, entre o cascata e a programação *ad hoc*: desenvolvimento interativo e evolutivo. Nessa abordagem os casos de uso e outros modelos são incrementalmente refinados, verificados e esclarecidos por meio de programação e testes iniciais.

Você sabe que está no caminho errado se a equipe tenta redigir em detalhes todos ou a maioria dos casos de uso antes de iniciar a primeira iteração de desenvolvimento – ou o contrário.

[†] N. de R.T.: Ver Seção 7.4 para detalhes sobre os códigos.

Lista de Variantes Tecnológicas e de Dados:

- 3a. Identificador de item inserido por leitora a laser ou por teclado.
- 3b. Identificador de item pode ser qualquer um dos seguintes esquemas de codificação: CUP (UCP), EAN, JAN ou SKU.
- 7a. Informações sobre a conta de crédito podem ser entradas por leitor de cartões ou pelo teclado.
- 7b. Assinatura de pagamento a crédito capturada em recibo de papel. No entanto, preveemos que, dentro de dois anos, muitos clientes desejarão captura de assinatura digital.

6.10 Notação: existem outros formatos? Uma variante de duas colunas

Alguns preferem o formato de duas colunas ou o formato conversacional, que enfatiza a ocorrência de uma interação entre os atores e o sistema. Esse formato foi proposto inicialmente por Rebecca Wirfs-Brock [Wirfs-Brock93] e também defendido por Constantine e Lockwood, para ajudar na análise e engenharia de usabilidade [CL99]. O mesmo conteúdo é apresentado no formato duas colunas:

Caso de uso CDU1: processar venda

Autor Principal

... como antes...

Cenário de Sucesso Principal:

Ação do Ator (ou Intenção)	Responsabilidade do Sistema
1. Cliente chega à saída de um PDV com bens e/ou serviços para adquirir.	
2. Caixa começa uma nova venda.	
3. Caixa insere o identificador de item.	4. Registra cada linha de venda e apresenta uma descrição do item...e o total parcial da venda.
Caixa repete os passos 3 e 4 até que indique ter terminado	5. Apresenta o total com impostos calculados.
6. Caixa informa o total ao Cliente e solicita o pagamento.	8. Trata o pagamento.
7. Cliente paga.	9. Registra a venda completada e envia as informações para o Sistema externo de contabilidade (para contabilidade e comissões) e para o Sistema de Estoque (para atualizar o estoque). Sistema apresenta recibo.

Qual é o melhor formato?

Não há um melhor formato; alguns preferem o estilo de uma coluna, outros, o estilo de duas colunas. Seções podem ser acrescentadas e removidas; nomes de cabeçalho podem ser mudados. Nada disso é particularmente importante; o mais importante é escrever os detalhes do cenário de sucesso principal e suas extensões de alguma forma. [Cockburn01] resume os muitos formatos utilizáveis.

Prática pessoal

Trata-se da minha prática, não é uma recomendação. Por muitos anos usei o formato de duas colunas por causa de sua separação visual clara na conversação. Entretanto, voltei para o estilo de uma coluna por ser mais compacto e fácil de formatar, e o ligeiro benefício de ter uma conversação visualmente separada não compensa esses outros benefícios. Eu acho ainda mais fácil de identificar visualmente as diferentes partes na conversação (Cliente, Sistema, etc.) se cada parte e as respectivas respostas do Sistema sempre forem colocadas nos seus próprios passos.

6.11 Diretriz: escreva casos de uso em um estilo essencial, independente da IU

Novo e melhorado! O caso das impressões digitais

Durante um workshop de requisitos, o caixa pode dizer que um dos seus objetivos é “registrar-se”. O caixa estava provavelmente pensando em uma interface com o usuário (IU), uma caixa de diálogo, um ID de usuário e uma senha. Esse é um mecanismo para atingir um objetivo e não o objetivo propriamente dito. Investigando mais acima, na hierarquia de objetivos (“qual é o objetivo daquele objetivo?”), o analista de sistemas chega a um objetivo independente de mecanismo: “identificar a si mesmo e ser autenticado pelo sistema”, ou mesmo um objetivo ainda mais alto: “prevenir roubo...”.

Esse processo de descoberta com base na *raiz do objetivo* pode abrir a visão para soluções novas e melhores. Por exemplo, teclados e mouses com leitores biométricos, normalmente usados para a leitura de impressões digitais, agora são comuns e baratos. Se o objetivo é “identificação e autenticação” por que não torná-lo mais fácil e rápido, usando um leitor biométrico no teclado? No entanto, responder adequadamente a essa questão envolve algum trabalho de análise de usabilidade também. Os dedos dos usuários estão sujos de graxa? Os usuários têm dedos?

Estilo essencial de escrita

Essa idéia foi resumida em várias diretrizes para casos de uso como “deixe de fora a interface de usuário; focalize a intenção” [Cockburn01]. Sua motivação e notação foram exploradas ampla e completamente por Larry Constantine no contexto de criar melhores IUs e executar uma engenharia de usabilidade [Constantine94, CL99].

Constantine chama um estilo de escrita de **essencial** quando este evita detalhes da IU e focaliza as reais intenções do usuário⁴.

Em um estilo essencial de redação, a narrativa é expressa no nível da *intenção* do usuário e das *responsabilidades* do sistema e não de suas ações concretas. Elas permanecem independentes de tecnologia e dos detalhes de mecanismos, especialmente aqueles relacionados com a IU.

Diretriz

Escreva casos de uso utilizando um estilo essencial; deixe de lado a interface de usuário e enfoque a intenção do ator.

Todos os exemplos anteriores de casos de uso deste capítulo, tal como *Processar Venda*, foram escritos visando a um estilo de redação essencial.

Exemplos contrastantes

Estilo essencial

Assuma que o caso de uso *Gerenciar Usuários* exija identificação e autenticação

- ...
- 1. O Administrador identifica-se
- 2. O Sistema autentica a identidade
- 3. ...

A solução de projeto para essas intenções e responsabilidades é bastante aberta: leitores biométricos, interfaces gráficas de usuário (GUIs⁵) e assim por diante.

Estilo concreto – evite-o durante o trabalho inicial com requisitos

Em contraste, existe um estilo **concreto de caso de uso**. Nesse estilo, as decisões sobre a interface de usuário estão embutidas no texto do caso de uso. O texto pode até mesmo mostrar imagens de telas com janelas, discutir a navegação entre janelas, a manipulação de elementos de tela (widgets⁶) da GUI, entre outros. Por exemplo:

- ...
- 1. O Administrador insere seu ID e senha na caixa de diálogo (ver Imagem 3).
- 2. O Sistema autentica o Administrador.
- 3. O Sistema exibe a janela “editar usuários” (ver Imagem 4).
- 4....

⁴ O termo provém dos “modelos essenciais” descritos em *Essential Systems Analysis* [MP84].

⁵ N. de R.T.: Utilizaremos a sigla GUI, do inglês *Graphical User Interface*, ao invés da sigla correspondente em português (IGU), por ser amplamente difundida e conhecida na área de computação.

⁶ N. de R.T.: Widgets são os diversos elementos gráficos que podem fazer parte de uma GUI, como, por exemplo, botões, listas, campos para entrada de dados, etc.

Esses casos de uso em estilo concretos[‡] podem ser úteis como auxílio para o trabalho de projeto concreto ou detalhado da GUI em um passo posterior, mas eles não são adequados para o trabalho inicial de especificação dos requisitos. Durante o trabalho inicial de especificação dos requisitos, “deixe de lado a interface de usuário – focalize a sua intenção”.

6.12 Diretriz: escreva casos de uso enxutos

Você gosta de ler montes de requisitos? Eu acho que não. Assim, escreva casos de uso enxutos. Despreze palavras de “ruído”. Mesmo pequenas modificações ajudam, tais como “sistema autentica...” em vez de “o sistema autentica...”.

6.13 Diretriz: escreva casos de uso caixa preta

Casos de uso caixa preta são a espécie mais comum e recomendada; eles não descrevem o funcionamento interno do sistema, seus componentes ou projetos. Em vez disso, o sistema é descrito como tendo *responsabilidades*, que é um tema metafórico comum de unificação no raciocínio orientado a objetos – elementos de software têm responsabilidades e colaboram com outros elementos que também têm responsabilidades.

Definindo as responsabilidades do sistema com casos de uso caixa preta, pode-se especificar *o que* o sistema deve fazer (o comportamento ou requisitos funcionais) sem decidir *como* ele o fará (o projeto). De fato, a definição de “análise” versus “projeto” é algumas vezes resumida como “o que” versus “como”. Esse é um tema importante para o bom desenvolvimento de software: durante a análise dos requisitos evite tomar decisões “como” e especifique o comportamento externo do sistema, como uma caixa preta. Depois, durante o projeto, crie uma solução que satisfaz as especificações.

Estilo Caixa Preta	Não
O sistema registra a venda	O sistema grava a venda numa base de dados... ou (ainda pior): O sistema gera um comando SQL INSERT para a venda...

6.14 Diretriz: considere a perspectiva do ator e do objetivo do ator

Segue a definição do RUP para caso de uso, feita pelo criador dos casos de uso Ivar Jacobson:

Um conjunto de instâncias de casos de uso em que cada instância é uma sequência de ações realizadas por um sistema e que produzem um resultado observável e com valor para um *ator em particular*.

[‡] N. de R.T.: No original, em inglês, menciona-se casos de uso concretos. Julgamos apropriado mencionar como casos de uso em estilo concreto.

A frase “*um resultado observável e com valor para um ator em particular*” é um conceito útil, mas importante, que Jacobson considera crucial, porque enfatiza duas atitudes durante a análise de requisitos:

- Escreva requisitos enfocando os usuários ou atores de um sistema, perguntando sobre os seus objetivos e situações típicas.
- Enfoque no entendimento do que o ator considera como resultado valioso.

Talvez pareça óbvio enfatizar o fornecimento de valor observável para o usuário e enfocar nos objetivos típicos dos usuários, mas a indústria de software está cheia de projetos fracassados que não forneceram o que as pessoas realmente necessitavam. A velha abordagem de lista de características e funções para eliciar requisitos pode contribuir para esse resultado negativo, porque ela não encorajava questionar quem está usando o produto e o que fornece valor.

*Lista de funções
(pág. 118)*

6.15 Diretriz: como encontrar casos de uso

Casos de uso são definidos para satisfazer aos objetivos dos atores principais. Assim, o procedimento básico é:

1. Escolher a fronteira do sistema. Ele é somente uma aplicação de software, é o hardware e a aplicação como uma unidade, é isso e mais uma pessoa usando o sistema ou é toda uma organização?
2. Identificar os atores principais – aqueles que têm objetivos satisfeitos por meio do uso dos serviços do sistema.
3. Identificar os objetivos para cada ator principal..
4. Definir casos de uso que satisfaçam os objetivos dos usuários; nomeie-os de acordo com o objetivo. Geralmente, os casos de uso no nível de objetivo do usuário estarão em uma relação de um-para-um com os objetivos dos usuários, mas existe pelo menos uma exceção que será examinada.

Certamente, em um desenvolvimento iterativo e evolutivo, nem todos os objetivos ou casos de uso vão estar total ou corretamente identificados logo no início. Trata-se de uma descoberta evolutiva.

Passo 1: escolher a fronteira do sistema

Para este estudo de caso, o próprio PDV é o sistema a ser projetado; tudo que estiver fora dele estará fora da fronteira do sistema, incluindo o caixa, serviço de autorização de pagamento, etc.

Se não estiver claro, a definição da fronteira do sistema que está sendo projetado pode ser esclarecida definindo-se o que está de fora – os atores principais e os atores de suporte externos. Uma vez que os atores externos tenham sido identificados, a fronteira se torna mais clara. Por exemplo, a responsabilidade completa pela autorização de pagamento está dentro da fronteira do sistema? Não, existe um ator que é um serviço externo de autorização de pagamento.

Passos 2 e 3: encontrar atores principais e objetivos

É artificial linearizar estritamente a identificação dos atores principais antes dos objetivos de usuário; em um workshop sobre requisitos, as pessoas fazem reuniões especulativas (brainstorms) e geram uma mistura de ambos. Às vezes, os objetivos revelam os atores ou vice-versa.

Diretriz: enfatize reuniões especulativas primeiro, para encontrar os atores principais, pois isso estabelece a estrutura para investigações adicionais.

Existem questões para ajudar a encontrar atores e objetivos?

Além dos atores principais e objetivos óbvios, as seguintes perguntas ajudam a identificar outros que podem ter passado despercebidos:

- Quem ativa e pára o sistema?
- Quem faz a administração de usuários e da segurança?
- Existe um processo de monitoração que reinicia o sistema em caso de falha?
- Como são tratadas as atualizações de software? As atualizações são do tipo forçadas ou negociadas?
- Além dos atores principais *humanos* existe algum sistema externo de software ou robótica que solicita os serviços do sistema?

- Quem faz a administração do sistema?
- O “tempo” é um ator porque o sistema faz algo em resposta a um evento temporal?
- Quem avalia a atividade ou desempenho do sistema?
- Quem avalia os registros? Eles são recuperados remotamente?
- Quem é notificado quando há erros ou falhas?

Como organizar os atores e objetivos?

Existem pelo menos duas abordagens:

Diagrama de caso de uso (págs. 115-116)

1. À medida que você descobre os resultados, desenhe-os em um diagrama de caso de uso, nomeando os objetivos como casos de uso.
2. Escreva primeiro uma lista de atores/objetivos, reveja-a e refine-a, e depois desenhe o diagrama de casos de uso.

Se você criar uma lista de atores/objetivos, então, em termos de artefatos do PU, ela pode ser uma seção no artefato Visão.

Por exemplo:

Ator	Objetivo	Ator	Objetivo
Caixa	processar vendas processar aluguéis tratar devoluções receber pagar ...	Administrador do Sistema	adicionar usuários modificar usuários excluir usuários gerenciar segurança gerenciar as tabelas do sistema ...

Ator	Objetivo	Ator	Objetivo
Gerente	iniciar encerrar ...	Sistema Atividade de Vendas	analisar vendas e dados de desempenho
...

O Sistema Atividade de Vendas é uma aplicação remota que freqüentemente solicitará dados de venda de cada nó do PDV da rede.

Por que perguntar sobre objetivos do ator em vez de casos de uso?

Atores têm objetivos e usam aplicações para ajudar a satisfazê-los. O ponto de vista da modelagem de casos de uso é encontrar esses atores e seus objetivos e criar soluções que produzam um resultado com valor. Isso significa um pequeno desvio e ênfase para o modelador de casos de uso. Em vez de perguntar “quais são as tarefas?” começa-se por perguntar “quem usa o sistema e quais são os seus objetivos?” De fato, o nome de um caso de uso para um objetivo do usuário deveria refletir o seu nome, para enfatizar esse ponto de vista – objetivo: captar ou processar uma venda; caso de uso: *processar venda*.

Assim, existe uma idéia-chave a respeito da investigação de requisitos e casos de uso:

Imagine que estamos juntos em um workshop de requisitos. Poderíamos perguntar:

- “O que você faz?” (Uma questão basicamente orientada à tarefa) ou,
- “Quais são seus objetivos cujos resultados têm valor mensurável?”

Prefira a segunda questão.

As respostas à primeira questão provavelmente refletem as soluções e procedimentos correntes e as complicações a elas associadas.

As respostas à segunda questão, especialmente combinadas com uma investigação para subir na hierarquia de objetivos (“qual é a raiz do objetivo?”) abrem a visão para soluções novas e aperfeiçoadas, enfocam a adição de valor de negócio e chegam ao coração do que os interessados desejam do sistema.

O caixa ou o cliente é o ator principal?

Por que é o caixa, e não o cliente, o ator principal no caso de uso *Processar Venda*?

A resposta depende da fronteira do sistema em projeto e para quem o sistema está sendo principalmente projetado, conforme ilustrado na Figura 6.2. Se enxergarmos a empresa ou o serviço de conclusão da venda (checkout) como um sistema agregado, o cliente é um ator principal, com o objetivo de obter bens ou serviços e ir embora. Todavia, do ponto de vista somente do sistema PDV (o qual é a fronteira escolhida para o sistema deste estudo de caso), o sistema atende ao objetivo de um caixa treinado (e da



Figura 6.2 Atores e objetivos principais em diferentes fronteiras do sistema.

loja) de processar a venda do cliente. Isso considera um ambiente tradicional de conclusão da venda com um caixa, apesar de haver um número crescente de sistemas PDV de auto-conclusão da venda (self-checkout) em operação para uso direto pelo cliente.

O cliente é um ator, mas no contexto do PDV ProxGer, não o principal; ao contrário, o Caixa é o ator principal porque o sistema está sendo projetado para servir principalmente os objetivos de “uso potente” de um caixa treinado (para rapidamente processar uma venda, procurar preços, etc.) O sistema não tem uma IU e funcionalidade que poderia ser usada igualmente pelo cliente ou pelo caixa. Em vez disso, ele está otimizado para satisfazer às necessidades e treinamento de um caixa. Um cliente na frente de um terminal PDV não saberia como usá-lo efetivamente. Em outras palavras, ele foi projetado para o caixa, não para o cliente e, assim, o caixa não é apenas o substituto do cliente.

Por outro lado, considere um site da Web de compra de bilhetes, que é igual para um cliente usar diretamente ou para um agente de vendas por telefone usar quando um cliente chama. Nesse caso, o agente é simplesmente um substituto do cliente – o sistema não está projetado para satisfazer especialmente os objetivos próprios do agente. Assim, mostrar o cliente em vez do agente telefônico como ator principal está certo.

Outros modos de encontrar atores e objetivos? Análise de eventos

Uma outra forma de ajudar a encontrar atores, objetivos e casos de uso é identificar eventos externos. Quais são eles, de onde vem e por que? Freqüentemente, um grupo de eventos pertence ao mesmo caso de uso. Por exemplo:

Evento Externo	Do Ator	Objetivo/Caso de Uso
entrar linha de item de venda	Caixa	processar uma venda
entrar pagamento	Caixa ou Cliente	processar uma venda
...		

Passo 4: definir casos de uso

Em geral, defina um caso de uso para cada objetivo de usuário. Nomeie o caso de uso de modo semelhante ao do objetivo do usuário – por exemplo, Objetivo: processar uma venda; Caso de Uso: *Processar Venda*.

Inicie os nomes dos casos de uso com um verbo.

Uma exceção comum à regra de um caso de uso por objetivo é concentrar os objetivos separados das operações CRUD (Create, Update, Retrieve e Delete – Criar, Recuperar, Atualizar e Remover) em um único caso de uso CRUD, idiomaticamente denotado pela expressão *Gerenciar <X>*. Por exemplo, os objetivos “editar usuário”, “remover usuário” e assim por diante, são todos satisfeitos pelo caso de uso *Gerenciar Usuários*.

6.16 Diretriz: que testes podem ajudar a encontrar casos de uso úteis?

Qual desses é um caso de uso válido?

- Negociar um contrato de fornecimento
- Tratar devoluções
- Registrar-se
- Mover peça no tabuleiro do jogo

Pode-se argumentar que todos esses são casos de uso *em níveis diferentes*, dependendo das fronteiras do sistema, atores e objetivos.

Mas, em vez de perguntar em geral “o que é um caso de uso válido?”, uma questão mais prática é “qual é um nível útil para expressar casos de uso para a análise de requisitos da aplicação?”. Há várias regras práticas, inclusive:

- O teste do chefe
- O teste PEN
- O teste de tamanho

O teste do chefe

Seu chefe pergunta, “O que você ficou fazendo o dia todo?”, você responde: “Registrando-me”. Seu chefe ficou satisfeito?

Em caso negativo, o caso de uso não passou no teste, o que implica que ele não está fortemente relacionado a alcançar resultados de valor mensurável. Ele pode ser um caso de uso em algum nível baixo de objetivo, mas não no nível desejável de ser focalizado para análise de requisitos.

Isso não significa que se deve sempre ignorar os casos de uso que não passam no teste do chefe. A autenticação do usuário pode não passar no teste do chefe, mas pode ser importante e difícil.

O teste PEN

Um **Processo Elementar de Negócio (PEN)** é um termo do campo da engenharia de processos de negócios⁵ definido como:

Uma tarefa realizada por uma pessoa em um lugar e em um instante em resposta a um evento de negócios, que adiciona valor de negócio mensurável e deixa os dados em um estado consistente, por exemplo, Aprovar Crédito ou Atribuir Preços ao Pedido [a fonte original perdeu-se].

Focalize nos casos de uso que refletem PENs.

O Teste PEN é semelhante ao Teste do Chefe, especialmente em termos da qualificação do valor de negócios mensurável.

A definição pode ser levada muito ao pé da letra: o caso de uso não passa como PEN se duas pessoas são necessárias ou se uma pessoa tem que dar uma volta? Provavelmente não, mas o sentimento da definição está praticamente certo. Não é um único pequeno passo como “remover linha de item” ou “imprimir o documento”. Pelo contrário, o principal cenário de sucesso tem provavelmente cinco ou dez passos. Não leva dias e sessões múltiplas como “negociar um contrato de fornecimento”; é uma tarefa realizada durante uma única sessão, leva provavelmente entre alguns minutos e uma hora para ser feito. Tal como a definição do PU, ele enfatiza a adição de valor de negócio observável ou mensurável e chega a um resultado em que o sistema e os dados ficam em um estado estável e consistente.

O teste de tamanho

Um caso de uso é muito raramente uma ação ou passo único; pelo contrário, um caso de uso tipicamente contém vários passos e no formato completo exigirá freqüentemente de três a dez páginas de texto. Um erro comum na modelagem de casos de uso é definir apenas um único passo dentro de uma série de passos relacionados como o caso de uso em si, como, por exemplo, definir um caso de uso chamado *Inserir o ID do item*. Você pode ter uma noção do erro pelo seu pequeno tamanho – o nome do caso de uso vai sugerir erradamente apenas um passo dentro de uma série maior de passos e se você imaginar o tamanho do seu texto completo, ele seria extremamente curto.

⁵ PEN é semelhante ao termo **tarefa de usuário** na engenharia de usabilidade, apesar do significado ser menos estreito naquele domínio.

Exemplo: aplicação dos testes

- Negociar um Contrato de Fornecimento
 - Mais amplo e maior que um PEN. Poderia ser modelado como um caso de uso *de negócio*, em vez caso de uso de sistema.
- Tratar Devoluções
 - Passa no teste do chefe. Parece com um PEN. O tamanho é bom.
- Registrar-se
 - O chefe não fica satisfeito se isso foi tudo o que você fez durante o dia todo!
- Mover Peça no Tabuleiro do Jogo
 - Passo único – não passa no teste de tamanho.

Violações razoáveis dos testes

Ver o relacionamento “Inclui” entre casos de uso para mais informação sobre como ligar sub-funções em casos de uso sub-função (págs. 499-500)

Apesar da maioria dos casos de uso identificados e analisados para uma aplicação deverem satisfazer os testes, exceções são comuns.

Algumas vezes, é útil escrever casos de uso separados, no nível de sub-função, representando sub-tarefas ou passos dentro de um caso de uso no nível PEN regular. Por exemplo, uma sub-tarefa ou extensão tal como “pagamento a crédito” pode ser repetida em vários casos de uso básicos. Nesse caso, é desejável separá-lo em seu próprio caso de uso, apesar desse não satisfazer realmente os testes PEN e de tamanho, e ligá-lo aos vários casos de uso básicos, para evitar duplicação do texto.

Autenticar Usuário pode não passar no teste do Chefe, mas ser suficientemente complexo para exigir cuidadosa análise, tal como para uma característica “assinatura única”.

6.17 Aplicação de UML: diagramas de casos de uso

A UML fornece a notação de diagramas de casos de uso para ilustrar os nomes dos casos de uso e dos atores, bem como os relacionamentos entre eles (ver Figura 6.3)⁶.

Diagramas de caso de uso e relacionamentos entre casos de uso são secundários no trabalho de definição dos casos de uso.

Casos de uso são documentos de texto. Fazer o trabalho de definição de casos de uso significa escrever texto.

⁶ “Abrir o Caixa” é o ato de caixa chegar com uma gaveta removível com dinheiro, registrar-se e informar a quantidade de dinheiro da gaveta.

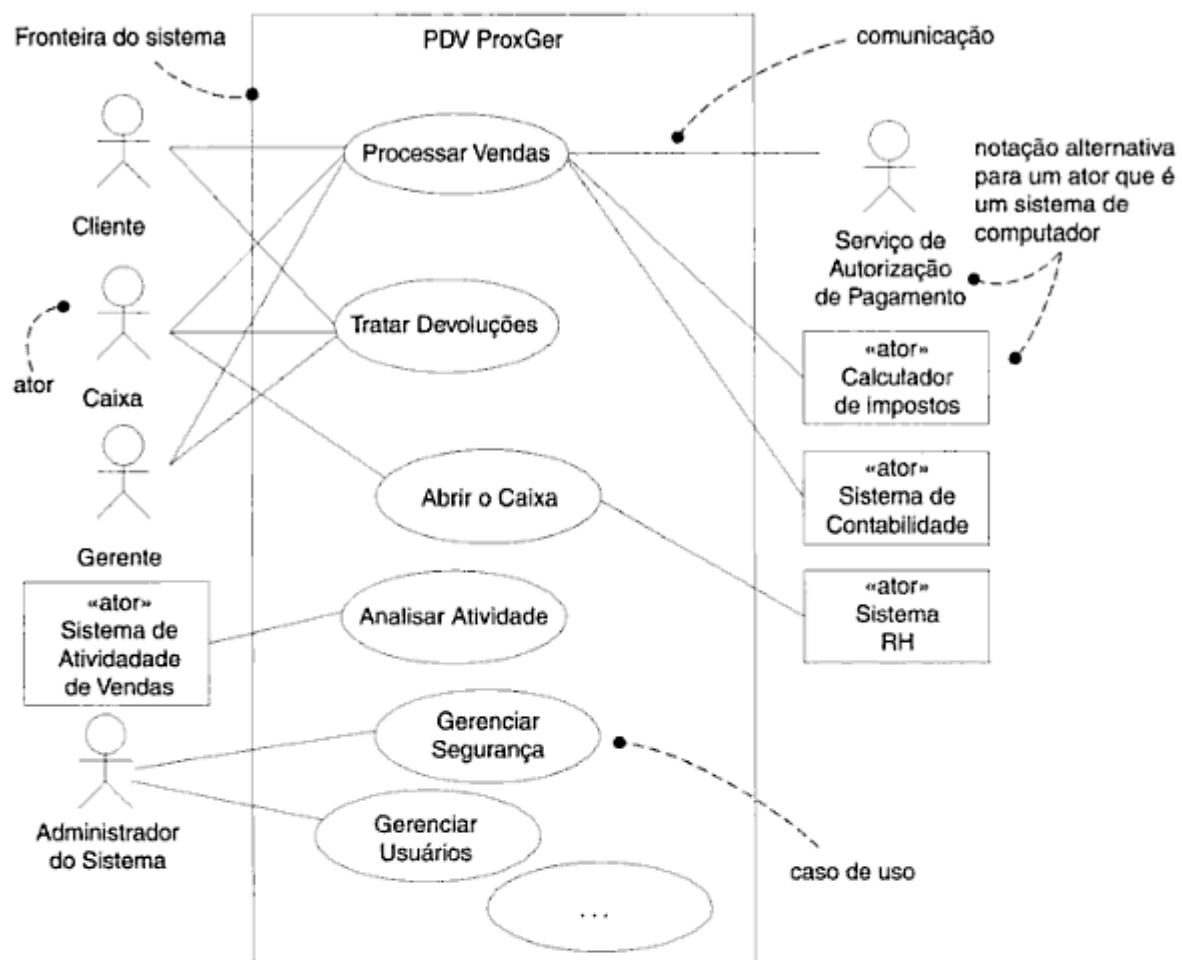


Figura 6.3 Diagrama parcial de contexto dos casos de uso.

Um sinal típico de modeladores de casos de uso iniciantes (ou acadêmicos) é a preocupação com diagramas de casos de uso e seus relacionamentos, em vez de escrever texto. Especialistas em casos de uso mundialmente reconhecidos, como Fowler e Cockburn, entre outros, não se preocupam com os diagramas de casos de usos e seus relacionamentos, concentrando-se, em vez disso, em sua redação. Tendo isso como um alerta, um simples diagrama de caso de uso oferece um diagrama de contexto visual sucinto para o sistema, que ilustra os atores externos e como eles usam o sistema.

Diretriz

Desenhe um diagrama de caso de uso simples em conjunto com uma lista ator/objetivo.

Um diagrama de caso de uso é uma excelente imagem do contexto do sistema; ele é um bom **diagrama de contexto**, ou seja, mostra a fronteira de um sistema, o que está

fora dele e como o sistema é usado. Serve como uma ferramenta de comunicação que resume o comportamento do sistema e seus atores. A Figura 6.3 é uma amostra de um diagrama de caso de uso de contexto *parcial* para o sistema ProxGer.

Diretriz: diagramação

A Figura 6.4 oferece algumas sugestões sobre o diagrama. Note a caixa de ator com o símbolo «ator». Este símbolo é usado para as **palavras-chave** e **estereótipos UML** e incluem aspas horizontais – guillemets – colchetes especiais de um único caractere («ator» e não “<>ator>>”), mais comumente usados na tipografia francesa para indicar uma citação.

Para maior clareza, alguns preferem destacar os atores externos ao sistema de computador com uma notação alternativa, como mostrado na Figura 6.5.

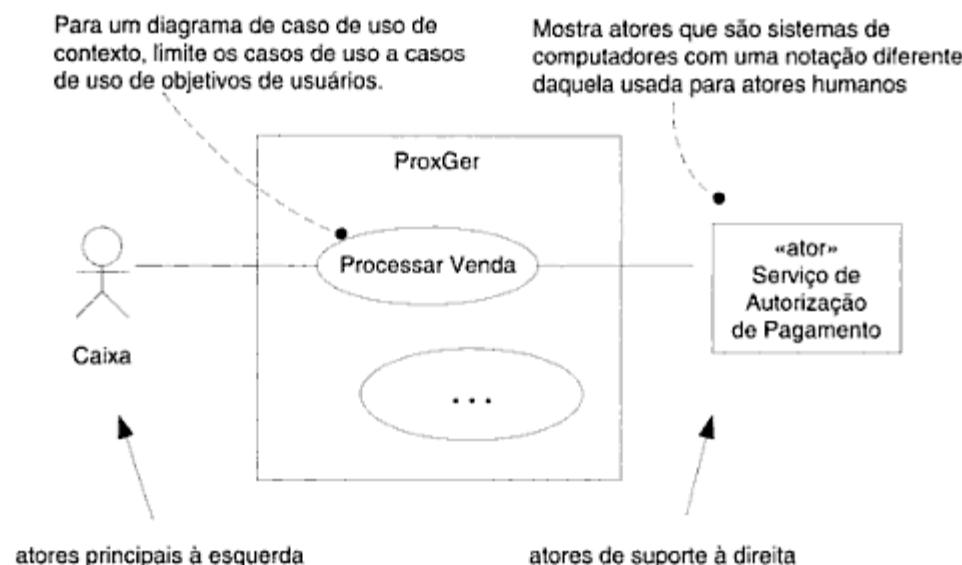


Figura 6.4 Sugestões de notação.



Figura 6.5 Notação alternativa para ator.

Diretriz: diminua a importância dos diagramas, faça-os pequenos e simples

Conforme já foi dito, o trabalho mais importante nos casos de uso é escrever um texto, não diagramar e focalizar os relacionamentos entre os casos de uso. Se uma organização está gastando muitas horas (ou pior ainda, dias) trabalhando em um diagrama de casos de uso e discutindo seus relacionamentos, em vez de se concentrar na redação do texto, o esforço relativo está sendo mal usado.

6.18 Aplicação da UML: diagramas de atividade

Diagramas de Atividade de UML
(pág. 483)

A UML inclui um diagrama útil para visualizar fluxos de trabalho e processos de negócio: o Diagrama de Atividade. Como os casos de uso envolvem a análise de processos e fluxos de trabalho, esses diagramas podem ser uma alternativa ou complemento úteis para escrever o texto do caso de uso, especialmente para casos de uso de negócio que descrevem fluxos de trabalho complexos que envolvem muitas partes e ações concorrentes.

6.19 Motivação: outros benefícios de casos de uso? Requisitos em contexto

Motivação
(pág. 90)

Uma motivação para casos de uso é o enfoque em quem são os atores-chave, seus objetivos e tarefas comuns. Além disso, em essência, casos de uso constituem uma forma simples, amplamente entendida (uma forma de estória ou cenário).

Uma outra motivação é substituir listas de funções detalhadas, de baixo nível (que eram comuns nos métodos tradicionais de requisitos de década de 1970), por casos de uso. Essas listas tendiam a aparecer como segue:

ID	Características
CAR1.9	O sistema deve aceitar a entrada dos identificadores de item.
...	...
CAR2.4	O sistema deve registrar pagamentos a crédito no sistema de contas a receber.

Como sugere o título do livro *Casos de Uso: Requisitos em Contexto* [GK00], casos de uso organizam um conjunto de requisitos no *contexto* de cenários típicos de uso de um sistema. Isso é bom – considerar e agrupar os requisitos com base na linha comum de cenários orientados a usuários (isto é, casos de uso) melhora a coesão e a compreensão. Em um projeto recente de sistema de controle de tráfego aéreo os requisitos foram originalmente escritos no antigo formato de lista de funções, enchendo volumes de especificações incompreensíveis e não relacionadas. Uma nova equipe de liderança analisou e re-organizou os requisitos massivos basicamente em casos de uso. Isso forneceu um modo unificador e fácil de entender de colocar juntos os requisitos – em estórias de requisitos no contexto do uso.

Especificação suplementar
(pág. 130)

Reiterando, os casos de uso não são os únicos artefatos necessários para definir requisitos. Requisitos não funcionais, leiautes de relatórios, regras de domínio e outros elementos difíceis de colocar são melhor captados nas Especificações Suplementares do PU.

Listas de características do sistema em alto nível são aceitáveis

Visão
(pág. 135)

Apesar das listas detalhadas de funções serem indesejáveis, uma lista sucinta de características de alto nível chamada *características do sistema*, adicionada a um documento de Visão pode resumir de maneira útil a funcionalidade do sistema. Em vez de 50 páginas de características de baixo nível, uma lista de características do sistema inclui apenas algumas dúzias de itens. Fornece um resumo sucinto da funcionalidade, independente da visão dos casos de uso. Por exemplo:

Resumo das características do sistema

- captação de vendas;
- autorização de pagamento (credito, débito, cheque);
- administração do sistema para usuários, segurança, tabelas constantes e de códigos, etc.;
- ...

Quando listas detalhadas de características são mais apropriadas do que casos de uso?

Algumas vezes, os casos de uso realmente não servem; algumas aplicações exigem que seja adotado um ponto de vista orientado às características. Por exemplo, servidores de aplicações, produtos de bancos de dados e outros sistemas de middleware, ou retaguarda, precisam ser considerados e evoluídos em termos de *características* ("Necessitamos suporte para Web Services na próxima versão"). Os casos de uso não são um recurso natural para definir essas aplicações ou a forma como devem evoluir em termos das forças do mercado.

6.20 Exemplo: Banco Imobiliário

O único caso de uso significativo no sistema de software Banco Imobiliário é *Jogar Banco Imobiliário* – mesmo se ele não passar no Teste do Chefe! Como o jogo é processado como uma simulação por computador, observada simplesmente por uma pessoa, podemos dizer que essa pessoa é um observador, não um jogador.

Esse estudo de caso vai mostrar que casos de uso não são sempre a melhor alternativa para requisitos de comportamento. Tentar captar todas as regras do jogo no formato de caso de uso é esquisito e não natural. Onde ficam as regras do jogo? Primeiro, geralmente, elas são **regras do domínio** (algumas vezes chamadas de regras de negócio). No PU, regras de domínio podem ser parte das Especificações Suplementares (ES). Na seção "regras do domínio" das ES haveria provavelmente uma referência, seja para o livrero oficial em papel das regras, ou para o site da Web que as descreve. Além disso, pode haver um indicador para essas regras a partir do texto do caso de uso, como mostrado a seguir.

Especificação Suplementar
(pág. 130)

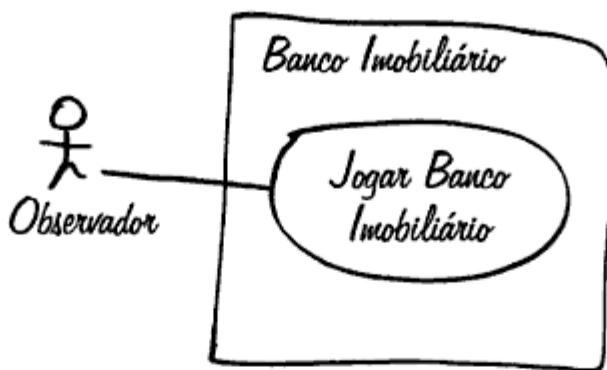


Figura 6.6 Diagrama de caso de uso (Diagrama de contexto) para o sistema Banco Imobiliário.

O texto para esse caso de uso é muito diferente daquele para o problema PDV Prox-Ger, pois é uma simples simulação e as várias possíveis ações de jogadores (simuladas) são captadas nas regras do domínio, em vez de na seção Extensões.

Caso de uso CDU1: Jogar Banco Imobiliário

Escopo: aplicação Banco Imobiliário

Nível: objetivo do usuário

Autor Principal: Observador

Interessados e Interesses:

- Observador: Deseja facilmente observar as saídas da simulação do jogo.

Principal Cenário de Sucesso:

1. Observador solicita início de novo jogo e insere número de jogadores.
2. Observador começa a jogar.
3. Sistema mostra rastreamento do jogo para o movimento do jogador seguinte (ver regras de domínio e “rastreamento do jogo” no glossário para obter detalhes do rastreamento).

Repetir passo 3 até que apareça o vencedor ou que o Observador cancele.

Extensões:

- *a. Em qualquer momento o Sistema falha:

(para apoiar a restauração, Sistema registra depois de cada movimento completado)

1. Observador reinicia Sistema.

2. Sistema detecta falha anterior, reconstrói estado e fica pronto para continuar.

3. Observador escolhe continuar (a partir da última vez completada pelo jogador).

Requisitos Especiais:

- Disponibilizar modos de rastreamento gráfico e textual.

6.21 Processo: como trabalhar com casos de uso em métodos iterativos?

Os casos de uso são centrais para o PU e muitos outros métodos iterativos. O PU encoraja o **desenvolvimento guiado por casos de uso**. Isso significa que:

- Requisitos funcionais são principalmente registrados em casos de uso (o Modelo de Casos de Uso); outras técnicas de requisitos (tais como listas de funções) são secundárias, se é que chegam a ser usadas.
- Casos de uso são uma parte importante do planejamento iterativo. O trabalho de uma iteração é – em parte – definido pela escolha de alguns cenários de casos de uso ou por casos de uso inteiros. Além disso, os casos de uso são uma entrada-chave para estimativas.
- **Realizações de caso de uso** orientam o projeto. Isto é, a equipe projeta objetos e subsistemas que colaboram, de maneira a executar ou realizar os casos de uso.
- Casos de uso freqüentemente influenciam a organização dos manuais de usuário.
- Teste funcional ou de sistema corresponde aos cenários dos casos de uso.
- *Wizards* de IU ou atalhos podem ser criados para os cenários mais comuns de casos de uso importantes para facilitar as tarefas comuns.

Como evoluir casos de uso e outras especificações ao longo das iterações?

Esta seção reitera uma idéia chave no desenvolvimento evolutivo e iterativo: a duração e o nível de esforço das especificações ao longo das iterações. A Tabela 6.1 apresenta uma amostra (não uma receita) que informa a estratégia de como os requisitos são desenvolvidos no PU.

Note que uma equipe técnica começa construindo o núcleo de produção do sistema, quando talvez 10% dos requisitos estão detalhados e, de fato, a equipe deliberadamente atrasa a continuação em profundidade do trabalho de requisitos até perto do fim da primeira iteração de elaboração.

Esta é a diferença-chave entre o desenvolvimento iterativo e o processo em cascata: o desenvolvimento do núcleo do sistema com qualidade de produção começa logo, muito antes que todos os requisitos sejam conhecidos.

Observe que, perto do fim da primeira iteração de elaboração, existe um segundo seminário de requisitos, durante o qual talvez 30% dos casos de uso sejam redigidos em detalhe. Essa análise de requisitos escalonada se beneficia da realimentação obtida por se ter construído um pouco do núcleo do software. A realimentação inclui a avaliação do usuário, os testes e uma melhoria no “conhecimento daquilo que não sabíamos”. Isto é, o ato de construir o software rapidamente traz à tona hipóteses e questões que necessitam de esclarecimento.

No PU, a escrita de casos de uso é incentivada em um workshop de requisitos. A Figura 6.7 oferece sugestões sobre a ocasião e o lugar para fazer esse trabalho.

Tabela 6.1 Amostra do esforço em requisitos entre as iterações iniciais; isto não é uma receita

Disciplina	Artefato	Concep. 1 semana	Comentários e Nível de Esforço em Requisitos			
			Elab. 1 4 semanas	Elab. 2 4 semanas	Elab. 3 3 semanas	Elab. 4 3 semanas
Requisitos	Modelo de Casos de Uso	<p>Seminário de requisitos de dois dias. A maioria dos casos de uso é identificada por nome, e estes são resumidos em um parágrafo curto.</p> <p>Pegue 10% dos casos da lista dos mais relevantes para analisar e redigir em detalhe. Esses 10% serão os mais arquiteturalmente importantes, arriscados e de maior valor de negócio.</p>	<p>Perto do fim desta iteração faça um seminário de requisitos de dois dias. Obtenha percepções e realimentação a partir do trabalho de implementação, e, então complete 30% dos casos de uso em detalhe.</p>	<p>Perto do fim desta iteração faça um seminário de requisitos de dois dias. Obtenha percepções e realimentação a partir do trabalho de implementação, e, então complete 50% dos casos de uso em detalhe.</p>	<p>Repita e complete 70% de todos os casos de uso em detalhe.</p>	<p>Repita com a metade de ter 80-90% dos casos de uso esclarecidos e redigidos em detalhes.</p> <p>Somente uma pequena parte destes foi construída na elaboração; o restante é feito na construção.</p>
Projeto	Modelo de Projeto	nenhum	Projete um pequeno conjunto de requisitos de alto risco significativos para a arquitetura.	Repita	Repita	Repita. Os aspectos significativos de alto risco e do ponto de vista arquitetural deveriam agora estar estabilizados.
Implementação	Modelo de Implementação (código, etc.)	Nenhum	Implemente estes.	Repita. Nesta etapa, 5% do sistema final está construído.	Repita. Nesta etapa, 10% do sistema final está construído.	Repita. Nesta etapa, 15% do sistema final está construído.
Gestão de Projeto	Plano de Desenvolvimento de Software	Estimativa muito vaga do esforço total.	As estimativas começam a tomar forma.	Um pouco melhor...	Um pouco melhor...	As estimativas de duração geral do projeto, os principais marcos de referência, as estimativas de esforço e de custo podem agora ser racionalmente determinadas.

Quando os vários artefatos do PU (inclusive casos de uso) devem ser criados?

A Tabela 6.2 ilustra alguns artefatos do PU e um exemplo do seu cronograma de início e refinamento. O Modelo de Casos de Uso é iniciado na concepção, com talvez apenas 10% dos casos de uso significativos arquiteturalmente escritos com algum detalhe. A maioria é redigida incrementalmente ao longo das iterações da fase de elaboração, de modo que ao fim da elaboração, um grande grupo de casos de uso detalhados e outros requisitos (nas Especificações Suplementares) estejam redigidos, fornecendo, assim, uma base realista para elaborar estimativas até o fim do projeto.

Tabela 6.2 Amostra dos Artefatos do PU e seqüência temporal (i = início, r = refino)

Disciplina	Artefato Iteração →	Concep. II	Elab. El.En	Const. C1..Cn	Trans. T1..T2
Modelagem de Negócio	Modelo de Domínio		i		
Requisitos	<i>Modelo de Caso de Uso</i>	i	r		
	Visão	i	r		
	Especializações Suplementares	i	r		
	Glossário	i	r		
Projeto	Modelo de Projeto		i	r	
	Documento de Arquitetura de Software		i		

Como redigir casos de uso na concepção

A discussão a seguir amplia as informações da Tabela 6.1.

Nem todos os casos de uso são redigidos no seu formato completo durante a fase de concepção. Ao contrário, suponha que ocorra um seminário de requisitos de dois dias durante a investigação inicial do ProxGer. A parte inicial do dia é usada para identificar objetivos e interessados no projeto e para especular o que está dentro e o que está fora do escopo do projeto. Escreve-se uma tabela ator-objetivo-caso de uso, que é exibida por um projetor ligado ao computador. Inicia-se um diagrama de caso de uso de contexto. Após algumas horas, talvez 20 objetivos de usuário estejam identificados por nome, inclusive *Processar Venda*, *Tratar Devoluções*, etc. A maioria dos casos de uso interessantes, complexos ou de alto risco são escritos no formato resumido, cada um levando, em média, dois minutos para ser redigido. A equipe começa a formar uma imagem de alto nível da funcionalidade do sistema.

Depois, 10% a 20% dos casos de uso que representam as funções complexas do núcleo do sistema, que exigem a construção da arquitetura do núcleo, ou que são particularmente de risco em alguma dimensão, são reescritos em formato completo; a equipe faz uma investigação um pouco mais profunda, para melhor compreender a

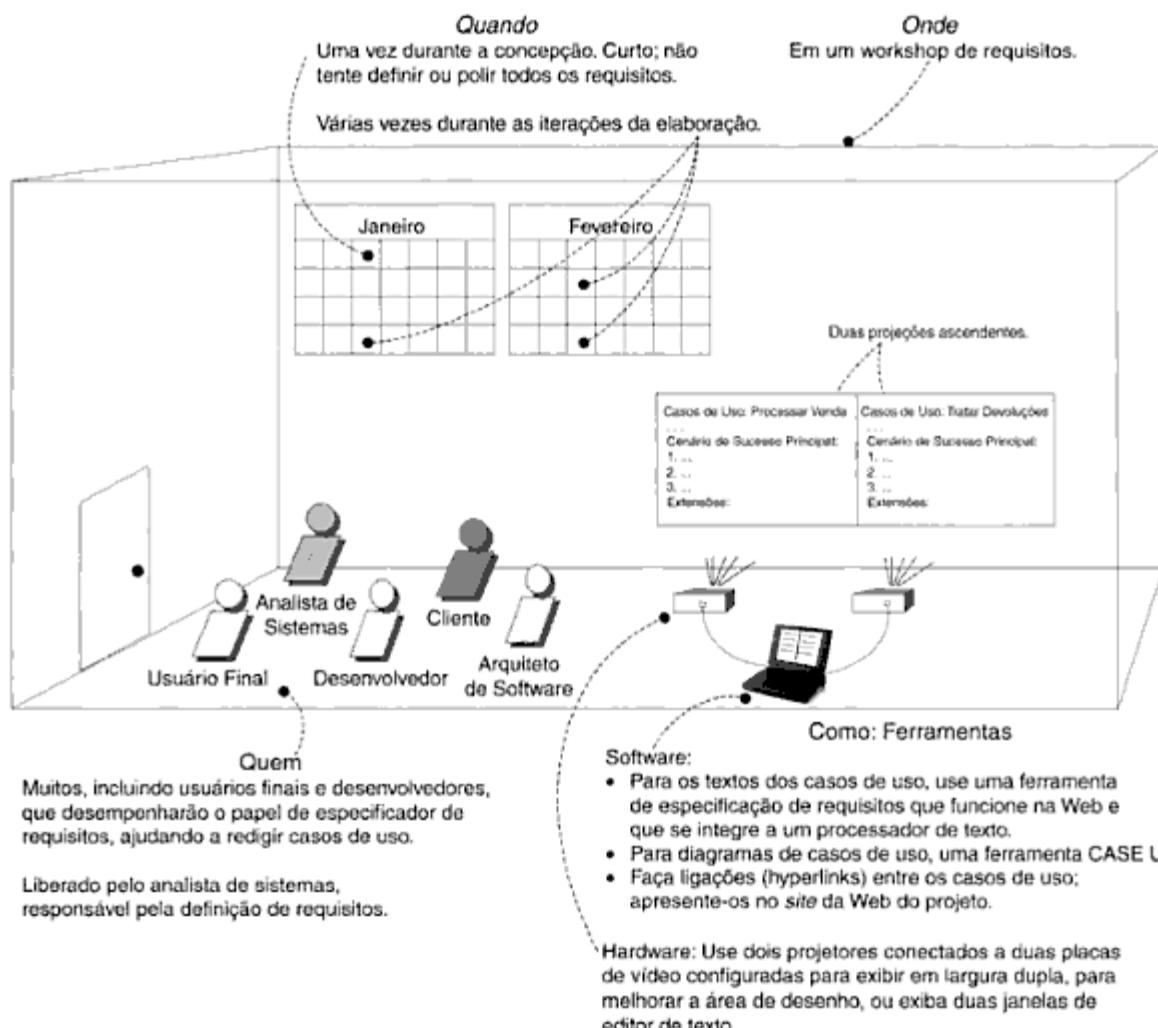


Figura 6.7 Processo e estabelecimento do contexto para escrita de casos de uso.

magnitude, as complexidades e os perigos ocultos do projeto, por meio da investigação profunda de uma pequena amostra de casos de uso interessantes. Talvez isso signifique dois casos de uso: *Processar Venda* e *Tratar Devoluções*.

Como redigir casos de uso na elaboração?

A discussão a seguir amplia as informações da Tabela 6.1.

Esta é uma fase de múltiplas iterações de tempo fixo (por exemplo, quatro iterações) na qual partes do sistema que ofereçam risco, tenham grande valor, ou sejam arquiteturalmente significativas, são construídas incrementalmente e a “maioria” dos requisitos são identificados e esclarecidos. A realimentação a partir dos passos concretos de programação influencia e informa a equipe na sua compreensão dos

requisitos, os quais são refinados de forma iterativa e adaptativa. Talvez ocorra um seminário de requisitos de dois dias em cada iteração – quatro seminários no total. Contudo, nem todos os casos de uso são investigados em cada seminário. Eles são priorizados; os seminários iniciais focalizam um subconjunto dos casos de uso mais importantes.

Cada seminário subsequente é um momento para adaptar e refinar a visão dos requisitos centrais, os quais serão instáveis nas iterações iniciais, estabilizando-se nas posteriores. Assim, existe uma interação iterativa entre a descoberta de requisitos e a construção de partes do software.

Durante cada seminário de requisitos, os objetivos dos usuários e a lista de casos de uso são refinados. Mais casos de uso são redigidos e reescritos no seu formato completo. Ao fim da elaboração, “80-90%” dos casos de uso estão escritos em detalhe. Em um sistema PDV com 20 casos de uso no nível de objetivos de usuários, 15 ou mais dos casos de uso mais complexos e de maiores riscos devem ser investigados, redigidos e reescritos em um formato completo.

Observe que a elaboração envolve a programação de partes do sistema. Ao fim desse passo, a equipe do ProxGer deveria não apenas ter uma melhor definição dos casos de uso, como também algum software executável.

Como redigir casos de uso na construção?

A fase de construção é composta de iterações de tempo fixo (por exemplo, 20 iterações, cada uma com duas semanas) dedicadas a completar o sistema, uma vez que os problemas instáveis e de risco já foram resolvidos na elaboração. Pode ainda ser necessário escrever alguns casos de uso de menor importância e, talvez, realizar seminários de requisitos, mas em número muito menor do que durante a elaboração.

Estudo de caso: casos de uso na fase de concepção do ProxGer

Conforme descrito na seção anterior, nem todos os casos de uso são redigidos no seu formato completo durante a concepção. O Modelo de Casos de Uso nessa fase do estudo de caso poderia ser detalhado conforme mostrado a seguir:

Completo	Informal	Resumido
Processar Venda Tratar Devoluções	Processar Aluguel Analizar Atividade de Vendas Gerenciar Segurança ...	Abrir Fechar Gerenciar Usuários Iniciar o Sistema Encerrar o Sistema Gerenciar as Tabelas do Sistema ...

6.22 Histórico

A idéia de casos de uso para descrever requisitos funcionais foi introduzida, em 1986, por Ivar Jacobson [Jacobson92], um dos principais contribuintes da UML e do PU. A idéia dos casos de uso de Jacobson foi pioneira e amplamente aceita. Apesar de muitos terem feito contribuições no assunto, discutivelmente o passo seguinte mais influente e coerente na definição do que são casos de uso e sobre como redigi-los veio de Alistair Cockburn (que foi treinado por Jacobson), baseado no seu trabalho anterior e nos escritos de 1992 em diante [por exemplo, Cockburn01].

6.23 Leituras recomendadas

O guia mais popular para elaboração de casos de uso, traduzido para vários idiomas, é *Writing Effective Use Cases* [Cockburn01]⁷. Esse guia tornou-se, por boas razões, o livro mais lido e seguido a respeito de casos de uso e, portanto, é recomendado como a referência básica sobre o assunto. Este capítulo introdutório é consequentemente baseado e consistente com o seu conteúdo.

Patterns for Effective Use Cases, escrito por Adolph e Bramble, continua de alguma forma a partir de onde *Writing* parou, abordando muitas sugestões úteis – em formato de padrão – relacionadas ao processo de criar excelentes casos de uso (organização da equipe, metodologia, edição) e como estruturá-los e redigi-los melhor (padrões para julgar e melhorar seu conteúdo e organização).

Geralmente, casos de uso são melhor redigidos em conjunto com um parceiro durante o seminário de requisitos. Um excelente guia na arte de conduzir um seminário é *Requirements by Collaborations: Workshops for Defining Needs*, escrito por Ellen Gottesdiener.

Use Case Modeling, escrito por Bittner e Spence, é outro recurso de qualidade, vindo de dois modeladores experientes que também entendem de desenvolvimento iterativo e evolutivo e RUP, e apresentam a análise de casos de uso naquele contexto.

Structuring Use Cases with Goals [Cockburn97] é o artigo mais freqüentemente citado sobre casos de uso, disponível online em alistair.cockburn.us.

Use Cases: Requirements in Context, escrito por Kulak e Guiney, também vale a pena. Enfatiza o importante ponto de vista – como diz o título – de que casos de uso não são somente outro artefato para requisitos, mas o veículo central que guia o trabalho de requisitos.

⁷ Note que Cockburn rima com “slow burn” (fogo lento, em português).

OUTROS REQUISITOS

Rápido, barato, bom: escolha qualquer par.
— Anônimo

Objetivos

- Mostrar a Especificação Suplementar, o Glossário, a Visão e as Regras de Negócio.
- Comparar e contrastar características do sistema com casos de uso.
- Definir atributos de qualidade.

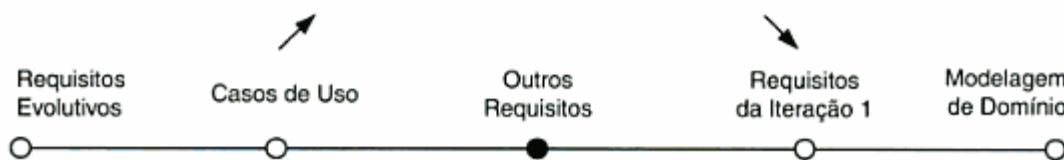
Introdução

Existem alguns outros artefatos de requisitos importantes do PU além de casos de uso, que são introduzidos neste capítulo. Se você deseja pular este capítulo – que trata de tópicos secundários de requisitos em vez de A/POO – não há problema. Passe para o assunto clássico de AOO sobre modelagem de domínio na pág. 157, mas antes leia o resumo dos requisitos da iteração 1 que começa na pág. 150.

Então por que incluir este capítulo, se ele não é essencial para aprender A/POO? Porque ele dá coesão aos estudos de caso e oferece um exemplo mais completo de requisitos.

O que vem a seguir?

Observados os casos de uso, este capítulo explora outros artefatos de requisitos no PU. O capítulo seguinte define o escopo do trabalho para a iteração 1.



Outros artefatos de requisitos

Casos de uso não são toda a estória.

A **Especificação Suplementar** capta e identifica outros tipos de requisitos, tais como: relatórios, documentação, empacotamento, capacidade de suporte, licenciamento, etc.

O **Glossário** contém termos e definições; ele também pode desempenhar o papel de um dicionário de dados.

A **Visão** resume a “visão” do projeto – um resumo executivo. Ela serve para comunicar, em termos sucintos, as grandes idéias.

As **Regras do Negócio** (ou Regras do Domínio) captam regras ou políticas de longo prazo e abrangentes, tais como leis fiscais que transcendem uma aplicação particular.

7.1 Quão completos são os exemplos?

O objetivo principal do livro é a A/POO básica, não os detalhes secundários dos requisitos do ProxGer discutidos neste capítulo. Assim, em vez de mostrar exemplos exaustivos de requisitos¹, o capítulo apresenta exemplos parciais.

Algumas seções são brevemente discutidas, para que se possa fazer a conexão entre o trabalho anterior e o trabalho futuro, destacar problemas dignos de nota, dar uma idéia do conteúdo e seguir em frente rapidamente.

7.2 Diretriz: deve-se fazer a análise em profundidade durante a concepção?

Não. O PU é um método iterativo e evolutivo, o que significa que a programação e o teste com qualidade de produção devem acontecer muito cedo, bem antes dos requisitos terem sido totalmente analisados ou registrados. A *realimentação* proveniente da programação e testes precoces serve para evoluir os requisitos.

No entanto, a pesquisa mostra que é útil ter, logo no início, uma lista de alto nível dos “dez mais” requisitos em uma granularidade maior. É também útil despender certo tempo no início entendendo os requisitos não funcionais (tais como, desempenho ou confiabilidade), pois eles têm um impacto significativo nas escolhas arquiteturais.

¹ A ênfase do escopo não é apenas um problema de requisitos, mas de escrever sobre requisitos!

Especificações confiáveis: uma contradição?

Os exemplos seguintes de requisitos escritos podem dar a ilusão de que os requisitos reais estão entendidos e bem definidos e podem (desde cedo) ser usados para estimar e planejar confiavelmente o projeto. Essa ilusão é mais forte para desenvolvedores, mas não de software; programadores sabem por penosa experiência como isso não é confiável. Como mencionado, estudos de caso (por exemplo, [Thomas01] e [Larman03]) mostram agora que é um mal entendido acreditar que requisitos detalhados iniciais são úteis ou confiáveis em projetos de software. De fato, pelo contrário, pois quase 50% das características iniciais especificadas pelo modelo em cascata nunca são usadas em um sistema.

O que realmente importa é rapidamente construir um software que passa pelos testes de aceitação definidos pelos usuários e que satisfaz os seus verdadeiros objetivos – que são freqüentemente não descobertos até que os usuários estejam avaliando ou trabalhando com o software.

Redigir uma Visão e uma Especificação Suplementar vale a pena como exercício no esclarecimento de uma *primeira aproximação* do que é desejado, da motivação do produto e como um repositório das grandes idéias. Mas elas não são – nem qualquer artefato de requisitos é – uma especificação confiável. Apenas a escrita de código, seu teste, obter realimentação com ele, a contínua e estreita colaboração com usuários e clientes e a adaptação, realmente atingem a meta.

Isto não é um apelo para abandonar a análise e o raciocínio e correr para o código, mas uma sugestão de tratar por alto os requisitos escritos, começando logo a programação e continuamente – idealmente, diariamente – engajar os usuários e testes para obter realimentação.

7.3 Diretriz: esses artefatos devem ficar no site da Web do projeto?

Definitivamente sim. Como este é um livro, esses exemplos e os casos de uso parecem ser estáticos e talvez orientados a papel. No entanto, esses artefatos devem ser usualmente digitais, registrados online no site da web do projeto. E, em vez de ser simples documentos estáticos, eles podem ser ligados por meio de links, ou registrados em ferramentas além de processadores de texto ou planilhas. Por exemplo, muitos deles podem ser armazenados em uma Wiki Web².

² Para uma introdução sobre Wikis, ver <http://en.wikipedia.org/wiki/WikiWiki>.

7.4 Exemplo ProxGer: especificação suplementar (parcial)

Especificação Suplementar

Histórico das Revisões

Versão	Data	Descrição	Autor
Minuta da concepção	10 de janeiro de 2031	Primeira minuta. A ser refinada, principalmente durante a elaboração.	Craig Larman

Introdução

Este documento é o repositório de todos os requisitos do PDV ProxGer não documentados nos casos de uso.

Funcionalidade

(*Funcionalidade comum a muitos casos de uso*)

Registro e tratamento de erros

Registrar todos os erros em dispositivo de armazenamento persistente.

Regras de negócio “plugáveis”

Em vários pontos de cenários de diversos casos de uso (a serem definidos), apoiar a habilidade de personalizar a funcionalidade do sistema com um conjunto de regras quaisquer que são executadas naquele ponto ou quando ocorre um evento pré-determinado.

Segurança

Toda utilização requer autenticação do usuário.

Usabilidade

Fatores humanos

O cliente será capaz de ver o que está escrito em um grande monitor de vídeo do PDV. Portanto:

- O texto deve ser facilmente visível a uma distância de um metro.
- Cores associadas com formas comuns de daltonismo devem ser evitadas.

Velocidade, facilidade e processamento isento de erros são de importância primordial no processamento de vendas, pois o comprador deseja ir embora rapidamente, senão ele perceberá, de forma menos positiva, a experiência da compra (bem como quem está vendendo).

O caixa freqüentemente está olhando para o cliente ou para os itens que este está comprando, e não para o monitor de vídeo do computador. Portanto, sinais e alertas devem ser informados por sons, em vez de somente por meios visuais.

Confiabilidade

Facilidade de recuperação

Se ocorrer uma falha no uso de um serviço externo (autorização de pagamentos, sistema de contabilidade....) tente resolver o problema com uma solução local (p. ex., armazenar e seguir em frente), de modo a poder completar uma venda. Aqui, é necessário muito mais análise...

Desempenho

Conforme mencionado em fatores humanos, os compradores querem completar o processamento de uma venda *muito* rapidamente. Um gargalo em potencial é a autorização externa de pagamento. Nosso objetivo é obter uma autorização em menos de um minuto, em 90% das vezes.

Manutenibilidade

Adaptabilidade

Diferentes clientes do PDV ProxGer têm regras de negócio e necessidades de processamento únicos quando processam uma venda. Portanto, em vários pontos definidos no cenário (por exemplo, quando é iniciada uma nova venda, quando é adicionado um novo item de compra), regras de negócio "plugáveis" serão habilitadas.

Facilidade de configuração

Diferentes clientes desejam diferentes configurações de rede para seus sistemas PDV, tais como clientes finos (thin clients) versus clientes espessos (thick clients), duas camadas versus N camadas físicas e assim por diante. Além disso, eles desejam a possibilidade de modificar essas configurações, de modo que elas possam refletir as mudanças nos seus negócios e correspondentes necessidades de desempenho. Portanto, o sistema deverá ser bastante configurável para refletir essas necessidades. É necessário muito esforço e trabalho de análise neste campo para definir as áreas e o grau de flexibilidade desejável, bem como o esforço para alcançá-lo.

Restrições de Implementação

A liderança do projeto ProxGer insiste em uma solução com tecnologias Java, prevendo que isto irá melhorar a portabilidade e a manutenibilidade a longo prazo, além da facilidade de desenvolvimento.

Componentes comprados

- Software para cálculo de impostos. A maioria fornece o suporte de componentes "plugáveis" para diferentes países.

Componentes livres com código-fonte aberto

Em geral, recomendamos maximizar o uso de tecnologia Java livre com componentes de código-fonte aberto neste projeto.

Embora seja prematuro projetar e escolher componentes definitivamente, sugerimos os seguintes como candidatos prováveis:

- Jlog framework para registro de entrada de usuários
- ...

Interfaces

Hardware e interfaces de interesse

- Monitor com tela sensível a toques (este é visto pelos sistemas operacionais como um monitor comum e os toques como eventos de mouse)
- Leitor scanner a laser, de código de barras (estes normalmente se conectam a um teclado especial, e a entrada lida pelo scanner é percebida pelo software como acionamento de teclas)
- Impressora de recibos
- Leitor de cartão de crédito/débito
- Leitor de assinaturas (mas não na versão 1)

Interfaces de software

Para a maioria dos sistemas externos que colaboraram (sistema de cálculo de impostos, contabilidade, estoques,...), precisamos ser capazes de "plugar" vários sistemas e, assim, interfaces diversas.

Regras de Domínio (de negócio) Específicas da Aplicação

(Para regras gerais, veja o documento "Regras de Negócios" em separado)

ID	Regra	Mutabilidade	Fonte
REGRA 1	Regras de desconto para compradores. Exemplos: Empregado – 20%. Cliente Preferencial – 10%. Idoso – 15%.	Alta. Cada varejista usa regras diferentes.	Política de varejo.
REGRA 2	Regras de desconto de venda (em nível de transação). Aplica-se ao total antes do imposto. Exemplos: 10% se o total for maior do que R\$100,00. 5% toda segunda-feira. 10% para todas as vendas de hoje das 10:00hs às 15:00 hs. Tofu: 50% hoje das 9:00 às 10:00hs.	Alta. Cada varejista usa regras diferentes e elas podem mudar diariamente ou de hora em hora.	Política de varejo.
REGRA 3	Regras de desconto por produto (nível de linha de item). Exemplos: 10% de desconto para tratores nesta semana. Compre 2 hambúrgueres vegetarianos e leve um de graça.	Alta. Cada varejista usa regras diferentes e elas podem mudar diariamente ou de hora em hora.	Política de varejo

Aspectos legais

Recomendamos o uso de alguns componentes com código-fonte livre, se suas restrições de licenciamento puderem ser resolvidas de maneira a permitir a revenda de produtos que incluem software com código-fonte livre.

Todas as regras relativas a tributos devem, de acordo com a legislação, ser aplicadas durante as vendas. Note que essas regras podem mudar freqüentemente.

Informações em domínios de interesse

Preços

Além das regras de preços descritas na seção regras do domínio, observe que produtos têm um *preço original* e, opcionalmente, um *preço permanentemente remarcado*. O preço de um produto (antes dos descontos adicionais) é o preço permanentemente remarcado, caso esse exista. As empresas mantêm o preço original mesmo se um preço permanentemente remarcado estiver sendo aplicado por razões contábeis e tributárias.

Tratamento de pagamentos a crédito e a débito

Quando um pagamento eletrônico a crédito ou a débito é aprovado por um serviço de autorização de pagamentos, esse é responsável pelo pagamento ao vendedor, não ao comprador. Consequentemente, para cada pagamento, o vendedor necessita registrar valores pecuniários a seu crédito em suas contas a receber, que deverão ser provenientes do serviço de autorização de créditos. Geralmente isso será feito toda noite: o serviço de autorização de crédito executará uma transferência eletrônica de fundos para a conta do vendedor para cobrir o total diário devido naquele dia, deduzida uma pequena taxa de serviço por transação cobrada pelo serviço.

Impostos sobre as vendas

Cálculos de valores de impostos sobre vendas podem ser muito complexos e mudam com regularidade, em consequência de mudanças de legislação tributária em todos os níveis de governo. Portanto, é aconselhável delegar o cálculo de impostos para um software de terceiros que faça esse cálculo (existem vários disponíveis). Os impostos podem ser municipais, regionais, estaduais ou federais. Alguns itens podem ser isentos de impostos incondicionalmente, ou podem ser isentos dependendo do comprador ou receptor final da mercadoria (por exemplo, um agricultor ou uma criança).

Identificadores de itens: CUPs, EANs, SKUs, códigos de barras e leitores de códigos de barras

O PDV ProxGer precisa dar suporte a vários esquemas de identificadores de itens. Os CUPs (Códigos Universais de Produto, do inglês Universal Product Code – UPC), EANs (European Article Numbering) e SKUs (Stock Keeping Units) são três sistemas comuns para identificadores de produtos vendidos. Os JANs (Japanese Article Numbers) são uma versão do EAN.

Os SKUs são identificadores totalmente arbitrários definidos pelo varejista. Entretanto, os CUPs e os EANs possuem um componente regulador e padrões. Veja www.adams1.com/pub/russadam/upccode.html para uma visão geral. Veja também www.uc.council.org e www.ean-int.org.

7.5 Comentário: especificação suplementar

A **Especificação Suplementar** documenta outros requisitos, informações e restrições que não são facilmente documentados nos casos de uso ou no Glossário, incluindo atributos ou requisitos de qualidade que se aplicam a todo o sistema – os atributos “URPS+” (usability (usabilidade), reliability (confiabilidade), performance (desempenho), supportability (facilidade de suporte) e outros).

Note que os requisitos não funcionais específicos a um caso de uso podem (e provavelmente devem) ser primeiros escritos resumidamente dentro do caso de uso, na seção *Requisitos Especiais*, enquanto você está pensando no caso de uso. Mas depois desse passo informal eles devem ser movidos para a Especificação Suplementar, a fim de manter todos os requisitos não funcionais em um único lugar e sem duplicação.

Os elementos da Especificação Suplementar incluem:

- requisitos FURPS+ – funcionalidade, usabilidade, confiabilidade, desempenho e manutenibilidade
- relatórios
- restrições de hardware e software (sistemas operacionais e de rede, etc.)
- restrições de desenvolvimento (por exemplo, processo ou ferramentas de desenvolvimento)
- outras restrições de projeto e implementação
- preocupações com internacionalização (unidades, línguas,...)
- documentação (do usuário, instalação, administração) e ajuda
- licenciamento e outras preocupações legais
- empacotamento
- padrões (técnico, segurança, qualidade)
- preocupações com o ambiente físico (por exemplo, calor ou vibração)
- preocupações operacionais (por exemplo, como os erros são tratados ou como e com que freqüência devem ser feitas cópias de segurança)
- regras do domínio específicas da aplicação
- informações sobre domínios de interesse (por exemplo, qual é o ciclo completo de tratamento de pagamentos a crédito?)

Atributos de qualidade

Alguns requisitos são chamados de **atributos de qualidade** [BCK98] (ou “-idades”) de um sistema. Esses incluem, dentre outros, a usabilidade e a confiabilidade. Note que essas são qualidades do sistema, não dos atributos em si, que não são necessariamente de alta qualidade. Por exemplo, a qualidade “facilidade de suporte” pode deliberadamente ser escolhida como sendo baixa se o produto não se destina a servir a um propósito de longo prazo.

Quando colocamos nosso “chapéu de arquiteto”[†], os atributos de qualidade aplicáveis a todo o sistema (e, consequentemente, também à Especificação Suplementar onde são registrados) são especialmente interessantes porque – como será introduzido no Capítulo 33 – *a análise e o projeto da arquitetura são bastante voltados à identificação e resolução dos atributos de qualidade no contexto dos requisitos funcionais*.

Por exemplo, suponha que um dos atributos de qualidade seja que o sistema Prox-Ger deve ser bastante tolerante a falhas quando os serviços remotos falham. Do ponto de vista arquitetural, isso terá uma grande influência sobre as decisões de projeto de grande escala.

Funcionalidade nas especificações suplementares? Isso não deveria ficar nos casos de uso?

Algumas funções ou características não se encaixam no formato de caso de uso. Na década de 1990 trabalhei em uma empresa que construiu um middleware em Java e uma plataforma baseada em agentes. Para a versão seguinte (como a maioria dos produtos de middleware ou servidor) não pensamos na sua funcionalidade em termos de casos de uso – isso não fazia sentido. Mas pensamos na funcionalidade em termos de **características**, tais como “adicionar apoio para EJB Entity Bean 1.0”.

O PU certamente permite essa abordagem de requisitos orientada a características, na qual a lista de requisitos está na Especificação Suplementar.

O PU encoraja, mas não requer casos de uso para descrever a funcionalidade; casos de uso são uma boa maneira de pensar sobre e colocar juntos um conjunto relacionado de características em termos de cenários típicos de uso de um produto. Eles nem sempre se encaixam.

Regras (de negócio) do domínio específicas da aplicação

Regras do domínio amplas e gerais, tais como a legislação de impostos, são próprias do artefato do PU “Regras de Negócio”, como um repositório central compartilhado. No entanto, regras mais estritas específicas da aplicação, tais como “como calcular um desconto de linha de item”, podem ser registrados na especificação suplementar.

Informações sobre domínios de interesse

É interessante que um especialista escreva (ou forneça URLs para) algumas explicações sobre os domínios relacionados com o novo sistema de software (vendas e contabilidade, a geofísica dos fluxos subterrâneos de petróleo/água/gás, etc.), oferecendo, assim, o contexto e uma percepção mais profunda para a equipe de desenvolvi-

[†] N. de R.T.: Isto é, nos comportamos segundo o papel de arquiteto de sistema.

mento. Tal documento pode conter indicações para alguma literatura importante, especialistas no assunto, fórmulas, leis ou outras referências. Por exemplo, os "mistérios" dos esquemas de codificação CUP e EAN, e a simbologia de códigos de barras, devem, até certo ponto, ser compreendidos pela equipe do ProxGer.

7.6 Exemplo ProxGer: visão (parcial)

Visão

Histórico das Revisões

Visão	Data	Descrição	Autor
Minuta da concepção	10 de janeiro de 2031	Primeira minuta. A ser refinada, principalmente, durante a elaboração.	Craig Larman

Introdução

A análise neste exemplo é ilustrativa, mas fictícia

Imaginamos a aplicação para ponto-de-vendas de nova geração (PDV ProxGer), tolerante a falhas e com flexibilidade para atender a diversas regras de negócios variáveis de acordo com o usuário, múltiplos mecanismos de interface com o usuário, múltiplos terminais e possibilidade de integração com múltiplos sistemas de suporte fornecidos por terceiros.

Posicionamento

Oportunidade de negócio

Os produtos PDV já existentes não são adaptáveis aos negócios dos clientes em termos de possibilitar regras de negócio e projetos de redes variáveis (por exemplo, cliente fino ou não; arquiteturas de 2, 3 ou 4 camadas). Além disso, eles não apresentam uma escabilidade satisfatória, à medida que o número de terminais aumenta e o negócio cresce. E nenhum deles pode trabalhar ora em modo online ora em modo offline, adaptando-se dinamicamente dependendo das falhas. Da mesma forma, nenhum deles se integra facilmente com muitos sistemas fornecidos por terceiros. Nenhum deles tem provisão para a incorporação de novas tecnologias em matéria de terminais, tais como os PDAs¹ móveis. Há uma insatisfação no mercado com essa situação de inflexibilidade e uma demanda para um sistema PDV que corrija essa situação.

Definição do problema

Sistemas PDV tradicionais são inflexíveis, intolerantes a falhas e difíceis de integrar com sistemas fornecidos por terceiros. Isso leva a problemas no processamento das vendas a tempo, instituindo processos aperfeiçoados que não são compatíveis com o software, e dados de contabilidade e de estoque precisos e oportunos para apoiar medições e planejamento, entre outros interesses. Isso afeta os caixas, os gerentes de lojas, os administradores de sistema e a gerência corporativa.

Definição da posição do produto

Resumo sucinto sobre a quem o sistema se destina, suas características mais notáveis e o que o diferencia da competição.

Alternativas e competição...

Compreenda quem são os jogadores e quais são os seus problemas

Descrições dos interessados

Demografia do mercado...

Resumo dos interessados (não-usuários)...

Resumo dos usuários...

¹ N. de R.T.: PDA – Portable Digital Assistant (Assistente Digital Portátil). Foi mantida a sigla em inglês por ser assim conhecida nos meios técnicos brasileiros de computação.

Objetivos-chave de alto nível e problemas dos interessados

Consolidar a entrada pela Lista Um workshop de requisitos de um dia com especialistas no assunto e outros interessados no projeto do sistema, além de levantamentos realizados em vários pontos de venda, levou à identificação dos seguintes objetivos-chave e problemas:

*de Atores e
Objetivos e
a seção de
Interesses e
Interessa-
dos dos ca-
sos de uso*

Objetivo de Alto Nível	Prioridade	Problemas e Preocupações	Soluções Correntes
Processamento de vendas rápido, robusto e integrado	Alta	<p>Diminuição da velocidade, à medida que a carga aumenta.</p> <p>Perda da capacidade de processamento de vendas se há falhas nos componentes.</p> <p>Falta de informação atualizada e precisa da contabilidade e de outros sistemas por causa da não integração com os sistemas já existentes existentes de contabilidade, estoque e RH.</p> <p>Leva a dificuldades de medição e planejamento.</p> <p>Impossibilidade de personalizar regras de negócios para requisitos especiais de negócio.</p> <p>Dificuldade em adicionar novos tipos de terminais ou interfaces de usuário (por exemplo, PDAs móveis).</p>	<p>Produtos PDV já existentes fornecem o processamento básico de vendas, mas não tratam esses problemas.</p>
...

Objetivos do usuário

Esta pode ser a Lista Ator-Objetivo, criada durante a modelagem de casos de uso, ou um resumo mais compacto.

- *Caixa*: processar vendas, tratar devoluções, abrir, fechar
 - *Administrador do Sistema*: administrar usuários, administrar a segurança, administrar as tabelas do sistema
 - *Gerente*: iniciar e encerrar o sistema
 - *Sistema de Atividade de Vendas*: analisar dados de vendas
 - ...

Ambiente do usuário...

Resumido a partir do diagrama de ca-
sos de uso.

Diagramas de contexto existem em diferentes formatos com níveis de detalhamento variáveis, mas todos mostram os principais atores exter- nos relacionados com um sistema.

Perspectiva do produto

O PDV ProxGer normalmente residirá nas lojas; se forem usados terminais móveis, eles estarão nas proximidades da rede da loja, dentro ou fora dela, mas muito próximos. Eles fornecerão serviços aos usuários e colaborarão com outros sistemas, conforme indicado na Figura Visão-1.

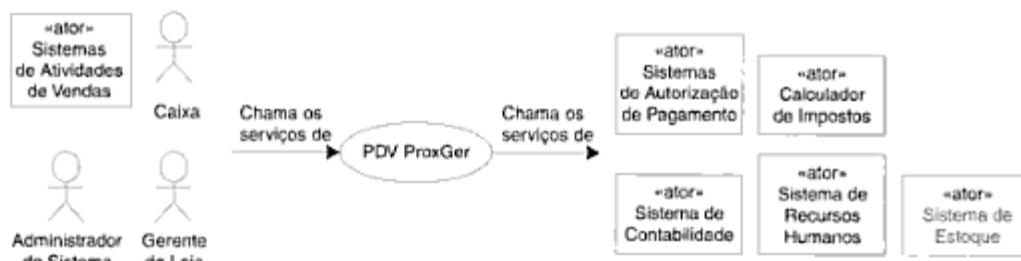


Figura Visão 1. Diagrama de contexto do sistema PDV ProxGer.

De modo similar à lista de Ator-Objetivo, esta tabela relaciona objetivos, benefícios e soluções, mas em um nível mais alto, não apenas relacionado com casos de uso.

Ela resume o valor e as qualidades que diferenciam o produto.

Característica que Suporta	Benefício para o Interessado
Funcionalmente, o sistema fornecerá todos os serviços comuns que uma organização de venda necessita, incluindo a captação de vendas, a autorização de pagamentos, o tratamento de devoluções, etc.	Serviços de ponto-de-venda rápidos e automatizados.
Detecção automática de falhas, mudando para processamento local os serviços que em um momento estão disponíveis.	Comunidade das vendas em caso de falha de componentes externos.
Regras de negócio "plugáveis" em vários pontos dos cenários durante o processamento das vendas.	Configuração flexível da lógica de negócios.
Transações em tempo real com sistemas de terceiros, usando protocolo-padrão da indústria.	Vendas realizadas em tempo hábil e com precisão, informações para contabilidade e estoque, para dar suporte a medições e planejamento.
...	...

Resumo dos benefícios

Hipóteses e dependências...

Custo e preços...

Licenças e instalação...

Conforme discutido ao lado, as características do sistema são um formato compacto para resumir a funcionalidade.

Resumo das características do sistema

- Captação de vendas.
- Autorização de pagamentos (crédito, débito, cheque).
- Administração do sistema: usuários, segurança, códigos e tabelas de constantes, etc.
- Processamento de vendas automático offline em caso de falha nos componentes externos.
- Transações em tempo real, baseadas nos padrões da indústria, com sistemas de terceiros, incluindo estoques, contabilidade, recursos humanos, calculadores de impostos e serviços de autorização de pagamentos.
- Definição e execução de regras de negócio personalizáveis e "plugáveis" em pontos fixos e comuns nos cenários de processamento.
- ...

Outros Requisitos e Restrições

Inclui restrições de projeto, usabilidade, confiabilidade, desempenho, capacidade de suporte, documentação, empacotamento, etc. Ver Especificação Suplementar e casos de uso.

7.7 Comentário: visão

Quando alguém se junta ao projeto, é útil poder dizer "Bem-vindo! Por favor leia as sete páginas sobre Visão no site da Web do projeto". Também é útil ter um resumo executivo que descreve rapidamente o projeto, fornecendo um contexto para os principais envolvidos estabelecerem uma visão comum do projeto.

A visão não deve ser longa, nem deve tentar descrever os requisitos efetivos em detalhe. Ela deve resumir algumas das informações do Modelo de Casos de Uso e da Especificação Suplementar.

Objetivos-chave de alto nível e problemas dos interessados

Esta seção resume os objetivos e problemas em alto nível – freqüentemente mais elevados do que os específicos dos casos de uso – e revela objetivos importantes não funcionais e de qualidade que podem pertencer a um caso de uso ou abranger vários, tais como:

- *É necessário processamento de vendas tolerante a falhas.*
- *É necessária a possibilidade de personalizar as regras de negócio.*

Diretriz: quais são os métodos de facilitação?

O trabalho de grupo criativo e investigativo ocorre especialmente durante atividades tais como definição do problema em alto nível e identificação dos objetivos. Seguem algumas técnicas úteis para facilitação (moderação) de grupos, com o intuito de descobrir os problemas e objetivos básicos e apoiar a geração de idéias e priorização: mapeamento da mente (mind mapping), criação da visão do produto em caixas (product vision box creation), diagramas de espinha de peixe (fishbone diagrams), diagramas de Pareto (Pareto diagrams), brainstorming, votação múltipla (multi-voting), votação pontual (dot voting), processo de grupo nominal (nominal group process), brainwriting e agrupamento por afinidade (affinity grouping). Pesquise-os na Web. Prefiro aplicar vários deles durante o mesmo workshop, para descobrir problemas e requisitos comuns a partir de diferentes ângulos.

Resumo de características do sistema

A simples listagem dos nomes dos casos de uso não é suficiente na Visão para detectar as principais características. Por quê?

- Eles são muito detalhados ou de baixo nível. As pessoas desejam um resumo breve das grandes idéias. Pode haver 30 ou 50 casos de uso.
- O nome do caso de uso pode esconder características interessantes e importantes que os interessados realmente desejam conhecer. Por exemplo, suponha que a descrição da funcionalidade da autorização de pagamento automatizada esteja embutida no caso de uso Processar Venda. Um leitor da lista de nomes dos casos de uso não pode afirmar se o sistema vai fazer autorização de pagamento.
- Algumas características notáveis abrangem mais de um caso de uso ou são ortogonais aos casos de uso. Por exemplo, durante o primeiro workshop de requisitos do ProxGer alguém poderia dizer “O sistema deve poder interagir com sistemas existentes de terceiros para contabilidade, estoque e cálculo de impostos”.

Portanto, uma maneira alternativa e complementar de expressar as funções do sistema é utilizar **características** ou, mais especificamente neste contexto, **características do sistema**, que são sentenças de alto nível, compactas, que resumem as funções do sistema. Em termos mais formais do PU, uma **característica do sistema** é “um serviço observável externamente fornecido pelo sistema que satisfaz diretamente a uma necessidade de um interessado no sistema” [Kruchten00].

Definição

Características são funções comportamentais que um sistema pode *fazer*. Deveriam passar no seguinte teste lingüístico:

O sistema faz <característica X>.

Por exemplo:

O sistema faz autorização de pagamentos.

As características funcionais do sistema devem ser contrastadas com vários tipos de requisitos não funcionais e com restrições, como: “*o sistema deve rodar sob Linux, deve ter 24/7 de disponibilidade e uma interface por tela sensível ao toque.*” Observe que esses não passam no teste lingüístico; por exemplo, *o sistema faz Linux*.

Diretriz: como escrever a lista de características?

Na Visão, “curto é bom” – na verdade, em qualquer documento.

Temos, aqui, um exemplo de descrição de características em alto nível para um grande projeto multi-sistema, do qual o PDV é apenas um elemento:

As características principais incluem:

- *Serviços PDV*
- *Gestão de estoque*
- *Compra pela Web*
- ...

É comum organizar uma hierarquia de dois níveis para as características do sistema. Entretanto, no documento Visão, mais do que dois níveis levam a um excesso de detalhes; o objetivo das características do sistema na Visão é resumir a funcionalidade, e não decompô-la em uma longa lista de elementos de granularidade fina. Um exemplo razoável, em termos de detalhe, seria:

As características principais incluem:

- *Serviços PDV:*
 - *captação de vendas*
 - *autorização de pagamentos*
 - ...
- *Gestão de estoque:*
 - *pedidos de reposição feitos automaticamente*
 - ...

Quantas características do sistema a Visão deveria conter?

Diretriz

É desejável que a Visão tenha menos de 10 características – mais do que isso não podem ser percebidas rapidamente. Caso tenha mais, considere agrupá-las e abstraí-las.

Diretriz: deve-se duplicar outros requisitos na Visão?

Na Visão, as características do sistema resumem, de forma breve, os requisitos funcionais, em geral, expressos em detalhes nos casos de uso. De maneira semelhante, a Visão *pode* resumir outros requisitos (por exemplo, confiabilidade e usabilidade) que são detalhados na Especificação Suplementar. Contudo, deve-se ter cuidado para evitar ser repetitivo.

Diretriz

Para outros requisitos, evite sua duplicação ou quase duplicação tanto na Visão como na Especificação Suplementar (ES). Em vez disso, registre-os somente na ES. Na Visão, direcione o leitor para a ES para ver os outros requisitos.

Diretriz: você deve escrever primeiro a Visão ou os casos de uso?

Não vale a pena ser rígido com relação à ordem. Enquanto os desenvolvedores estão colaborando para criar diferentes artefatos de requisitos, surge uma sinergia na qual trabalhar em um artefato influencia e ajuda a esclarecer um outro. Entretanto, uma sugestão de seqüência é:

1. Escreva uma primeira minuta da Visão.
2. Identifique os objetivos do usuário e os casos de uso correspondentes pelo nome.
3. Escreva alguns casos de uso em detalhe e inicie a Especificação Suplementar.
4. Refine a Visão, resumindo as informações a partir desses últimos.

7.8 Exemplo ProxGer: um Glossário (parcial)

Glossário

Histórico das Revisões

Versão	Data	Descrição	Autor
Minuta da concepção	10 de janeiro de 2031	Primeira minuta. A ser refinada, principalmente durante a elaboração.	Craig Larman

Definições

Termo	Definição e Informações	Formato	Regras de Validação	Sinônimos
Item	Um produto ou serviço para venda.			
Autorização de Pagamento	Validação por um serviço externo de autorização de pagamento que fará ou garantirá o pagamento para o vendedor.			
Solicitação de Autorização de Pagamento	Uma composição de elementos é enviada eletronicamente para um serviço de autorização, normalmente como uma sequência de caracteres. Os elementos incluem o ID da loja, o número da conta do cliente, a quantia e uma marca de tempo.			
CUP	Código numérico que identifica um produto. Geralmente é simbolizado por um código de barras colocado nos produtos. Ver www.uc-council.org para detalhes sobre formato e validação.	Código de 12 dígitos com diversas sub-partes.	Dígito 12 é um dígito de controle.	UPC -Universal Code Product ou Código Universal de Produto.
...	...			

7.9 Comentário: Glossário (dicionário de dados)

Na sua forma mais simples, o **Glossário** é uma lista de termos relevantes e suas definições. É surpreendentemente comum que um termo muitas vezes técnico ou específico do domínio seja usado de maneiras ligeiramente diferentes por diferentes interessados no projeto; é necessário solucionar isso para reduzir problemas de comunicação e de ambiguidades nos requisitos.

Diretriz

Comece o Glossário desde o início do projeto. Ele vai se tornar rapidamente um repositório útil de informação detalhada relativa a elementos de granularidade fina.

O Glossário como dicionário de dados

No PU, o Glossário também exerce o papel de um **dicionário de dados**, um documento que registra dados sobre os dados – ou seja, registra **metadados**. Durante a concepção, o glossário deve ser um documento simples de termos e descrições. Durante a elaboração, pode se expandir, tornando-se um dicionário de dados.

Os atributos dos termos podem incluir:

- sinônimos;
- descrição;
- formato (tipo, tamanho, unidade);

- relacionamentos com outros elementos;
- intervalo de valores válidos;
- regra de validação.

Observe que os intervalos de valores válidos e as regras de validação registradas no Glossário são requisitos que têm implicações no comportamento do sistema.

Diretriz: pode-se usar o Glossário para registrar termos compostos?

O Glossário não se destina somente a termos atômicos, como “preço do produto”. Ele pode e deve incluir elementos compostos, como “venda” (que inclui outros elementos, como data e localização) e apelidos usados para descrever uma coleção de dados transmitida entre atores nos casos de uso. Por exemplo, no caso de uso *Processar Venda*, considere a seguinte afirmação:

O sistema envia uma solicitação de autorização de pagamento para um Serviço de Autorização de Pagamentos externo e solicita sua aprovação.

“Solicitação de autorização de pagamento” é um apelido para um agregado de dados que necessita ser explicado no Glossário.

7.10 Exemplo ProxGer: regras de negócio (regras de domínio)

Regras de Domínio

Histórico de Revisões

Lista de Regras

Versão	Data	Descrição	Autor
Minuta da concepção	10 de janeiro de 2031	Primeira minuta. A ser refinada, principalmente durante a elaboração.	Craig Larman

(Ver também em separado as regras específicas da aplicação na Especificação Suplementar)

ID	Regra	Mutabilidade	Fonte
REGRA 1	Necessária assinatura para pagamentos a crédito.	A “assinatura” do comprador vai continuar a ser exigida, mas dentro de dois anos a maioria de nossos clientes desejará a captação da assinatura por meio de um dispositivo digital de captação, e dentro de cinco anos desejamos ter demanda para apoio do novo código único digital de “assinatura” já apoiado pela lei dos EUA.	Política virtualmente adotada por todas as companhias de autorização de crédito.

ID	Regra	Mutabilidade	Fonte
REGRA 2	Regras de impostos. Vendas exigem impostos adicionados. Ver disposições governamentais para os detalhes atuais.	Alta. Leis de impostos mudam anualmente em todos os níveis governamentais.	Lei
REGRA 3	Restituições de pagamento a crédito podem somente ser pagas como crédito à conta de crédito do comprador, não em dinheiro.	Baixa.	Política da companhia de autorização de crédito.

7.11 Comentário: regras de domínio

Regras de Domínio [Ross97, GK00] ditam como um domínio ou negócio pode operar. Elas não são requisitos de qualquer aplicação, apesar dos requisitos de uma aplicação serem freqüentemente influenciados pelas regras de domínio. As políticas da companhia, leis físicas (por exemplo, como o petróleo flui no subsolo) e leis governamentais são regras de domínio comuns.

Elas são comumente chamadas **Regras de Negócio**, que é o tipo mais comum, mas esse termo é pobre, pois muitas aplicações de software são para problemas não relativos a negócios, tais como simulação das condições meteorológicas ou logística militar. Uma simulação meteorológica tem “regras de domínio” relativas a leis físicas e seus relacionamentos, que influenciam os requisitos da aplicação.

É útil identificar e registrar regras de domínio em um artefato separado independente da aplicação – que o PU chama de artefato de Regras de Negócio – de modo que essa análise possa ser compartilhada e reusada por toda a organização e por todos os projetos, em vez de ficar enterrada dentro de um documento específico de um projeto.

As regras podem ajudar a esclarecer ambigüidades nos casos de uso, que enfatizam o fluxo da estória em vez dos detalhes. Por exemplo, no PDV ProxGer, se alguém perguntar se o caso de uso *Processar Venda* deve ser escrito como uma alternativa para permitir pagamentos a crédito sem captação de assinaturas, existe uma regra de negócio (REGRA 1) que esclarece que isso não será permitido por nenhuma companhia de autorização de crédito.

7.12 Processo: requisitos evolutivos em métodos iterativos

Como tem sido repetidamente enfatizado (por ser crucial, embora muitas vezes ignorado) em métodos iterativos, inclusive o PU, esses requisitos não são totalmente analisados e redigidos próximo ao início do projeto. Pelo contrário, eles evoluem durante uma série de workshops de requisitos (por exemplo), intercalados com programação e teste com qualidade de produção logo no início do projeto. A implementação desse desenvolvimento feito logo no início do projeto é usada para refinar as especificações.

Como foi feito no capítulo de casos de uso, a Tabela 7.1 resume uma amostra de artefatos e sua possível seqüência temporal no PU. Usualmente, a maioria dos artefatos de requisitos são iniciados na concepção e trabalhados principalmente ao longo da elaboração.

Tabela 7.1 Amostra dos Artefatos do PU e seqüência temporal (i = início, r = refinado)

Disciplina	Artefato Iteração →	Concep. II	Elab. EI,En	Const. C1..Cn	Trans. T1..T2
Modelagem de Negócio	Modelo de domínio		i		
Requisitos	Modelo de caso de uso	i	r		
	Visão	i	r		
	Especializações Suplementares	i	r		
	Glossário	i	r		
	Regras de negócio	i	r		
Projeto	Modelo de projeto		i	r	
	Documento de arquitetura de software		i		
	Modelo de dados		i	r	

Concepção

Os interessados no projeto precisam decidir se ele merece uma investigação séria; essa investigação real ocorre durante a fase de elaboração, não durante a concepção. Durante a concepção, a Visão resume a idéia do projeto, de modo a ajudar os tomadores de decisão a determinar se vale a pena continuar, e por onde começar.

Como a maior parte da análise de requisitos ocorre durante a fase de elaboração, a Especificação Suplementar deve ser pouco desenvolvida durante a concepção, destacando os atributos de qualidade dignos de nota que exponham os riscos e desafios principais (por exemplo, o PDV ProxGer deve ter condições de recuperação quando houver falha de serviços externos).

A informação de entrada para esses artefatos poderia ser gerada durante um workshop de requisitos na fase de concepção.

Elaboração

Ao longo das iterações de elaboração, a “visão” e a Visão são refinadas, com base na realimentação obtida a partir da construção incremental de partes do sistema, da sua adaptação e de múltiplos workshops de requisitos ao longo de várias iterações de desenvolvimento.

Por meio de uma investigação constante de requisitos e do desenvolvimento iterativo, os outros requisitos se tornarão mais claros e poderão ser registrados na ES.

Ao fim da elaboração, é viável ter casos de uso, uma Especificação Suplementar e uma Visão que refletem, de forma razoável, as principais características e outros requisitos, já estabilizados, que devem ser completados para entrega. Entretanto, não se deve congelar a Especificação Suplementar e a Visão, nem “assinar embaixo” delas como se fossem uma especificação fixa; a adaptação – e não a rigidez – é um valor fundamental do desenvolvimento iterativo e do PU.

Para esclarecer este comentário sobre “congelar e assinar embaixo”: é perfeitamente razoável – no fim da elaboração – estabelecer um acordo com os interessados no projeto sobre o que será feito no restante do projeto, além de assumir compromissos (talvez contratuais) com relação a requisitos e cronogramas. Em algum ponto (no PU, o fim da fase de elaboração), precisamos ter uma idéia confiável sobre “o quê, quanto e quando”. Nesse sentido, um acordo formal sobre os requisitos é normal e esperado. É também necessário ter um processo de controle de mudanças (uma das práticas explicitamente recomendadas no PU), de forma que as mudanças nos requisitos sejam consideradas e aprovadas formalmente, em vez de termos mudanças caóticas e não controladas.

Assim, várias idéias estão implícitas no comentário “congelar e assinar embaixo”:

- No desenvolvimento iterativo e no PU, entende-se que não importa quanto esforço e trabalho dediquemos à especificação dos requisitos, algumas mudanças serão inevitáveis e devem ser aceitáveis. Essas mudanças podem ser uma oportunidade de última hora para se efetuar melhorias no sistema, que ofereçam aos seus proprietários uma vantagem competitiva ou ainda mudanças para a obtenção de uma melhor percepção.
- No desenvolvimento iterativo, é um valor fundamental que tenhamos um envolvimento contínuo dos interessados no projeto para avaliá-lo, fornecer realimentação e direcionar o projeto como eles realmente desejam. Não é benéfico para os interessados que eles simplesmente não se comprometam com um engajamento efetivo no projeto, concordando com um conjunto de requisitos congelados e esperando pelo produto acabado, porque eles provavelmente não obterão o que de fato necessitam.

Construção

Na altura da fase de construção, os principais requisitos – tanto funcionais como não funcionais – devem estar estabilizados – não finalizados, mas estabelecidos e sujeitos somente a perturbações menores. Portanto, é improvável que a ES e a Visão sofram muitas mudanças nesta fase.

7.13 Leituras recomendadas

A maioria dos livros sobre arquitetura de software inclui discussões sobre análise de requisitos para atributos de qualidade da aplicação, uma vez que esses requisitos de qualidade tendem a influenciar fortemente o projeto da arquitetura. Um exemplo desses livros é *Software Architecture in Practice* [BCK98].

As regras de negócio recebem um tratamento exaustivo no livro *The Business Rule Book* [Ross97]. O livro apresenta uma teoria de regras de negócio ampla, profunda e cuidadosamente elaborada, mas o método não está bem conectado com outras técnicas modernas de requisitos, como os casos de uso ou o desenvolvimento iterativo.

No PU, o trabalho sobre Visão e Especificação Suplementar é uma atividade da disciplina de requisitos, que poderia ser iniciada durante um workshop de requisitos, junto com a análise de casos de uso. Um bom guia para conduzir um workshop é *Requirements by Collaboration: Workshops for Defining Needs*, de Ellen Gottesdiener.

O produto RUP online contém gabaritos para os artefatos discutidos neste capítulo.

Na Web, gabaritos para as especificações estão disponíveis em muitas fontes, como, por exemplo, os gabaritos ReadySET em readyset.tigris.org.

PARTE III

ELABORAÇÃO:

ITERAÇÃO 1 –

CONCEITOS

BÁSICOS

ITERAÇÃO 1 – CONCEITOS BÁSICOS

O duro e rijo quebra. O flexível prepondera.
– Tao Te Ching

Objetivos

- Definir a primeira iteração na fase de elaboração.
- Motivar os capítulos seguintes.
- Descrever os conceitos-chave das fases de concepção e elaboração.

Introdução

Este capítulo resume os requisitos da iteração 1 dos estudos de caso e depois discute resumidamente as idéias de processo das fases de concepção e elaboração. A leitura dos requisitos escolhidos é importante para entender o que está sendo tratado nos capítulos seguintes a esta iteração; a leitura do restante depende de sua necessidade ou interesse nos tópicos do processo iterativo.

O que vem a seguir?

Observados outros requisitos além dos casos de uso, este capítulo define o objetivo da iteração 1. O seguinte explora a criação de um modelo de domínio OO que ilustra os conceitos relevantes do domínio.



8.1 Requisitos e ênfase da iteração 1: habilidades fundamentais em A/POO

Nestes estudos de caso, a Iteração 1 da fase de elaboração enfatiza uma gama de habilidades fundamentais e comuns em A/POO usadas na construção de sistemas baseados em objetos. Muitas outras habilidades e passos – por exemplo, o projeto de banco de dados, engenharia de usabilidade e projeto da IU – são necessárias para construir um software; porém, estão fora do objetivo desta introdução, que é focada na A/POO e aplicação da UML.

Iterações do livro vs. Iterações de um projeto real

A iteração 1 dos estudos de caso deste livro é dirigida por objetivos de aprendizado, em vez de objetivos reais do projeto. Assim sendo, a iteração 1 não é centrada na arquitetura ou guiada por risco. Em um projeto PU, iríamos tratar primeiro das coisas difíceis, de alto risco. Mas, no contexto de um livro para ajudar pessoas a aprenderem a A/POO e UML fundamentais, desejamos começar com os tópicos mais fáceis.

PDV ProxGer

A seguir, temos os requisitos para a primeira iteração da aplicação PDV ProxGer:

- Implementar um cenário-chave e básico do caso de uso *Processar Venda*: fazer a entrada dos itens, recebendo o pagamento em dinheiro.
- Implementar o caso de uso *Iniciar*, conforme necessário, para dar suporte às necessidades de início da iteração.
- Nada sofisticado ou complexo é tratado, apenas um simples cenário de um caminho em que tudo dá certo e o projeto e a implementação para satisfazê-lo.
- Não existe colaboração com serviços externos, como o calculador de impostos ou um banco de dados de produtos.
- Não são aplicadas regras complexas para estabelecimento de preços.

O projeto e a implementação da IU correspondente também deve ser feito, porém não será abordado aqui em nenhum detalhe.

Banco Imobiliário

A seguir, temos os requisitos para a primeira iteração da aplicação do Banco Imobiliário:

- Implementar um cenário-chave e básico do caso de uso *Jogar Banco Imobiliário*: jogadores movendo-se pelas casas do tabuleiro.
- Implementar um caso de uso *Iniciar*, conforme necessário, para dar suporte às necessidades de início da iteração.

- De dois a oito jogadores podem jogar.
- Um jogo é jogado como uma série de rodadas. Durante uma rodada, cada jogador tem sua vez. Em cada vez um jogador avança sua peça no sentido horário no tabuleiro, por um número de casas igual à soma dos números sorteados pelo lançamento de dois dados de seis faces.
- O jogo vai até 20 rodadas apenas.
- Depois que os dados são lançados, o nome do jogador e o resultado são mostrados. Quando o jogador se move e chega a uma casa, o nome do jogador e o nome da casa em que o jogador chega são mostrados.
- Na iteração 1 não existe dinheiro, nem ganhador ou perdedor, nem propriedades a comprar ou aluguéis a pagar e nem casas especiais de quaisquer tipos.
- Cada casa tem um nome. Cada jogador começa o jogo com a sua peça colocada na casa chamada “Vá”. Os nomes das casas serão: Vá, Casa1, Casa2,..., Casa 39.
- Executar o jogo como uma simulação que não requer entrada do usuário, a não ser do número de jogadores.

As iterações subsequentes serão baseadas nesses itens.

No desenvolvimento iterativo não se implementam todos os requisitos de uma só vez

Observe que esses requisitos para a iteração 1 são subconjuntos dos requisitos ou casos de uso completos. Por exemplo, os requisitos da iteração 1 do PDV ProxGer são uma versão simplificada do caso de uso *Processar Venda*; eles descrevem um cenário simples de venda à vista em dinheiro.

Observe também que não foi feita toda a análise de requisitos para o sistema PDV ProxGer, foi analisado em detalhes apenas o caso de uso *Processar Venda*; muitos outros ainda não foram analisados.

Esse é um entendimento-chave nos métodos de ciclo de vida iterativo (tais como, PU, XP, SCRUM, etc.). Começa-se a programar e testar com qualidade de produção para atender a um subconjunto dos requisitos e esse desenvolvimento é iniciado *antes* que a análise de requisitos seja completada – ao contrário do processo em cascata.

Desenvolvimento incremental para o mesmo caso de uso ao longo de iterações

Observe que nem todos os requisitos do caso de uso *Processar Venda* estão sendo tratados na iteração 1. É comum trabalhar em vários cenários do mesmo caso de uso ao longo de várias iterações e estender o sistema de modo gradual para finalmente tratar toda a funcionalidade exigida (ver Figura 8.1). Por outro lado, os casos de uso simples e curtos podem ser completados em uma iteração.

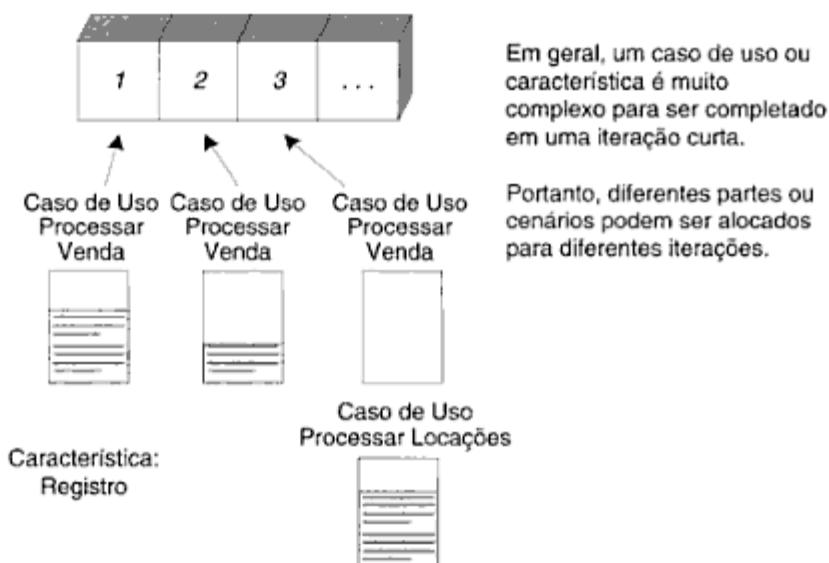


Figura 8.1 A implementação de um caso de uso pode se estender por várias iterações.

8.2 Processo: concepção e elaboração

Em termos do PU e dos nossos estudos de caso, imagine que terminamos a fase de concepção e estamos entrando na fase de elaboração.

O que aconteceu na concepção?

A fase de **concepção** dos estudos de caso pode durar apenas uma semana. Como essa *não* é a fase de requisitos do projeto, os artefatos criados devem ser curtos e incompletos, a fase deve ser rápida e a investigação deve ser feita por alto.

A concepção é um passo curto para a elaboração. Ela determina a viabilidade, risco e escopo básicos para decidir se o projeto merece investigação mais séria. Nem todas as atividades que poderiam ocorrer normalmente na concepção foram abordadas; essa exploração dá ênfase a artefatos orientados para os requisitos. Algumas atividades e artefatos prováveis da concepção incluem:

- Um seminário curto de requisitos.
- A maioria dos atores, objetivos e casos de uso deve ser nomeada.
- A maioria dos casos de uso deve ser escrita em formato resumido; 10 a 20% dos casos de uso devem ser escritos com detalhes completos para melhorar a compreensão do escopo e da complexidade.
- Os requisitos mais influentes e mais arriscados quanto à qualidade devem ser identificados.
- A versão 1 da Visão e da Especificação Suplementar deve ser escrita.
- Lista de riscos:
 - Por exemplo, os líderes desejam de fato uma demonstração na feira de comércio POSWorld em Hamburgo dentro de 18 meses. Entretanto, o

esforço para elaborar uma demonstração não pode ser sequer estimado até que tenhamos uma investigação mais profunda.

- São feitos protótipos para demonstração de conceitos técnicos, bem como outras investigações para explorar a viabilidade técnica de requisitos especiais (“O Java Swing funciona adequadamente em monitores de vídeo com telas sensíveis ao toque?”).
- São feitos protótipos voltados para a interface com o usuário para esclarecer a visão de requisitos funcionais.
- Recomendações sobre quais componentes comprar/construir/reusar, que serão refinadas na elaboração.
 - Por exemplo, uma recomendação para comprar um pacote de cálculos de impostos.
- Proposta em alto nível de uma arquitetura *candidata* e proposta de componentes.
 - Essa não é uma descrição detalhada da arquitetura e não pretende ser definitiva ou correta. Ao invés disso, é uma breve especulação para servir de ponto de partida na investigação conduzida durante a elaboração. Por exemplo, “uma aplicação Java do lado do cliente, nenhum servidor de aplicação, Oracle para o banco de dados...”. Na elaboração, pode-se demonstrar que isso vale a pena ou descobrir que não é uma boa idéia e que deve ser rejeitada.
- Um plano para a primeira iteração.
- Lista de ferramentas candidatas.

Em direção à elaboração

A **elaboração** é a série inicial de iterações durante a qual, em um projeto padrão:

- A arquitetura central e de alto risco do software é programada e testada.
- A maioria dos requisitos é descoberta e estabilizada.
- Os principais riscos são mitigados ou retirados.

A elaboração é a série inicial de iterações durante a qual a equipe faz uma investigação séria, implementa (programa e testa) a arquitetura central, esclarece a maioria dos requisitos e ataca os problemas de alto risco. No PU, a palavra “risco” inclui o valor do negócio. Portanto, o trabalho inicial pode incluir a implementação de cenários considerados importantes, porém não são especialmente arriscados em termos técnicos.

A elaboração freqüentemente consiste em duas ou mais iterações; recomenda-se que cada uma tenha entre duas e seis semanas; prefira as versões curtas, a menos que a equipe seja de grande porte. Cada iteração é limitada em tempo, o que significa que a data de término está fixada.

A elaboração não é uma fase de projeto ou uma fase na qual os modelos são desenvolvidos completamente, como a preparação para a implementação na etapa de construção – isto seria um exemplo de superposição das idéias de ciclos de vida em cascata ao desenvolvimento iterativo e ao PU.

Durante essa fase, não se está criando protótipos descartáveis; ao contrário, o código e o projeto são partes do sistema final, isto é, com qualidade de produção. Em algumas descrições do PU, o termo potencialmente mal compreendido “**protótipo arquitetural**” é usado para descrever o sistema parcial. Isso não significa que é um protótipo, no sentido de ser um experimento descartável; no PU, ele significa um subconjunto de produção do sistema final. Mais comumente, é chamado de **arquitetura executável** ou **marco referencial da arquitetura**.

A elaboração em uma frase:

Construa a arquitetura central, resolva os elementos de alto risco, defina a maioria dos requisitos e estime o cronograma geral e os recursos.

Algumas idéias-chave e melhores práticas que se manifestarão na elaboração incluem:

- Fazer iterações curtas dirigidas pelos riscos e limitadas em tempo.
- Começar a programar cedo.
- Projetar de maneira adaptativa, implementar e testar a parte central e as partes de risco da arquitetura.
- Começar a testar cedo, com freqüência e de maneira realista.
- Adaptar de acordo com a realimentação obtida dos testes, usuários e desenvolvedores.
- Escrever a maioria dos casos de uso e outros requisitos em detalhes, por meio de uma série de seminários – uma vez para cada iteração da elaboração.

Que artefatos podem ser iniciados na elaboração?

A Tabela 8.1 lista *amostras* de artefatos que podem ser *iniciados* na elaboração e indica os problemas que eles tratam. Capítulos subsequentes examinarão alguns de forma mais detalhada, especialmente o Modelo de Domínio e o Modelo de Projeto. Para sintetizar, a tabela exclui artefatos que podem ter sido iniciados na concepção; ela introduz artefatos cuja probabilidade de iniciar é maior na elaboração. Observe que esses não serão completados em uma iteração; ao contrário, serão refinados ao longo de uma série de iterações.

Você sabe que não compreendeu a elaboração quando...

- Ela tem mais do que “uns poucos” meses de duração para a maioria dos projetos.
- Ela tem somente uma iteração (com raras exceções para problemas bem compreendidos).
- A maioria dos requisitos foi definida antes da elaboração.
- Os elementos de risco e a arquitetura central não estão sendo tratados.

Artefato	Comentário
Modelo de Domínio	Esta é uma visualização dos conceitos do domínio; é similar a um modelo estático de informação das entidades do domínio.
Modelo de Projeto	Este é um conjunto de diagramas que descreve o projeto lógico. Inclui, dentre outros, diagramas de classes de software, de interações entre projetos e de pacotes (packages) e outros.
Documento de Arquitetura de Software	Um auxílio para o aprendizado que resume os problemas-chave de arquitetura e sua solução no projeto. É um resumo das idéias de projeto mais notáveis e do motivo da sua adoção no sistema.
Modelo de Dados	Este modelo inclui os esquemas de bancos de dados e as estratégias de mapeamento entre as representações de objetos e não-objetos.
Modelo de Teste	Uma descrição do que será testado e como.
Storyboards de Casos de Uso e Protótipos IU	Uma descrição da interface do usuário, de trajetórias de navegação, modelos de utilização, etc.

- Ela não resulta em uma arquitetura *executável*; não há programação de código destinado à produção.
- Ela é primordialmente uma fase de requisitos ou projeto, que precede uma fase de implementação na construção.
- Existe a tentativa de executar um projeto completo e cuidadoso antes de programar.
- A realimentação e a adaptação são mínimas; os usuários não estão continuamente envolvidos na avaliação e na realimentação.
- Não há atividade de teste precoce e realista.
- A arquitetura é finalizada de modo especulativo antes de programar.
- Ela é considerada um passo para executar a programação de demonstrações de conceito, não a programação da arquitetura central executável já destinada à produção.

Se o projeto exibir esses sintomas, a fase de elaboração não terá sido compreendida e o pensamento em cascata foi superposto ao PU.

8.3 Processo: planejamento da próxima iteração

Planejamento e gestão de projetos são tópicos importantes, porém vastos. Algumas das idéias-chave são aqui apresentadas brevemente e há sugestões adicionais começando na págs. 665-666.

Organize os requisitos e as iterações por risco, cobertura e criticalidade.

- O Risco inclui tanto a complexidade técnica quanto outros fatores, como a incerteza quanto ao esforço necessário ou à usabilidade.

- A **Cobertura** implica que todas as partes principais do sistema são pelo menos tocadas nas iterações iniciais – talvez uma implementação “ampla e superficial” que abranja muitos componentes.
- A **Criticalidade** se refere a funções que o cliente considera de alto valor de negócio.

Esses critérios são usados para priorizar o trabalho ao longo das iterações. Os casos de uso ou cenários de casos de uso são priorizados para implementação – as iterações iniciais implementam os cenários com classificação mais alta. Além disso, alguns requisitos são expressos como características em alto nível não relacionadas com qualquer caso de uso em particular, como, por exemplo, um serviço de controle de acesso. Esses também são priorizados.

A priorização é feita antes da iteração 1, mas também novamente antes da iteração 2, e assim por diante, à medida que novos requisitos e novas descobertas influenciam a ordem. Ou seja, o plano é *adaptativo*, em vez de congelado de modo especulativo logo no começo do projeto. Em geral, um agrupamento de requisitos acabará surgindo com base em alguma técnica colaborativa de priorização. Por exemplo:

Classificação	Requisito (Caso de Uso ou Característica)	Comentário
Alto	Processar Venda Controle de Acesso ...	Pontuação alta em todos os critérios. Difuso. Difícil de adicionar mais tarde. ...
Médio	Manutenção de Usuários ...	Afeta o subdomínio de segurança. ...
Baixo

Com base nessa priorização, vemos que alguns cenários-chave arquiteturalmente significativos do caso de uso *Processar Venda* deveriam ser tratados nas iterações iniciais. Essa lista não é exaustiva; outros requisitos também serão tratados. Além disso, um caso de uso *Iniciar* implícito ou explícito será trabalhado em cada iteração, para atender às necessidades de início do sistema.

MODELOS DE DOMÍNIO

Está tudo bem na prática, mas nunca funcionará na teoria.
– máxima gerencial, anônima

Objetivos

- Identificar classes conceituais relacionadas com os requisitos da iteração corrente.
- Criar um modelo de domínio inicial.
- Modelar os atributos e associações adequadas.

Introdução

Modelagem
de domínio
mais avan-
çada
(pág. 511)

Um modelo de domínio é o modelo mais importante – e clássico – em *análise OO*.¹ Ele ilustra importantes conceitos em um domínio. Pode agir como fonte de inspiração para projetar alguns objetos de software e será um dado de entrada para vários artefatos explorados nos estudos de caso. Este capítulo também mostra o valor do conhecimento de A/POO com notação UML; a notação básica é trivial, mas há diretrizes de modelagem sutis para obter um modelo útil – especializar-se nela pode levar semanas ou meses. Este capítulo explora as habilidades básicas para a criação de modelos de domínio.

O que vem a seguir?

Tendo sido estabelecidos os objetivos do trabalho para a iteração 1, este capítulo explora um modelo de domínio parcial. O seguinte examina as operações específicas do sistema que estão implícitas nos cenários dos casos de uso que estão sendo projetadas nesta iteração.



¹ Os casos de uso são um artefato importante da análise de requisitos, porém não são realmente orientados a *objetos*. Eles enfatizam uma visão de atividade.

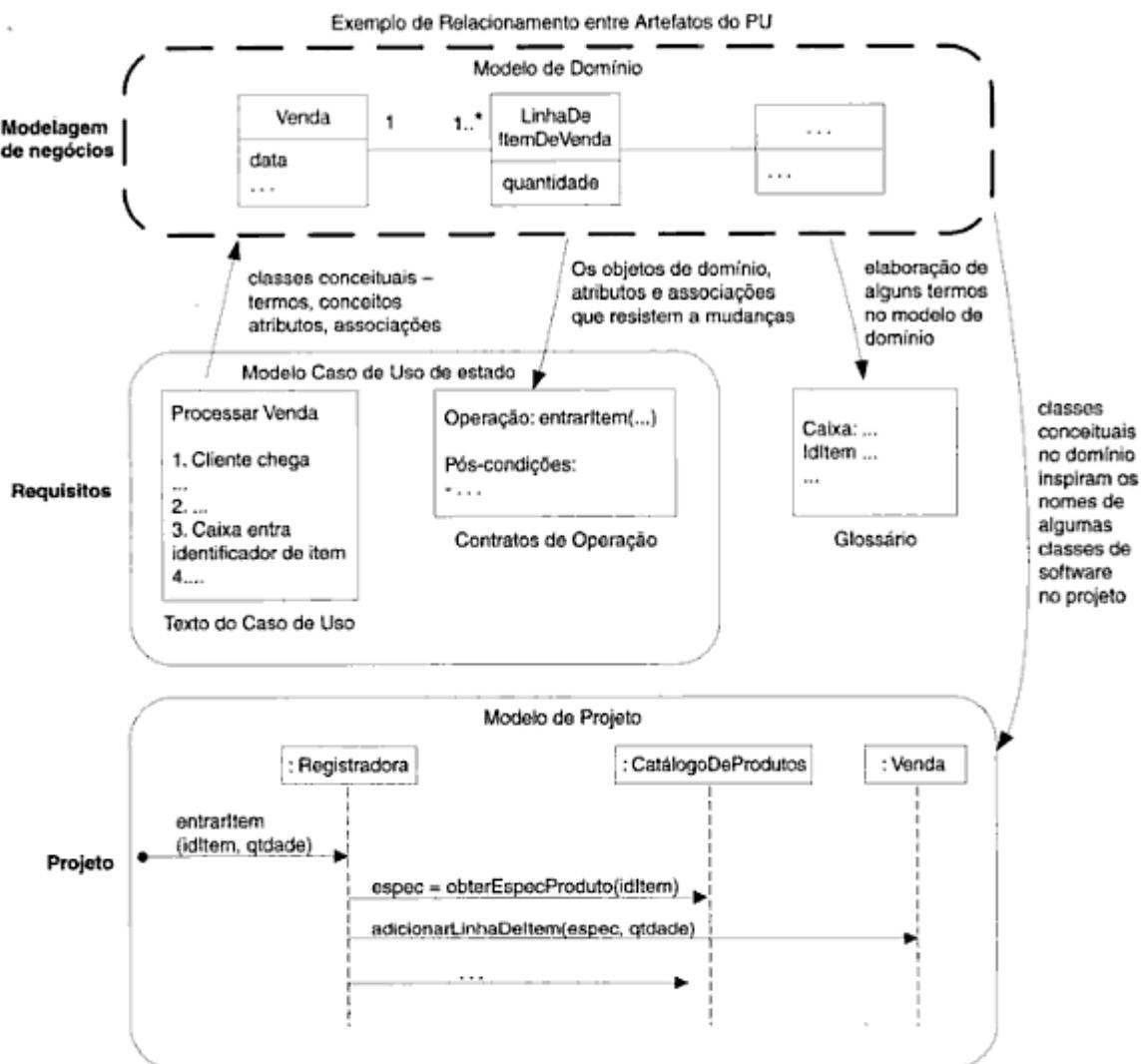


Figura 9.1 Amostra da influência entre artefatos do PU.

Como ocorre com todas as coisas no espírito da modelagem ágil e do PU, um modelo de domínio é opcional. A influência entre os artefatos do PU, enfatizando um modelo de domínio, é mostrada na Figura 9.1. Limitado pelos cenários de caso de uso em desenvolvimento na atual iteração, o modelo de domínio pode sofrer evolução para mostrar os conceitos importantes a ele relacionados. Os conceitos dos casos de uso relacionados e a visão aprofundada dos especialistas vão servir de entrada para sua criação. O modelo, por sua vez, pode influenciar os contratos de operação, o glossário e o modelo de projeto, especialmente os objetos de software na **camada de domínio** do Modelo de Projeto.

Camada de domínio
(pág. 162)

9.1 Exemplo

A Figura 9.2 mostra um modelo de domínio parcial desenhado usando a notação UML para **diagrama de classes**. Ele ilustra que as **classes conceituais Pagamento** e **Venda** são significativas neste domínio, que um **Pagamento** está relacionado a uma

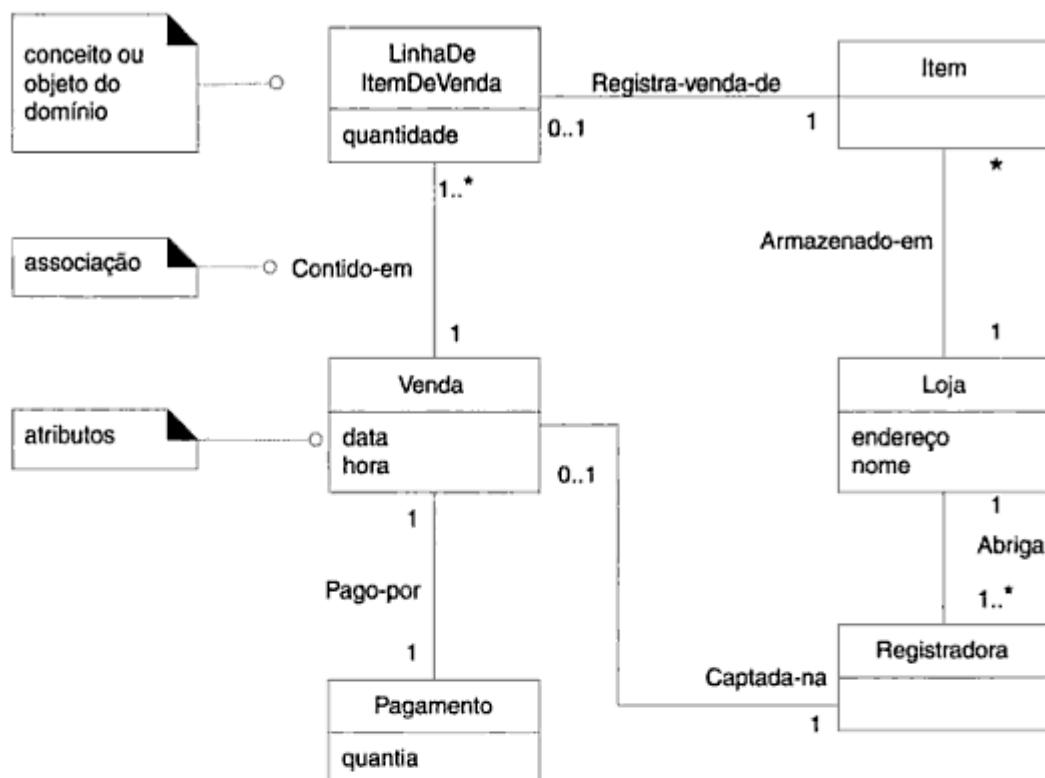


Figura 9.2 Modelo de domínio parcial – um dicionário visual.

Venda de maneira importante e que uma *Venda* tem uma data e uma hora de ocorrência, atributos de informação importantes.

Perspectiva conceitual (págs. 39-40) A aplicação da notação UML para diagrama de classes a um modelo de domínio produz um modelo de *perspectiva conceitual*.

A identificação de um conjunto rico de classes conceituais é essencial na análise OO. Se isso for feito com habilidade e *pouco* investimento de tempo (digamos, não mais que algumas horas em cada iteração inicial), o esforço geralmente é recompensado durante o projeto, quando há maior apoio ao entendimento e comunicação.

Diretriz

Evite um esforço grande de modelagem pensando na abordagem em cascata para fazer um modelo de domínio completo ou “correto” – ele nunca será nenhum dos dois e esses esforços excessivos de modelagem levam à *paralisia de análise*, com pequeno ou nenhum retorno do investimento.

9.2 O que é um modelo de domínio?

O passo mais essencialmente orientado a *objetos* na análise é a decomposição de um domínio em conceitos ou objetos importantes.

Um **modelo de domínio** é uma representação *visual* de classes conceituais, ou objetos do mundo real, em um domínio [MO95, Fowler96]. Modelos de domínio também têm sido chamados de **modelos conceituais** (o termo usado na primeira edição deste livro), **modelos de objetos do domínio** e **modelos de objetos da análise**.²

Definição

No PU, o termo “Modelo de Domínio” significa uma representação de classes conceituais do mundo real, não de objetos de software. O termo *não* significa um conjunto de diagramas que descreve classes de software, a camada de domínio de uma arquitetura de software ou objetos de software com responsabilidades.

O PU define o Modelo de Domínio³ como um dos artefatos que podem ser criados na disciplina de Modelagem de Negócios. Mais precisamente, o Modelo de Domínio do PU é uma especialização do **Modelo de Objetos de Negócio (Business Object Model – BOM)** do PU “focalizando a explicação de ‘coisas’ e produtos importantes a um domínio de negócio” [RUP]. Isto é, um Modelo de Domínio enfoca um domínio, como coisas relacionadas a PDV. O BOM mais amplo, não abordado neste texto introdutório e cuja criação não incentivo (porque pode levar a muita modelagem inicial), é um modelo multidomínio expandido, freqüentemente muito grande e difícil de criar, que cobre *tudo* o negócio e todos os seus subdomínios.

Aplicando a notação UML, um modelo de domínio é ilustrado com um conjunto de **diagramas de classes** em que nenhuma operação (assinatura de método) é definida. Ele fornece uma *perspectiva conceitual*. Pode mostrar:

- objetos do domínio ou classes conceituais;
- associações entre classes conceituais;
- atributos de classes conceituais.

Definição: por que chamar um modelo de domínio de “dicionário visual”?

Por favor, reflita um instante sobre a Figura 9.2. Veja como ela visualiza e relaciona palavras ou conceitos do domínio. Também ilustra uma *abstração* das classes conceituais, visto que há muitas outras coisas que se desejará comunicar sobre registradoras,⁴ vendas, etc.

² Eles também estão relacionados a modelos conceituais entidades-relacionamentos, os quais são capazes de mostrar visões puramente conceituais de domínios, mas que têm sido amplamente re-interpretados como modelo de dados para o projeto de banco de dados. Os modelos de domínio não são modelos de dados.

³ As letras maiúsculas em “Modelo de Domínio” são usadas para enfatizá-lo como um modelo oficial definido no PU, em oposição ao conceito geral bem conhecido de “modelos de domínio”.

⁴ N. de R.T.: Registradora refere-se à caixa registradora, isto é, ao dispositivo eletroeletrônico utilizado pelo Caixa para registrar as mercadorias adquiridas. Posteriormente, utilizaremos Caixa para nos referir à pessoa que opera a Registradora.

A informação que ela ilustra (usando a notação UML) poderia, de forma alternativa, ter sido expressa por texto em prosa (no Glossário PU). Entretanto, é fácil compreender os termos e, especialmente, seus relacionamentos em uma linguagem visual, uma vez que é mais fácil para nosso cérebro entender elementos visuais e conexões lineares.

Assim, o modelo de domínio é um *dicionário visual* das abstrações relevantes, do vocabulário do domínio e do conteúdo de informação do domínio.

Definição: um modelo de domínio é uma imagem de objetos de negócio de software?

Um modelo de domínio do PU, como o mostrado na Figura 9.3, é uma descrição de coisas em uma situação real do domínio de interesse, *não* de objetos de software, tais como classes Java ou C#, ou objetos de software com responsabilidades (ver Figura 9.4). Portanto, os seguintes elementos não são adequados em um modelo de domínio:

- Artefatos de software, como uma janela ou um banco de dados, a menos que o domínio que esteja sendo modelado seja de conceitos de software, por exemplo, um modelo de interfaces gráficas com o usuário.
- Responsabilidades ou métodos.⁴

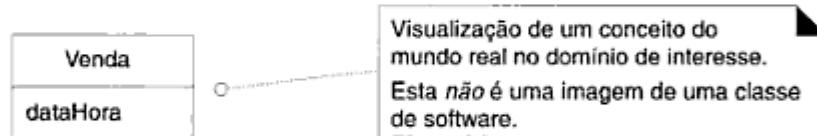


Figura 9.3 Um modelo de domínio mostra classes conceituais de uma situação real, não classes de software.



Figura 9.4 Um modelo de domínio não mostra artefatos ou classes de software.

⁴ Na modelagem de objetos, geralmente falamos de responsabilidades relacionadas a objetos de software. E métodos são puramente um conceito de software. Entretanto, o modelo de domínio descreve conceitos de situações reais, não de objetos de software. A consideração das responsabilidades durante a fase de projeto é muito importante; mas não faz parte do modelo de domínio.

Definição: quais são dois significados tradicionais de “modelo de domínio”?

No PU e, consequentemente, neste capítulo, “Modelo de Domínio” é uma perspectiva conceitual de objetos em uma situação real do mundo, não uma perspectiva de software. Mas o termo é sobrecarregado; também tem sido usado (especialmente na comunidade Smalltalk em que fiz a maior parte do meu trabalho inicial de desenvolvimento, na década de 1980) com o significado de “a camada de domínio dos objetos de software”, isto é, a camada de objetos de software abaixo da camada de apresentação ou IU, composta de **objetos de domínio** – objetos de software que representam coisas no espaço do domínio do problema com métodos relacionados à “lógica de negócios” ou “lógica de domínio”. Por exemplo, uma classe de software *Tabuleiro* com um método *obterACasa*.

Qual definição é correta? Bem, todas elas! O termo tem antigos usos em diferentes comunidades com diferentes significados.

Tenho visto muita confusão gerada por pessoas usando o termo de modos diferentes sem explicar qual significado é pretendido e sem reconhecer que outros podem estar usando-o de modo diferente.

Neste livro geralmente escreverei **camada de domínio** para indicar o segundo significado de modelo de domínio, que é o orientado a software, por ser o mais comum.

Definição: o que são classes conceituais?

O modelo de domínio ilustra classes conceituais ou vocabulário no domínio. De modo informal, uma **classe conceitual** é uma idéia, uma coisa ou um objeto. Mais formalmente, uma classe conceitual pode ser considerada em termos do seu símbolo, da sua intenção e da sua extensão [MO95] (ver Figura 9.5).

- **Símbolo** – palavras ou imagens que representam uma classe conceitual.
- **Intenção** – a definição de uma classe conceitual.
- **Extensão** – o conjunto de exemplos aos quais a classe conceitual se aplica.

Por exemplo, considere a classe conceitual para o evento de uma transação de compra. Posso escolher denominá-la com o símbolo *Venda*. A intenção de uma *Venda* pode determinar que ela “representa o evento de uma transação de compra e tem uma data e uma hora”. A extensão de *Venda* são todos os exemplos de vendas; em outras palavras, o conjunto de todas as instâncias de vendas no universo.

Definição: modelos de domínio e modelos de dados são a mesma coisa?

Um modelo de domínio não é um **modelo de dados** (que por definição mostra dados persistentes a serem armazenados em algum lugar). Portanto, não exclua uma classe simplesmente porque os requisitos não indicam qualquer necessidade óbvia de lembrar informação sobre ela (um critério comum na modelagem de dados para o projeto de banco de dados relacionais, mas não relevante para a modelagem de domínio) ou porque a classe conceitual não tem atributos. Por exemplo, é válido ter classes

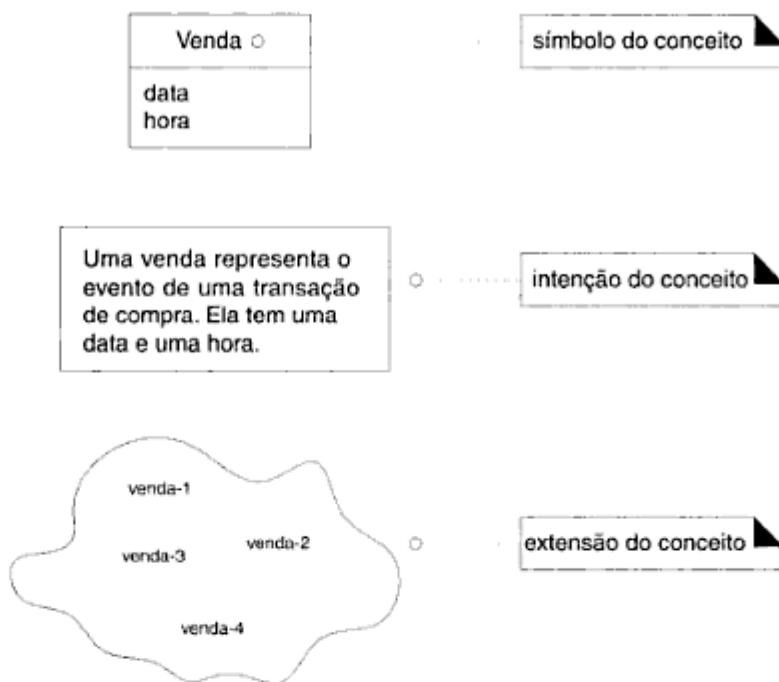


Figura 9.5 Uma classe conceitual tem um símbolo, uma intenção e uma extensão.

conceituais sem atributos ou classes conceituais que têm um papel simplesmente comportamental no domínio, em vez de um papel de informação.

9.3 Motivação: por que criar um modelo de domínio?

Vou compartilhar uma estória que experienciei várias vezes na consultoria e supervisão OO. No início da década de 1990, eu estava trabalhando com um grupo desenvolvendo um sistema de negócios de serviço funerário em Smalltalk, em Vancouver (você deveria ver o modelo de domínio!). No entanto, eu não sabia praticamente nada sobre esse negócio, assim, uma razão para criar um modelo de domínio era começar a entender seus principais conceitos e vocabulário.

*Camada de domínio
(págs. 227-228)*

Também queríamos criar uma **camada de domínio** de objetos Smalltalk representando objetos e lógica do negócio. Assim, gastamos talvez uma hora rascunhando um modelo de domínio *a la* UML (na verdade *a la* OMT, cuja notação inspirou a UML), não nos preocupando com software, mas simplesmente identificando os termos-chave. Posteriormente, esses termos que rascunhamos no modelo de domínio, como, *Serviço* (por exemplo, flores na sala do funeral ou tocar a música fúnebre "You Can't Always Get What You Want"), foram também usados como nomes de classes de software em nossa camada de domínio implementada em Smalltalk.

Essa similaridade na denominação entre o modelo de domínio e a camada de domínio (um "serviço" real e um *Serviço* em Smalltalk) favoreceu um hiato menor entre a representação de software e o nosso modelo mental do domínio.

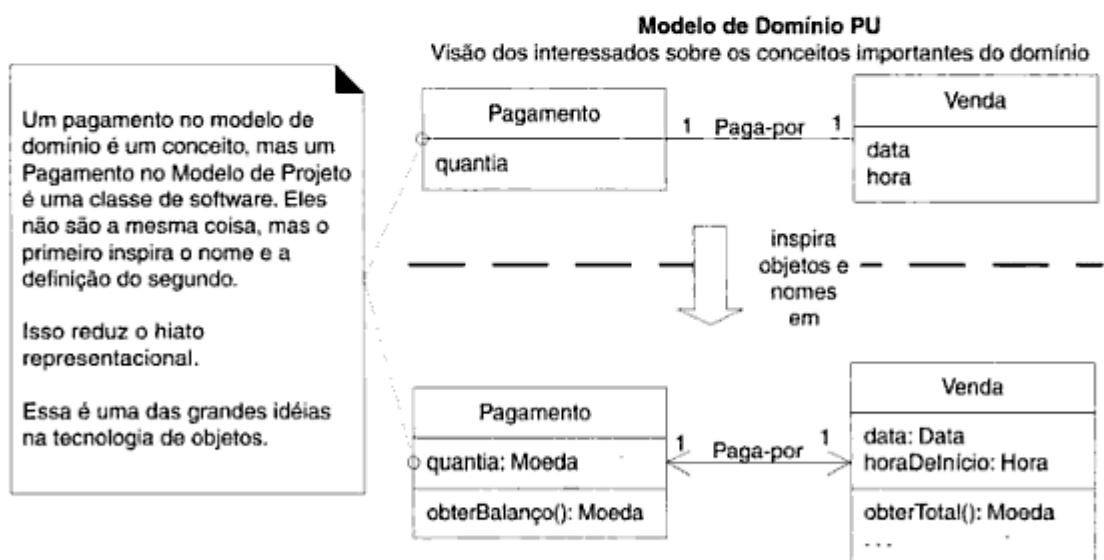
Motivação: diminuição do hiato representacional com modelagem OO

Aqui está uma idéia-chave em OO: use nomes de classes de software na camada de domínio inspirada nos nomes do modelo de domínio, com os objetos possuindo as informações e responsabilidades familiares ao domínio. A Figura 9.6 ilustra a idéia. Isso apóia uma **diminuição do hiato representacional** entre os nossos modelos mental e de software. E isso não é apenas uma conveniência filosófica – tem um impacto prático de tempo e dinheiro, por exemplo, segue-se um código fonte de um programa de pagamento escrito em 1953:

10000101010001111010101010001010101010101111010101 ...

Como pessoal de ciência da computação, sabemos que ele roda, mas o hiato entre essa representação de software e nosso modelo mental do domínio de pagamento é enorme; isso afeta profundamente a compreensão (e modificação) do software. A modelagem OO pode diminuir esse hiato.

Sem dúvida, a tecnologia de objetos também é valiosa, porque pode apoiar o projeto de sistemas elegantes, fracamente acoplados, que facilmente aumentam de escala e se estendem, como vai ser tratado no restante do livro. Um hiato representacional menor é útil, mas discutivelmente secundário diante da vantagem que os objetos têm em apoiar a facilidade de modificação, extensão, gestão e ocultamento de complexidade.



O desenvolvedor orientado a objeto inspirou-se com o domínio do mundo real ao criar classes de software.

Portanto, o hiato representacional entre como os interessados concebem o domínio e sua representação no software foi reduzida.

Figura 9.6 Menor hiato representacional com modelagem OO.

9.4 Diretriz: como criar um modelo de domínio?

Limitado pelos requisitos em projeto na iteração corrente:

1. Encontre as classes conceituais (ver uma diretriz a seguir).
2. Faça seu desenho como classes em um diagrama de classes UML.
3. Acrescente associações e atributos. Ver págs. 174-175 e pág. 183.

9.5 Diretriz: como encontrar classes conceituais?

Considerando que um modelo de domínio mostra classes conceituais, a questão central é: como encontrá-las?

Quais são as três estratégias para encontrar classes conceituais?

1. Reusar ou modificar os modelos existentes. Essa é a primeira, a melhor e, geralmente, a abordagem mais fácil, e é por onde vou começar, se puder. Existem modelos de domínio e modelos de dados já publicados que foram bem concebidos (e que podem ser modificados para modelos de domínio) para vários domínios comuns, como controle de estoque, financeiro, de saúde, etc. Exemplos de livros que recomendaria: *Analysis Patterns*, de Martin Fowler, *Data Model Patterns*, de David Hay, e *Data Model Resource Book* (volumes 1 e 2), de Len Silverston.
2. Usar uma lista de categorias.
3. Identificar substantivos ou frases nominais.⁴

Reusar modelos existentes é excelente, mas fora do nosso escopo. O segundo método, usar uma lista de categorias, também é útil.

Método 2: usar uma lista de categorias

Podemos começar a criação de um modelo de domínio fazendo uma lista das candidatas a classes conceituais. A Tabela 9.1 contém muitas categorias comuns que geralmente valem a pena considerar, com ênfase nas necessidades do sistema de informação de negócio. As diretrizes também sugerem algumas prioridades na análise (das candidatas). Exemplos são retirados dos domínios de: 1) PDV, 2) Banco Imobiliário e 3) reservas de passagens aéreas.

Método 3: como encontrar classes conceituais pela identificação de frases nominais?

Uma outra técnica útil (por sua simplicidade), proposta em [Abbot83], é a **análise lingüística**: identificar os substantivos e as frases nominais nas descrições textuais de um domínio de problema e considerá-los candidatos a classes conceituais ou atributos.⁵

⁴ N. de R.T.: Frases nominais são frases que podem, em uma sentença, ocupar o lugar de um substantivo.

⁵ A análise lingüística tem se tornado mais sofisticada; ela também é chamada de **modelagem de linguagem natural**. Ver, por exemplo, [Moreno97].

Tabela 9.1 Lista de categorias de classes conceituais

Categoria de Classe Conceitual	Exemplos
transações de negócio <i>Diretriz:</i> são cruciais (envolvem dinheiro), assim, comece com as transações.	<i>Venda, Pagamento, Reserva</i>
transações de linhas de item <i>Diretriz:</i> as transações freqüentemente vêm com as linhas de itens relacionadas, assim, considere-as a seguir.	<i>LinhaDeItemDeVenda</i>
produtos ou serviços relacionados com uma transação ou com uma linha de item de transação <i>Diretriz:</i> transações são alguma coisa (um produto ou serviço). Considere-as a seguir.	<i>Item Vôo, Lugar, Refeição</i>
onde a transação é registrada? <i>Diretriz:</i> importante.	<i>Registradora, LivroDiário⁴ ManifestoDeVôo</i>
papéis de pessoas ou organizações relacionadas à transação; atores no caso de uso <i>Diretriz:</i> geralmente precisamos saber sobre as partes envolvidas em uma transação.	<i>Caixa, Cliente, Loja JogadorDeBancoImobiliário Passageiro, LinhaAérea</i>
local da transação; local do serviço	<i>Loja Aeroporto, Avião, Lugar</i>
eventos notáveis, freqüentemente com uma hora ou local que precisamos lembrar	<i>Venda, Pagamento BancoImobiliário Vôo</i>
objetos físicos <i>Diretriz:</i> especialmente importante quando se cria software de controle de dispositivos ou simulações.	<i>Item, Registro Tabuleiro, Peça, Dado Avião</i>
descrição de coisas <i>Diretriz:</i> ver págs. 172-173 para discussão	<i>DescriçãoDoProduto DescriçãoDeVôo</i>
catálogos <i>Diretriz:</i> descrições estão freqüentemente em um catálogo.	<i>CatálogoDeProdutos CatálogoDeVôos</i>

⁴ N. de R.T.: O Livro Diário constitui o registro básico de toda escrituração contábil, sendo que sua utilização por pessoas jurídicas é exigida por lei no Brasil.

Categoria de Classe Conceitual	Exemplos
contêiner de coisas (físico ou informacional)	<i>Loja, Prateleira Tabuleiro Avião</i>
coisas em um contêiner	<i>Item Casa (em um Tabuleiro) Passageiro</i>
outros sistemas colaboradores	<i>SistemaDeAutorizaçãoDeCrédito ControleDeTráfegoAéreo</i>
registros de finanças, trabalho, contrato, assuntos legais	<i>Recibo, LivroDiário RegistroDeManutenção</i>
instrumentos financeiros	<i>DinheiroEmEspécie, Cheque, LinhaDeCrédito TiqueteDeCrédito</i>
horários, manuais, documentos consultados regularmente para realizar trabalho	<i>ListaDiáriaDeModificaçãoDePreço CronogramaDeReparo</i>

Diretriz

Deve-se tomar cuidado ao aplicar esse método; não é possível um mapeamento mecânico de substantivos para classes, e palavras em uma linguagem natural são ambíguas.

Não obstante, a análise lingüística é uma outra fonte de inspiração. Os casos de uso no formato completo fornecem excelentes descrições a serem usadas como fontes para esse tipo de análise. Por exemplo, o cenário atual do caso de uso *Processar Venda* pode ser usado.

Cenário de Sucesso Principal (ou Fluxo Básico):

1. Cliente chega ao ponto de pagamento do PDV com as mercadorias e/ou serviços para adquirir.
 2. Caixa inicia uma nova venda.
 3. Caixa digita identificador do item.
 4. Sistema registra linha de item de venda e apresenta uma descrição do item, seu preço e total parcial da venda. Preço é calculado de acordo com um conjunto de regras de preços.
- Caixa repete os passos 3 e 4 até que indique ter terminado.
5. Sistema apresenta o total com os impostos calculados.
 6. Caixa diz ao Cliente o total e solicita o pagamento.
 7. Cliente paga e Sistema trata o pagamento.

8. Sistema registra a **venda** completada e envia as informações de venda e pagamento para Sistema externo de **Contabilidade** (para contabilização e **comissões**) e Sistema de **Estoque** (para atualizar estoque).
9. Sistema apresenta **recibo**.
10. Cliente sai com recibo e mercadorias (se for o caso)

Extensões (ou Fluxos Alternativos):

...

7a. Pagamento com dinheiro:

1. Caixa digita a quantia de **dinheiro fornecida**.
2. Sistema apresenta o **valor do troco** e libera a **gaveta de dinheiro**.
3. Caixa deposita dinheiro fornecido e entrega o troco para Cliente.
4. Sistema registra o pagamento em dinheiro.

O modelo de domínio é uma visualização de conceitos e de vocabulário do domínio dignos de nota. Onde eles se encontram? Alguns nos casos de uso. Outros em outros documentos ou na cabeça dos especialistas. Em qualquer evento, os casos de uso são uma fonte rica a ser explorada para identificação de frases nominais.

Algumas dessas frases nominais são candidatas a classes conceituais, outras podem se referir a classes conceituais que são ignoradas nessa iteração (por exemplo “Contabilidade” e “comissões”) e outras ainda podem ser simplesmente atributos de classes conceituais. Ver na pág. 185 sugestão sobre como distinguir os dois.

Um ponto fraco dessa abordagem é a imprecisão da linguagem natural; diferentes frases nominais podem representar a mesma classe conceitual ou atributo, entre outras ambigüidades. Apesar disso, ela é recomendada em combinação com a técnica *Lista de Categorias de Classes Conceituais*.

9.6 Exemplo: encontrar e desenhar as classes conceituais

Estudo de caso: domínio PDV

Requisitos da iteração 1 (pág. 150)

A partir da lista de categorias e da análise de frases nominais, é gerada uma lista de candidatos a classes conceituais para o domínio. Como esse é um sistema de informação de negócio, enfocarei primeiro as diretrizes da lista de categorias, que enfatiza as transações de negócio e seus relacionamentos com outras coisas. A lista está restrita aos requisitos e às simplificações atuais que estão sendo considerados para a iteração 1, o cenário básico em que são consideradas apenas as vendas em dinheiro de *Processar Vendas*.

<i>Venda</i>	<i>Caixa</i>
<i>PagamentoEmDinheiro</i>	<i>Cliente</i>
<i>LinhaDeItemDeVenda</i>	<i>Loja</i>

<i>Item</i>	<i>DescriçãoDoProduto</i>
<i>Registradora</i>	<i>CatálogoDeProdutos</i>
<i>LivroDiário</i>	

Não existe algo que possa ser chamado de lista “correta”. O que se tem é uma coleção um tanto arbitrária de abstrações e vocabulário do domínio, que os modeladores consideram dignos de nota. De qualquer forma, seguindo as estratégias de identificação, listas similares serão produzidas por diferentes modeladores.

Na prática, não crio primeiro uma lista textual, mas desenho imediatamente um diagrama de classes UML das classes conceituais à medida que elas são descobertas. Ver Figura 9.7.

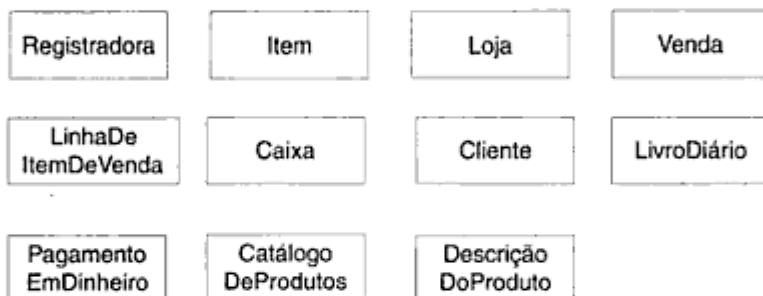


Figura 9.7 Modelo inicial do domínio PDV.

A adição de associações e atributos é abordada nas seções seguintes.

Estudo de caso: domínio Banco Imobiliário

Requisitos da iteração 1 (pág. 150)

A partir da Lista de Categorias e análise de frases nominais, gero uma lista de candidatos a classes conceituais para o cenário simplificado da iteração 1 de *Jogar Banco Imobiliário* (ver Figura 9.8). Como isso é uma simulação, enfatizo os objetos físicos, tangíveis do domínio, que sejam dignos de nota.

9.7 Diretriz: modelagem ágil – esboço de um diagrama de classes

Observe o estilo de rascunho no diagrama de classes UML da Figura 9.8 – deixando abertos os lados direito e inferior das caixas das classes. Isso torna mais fácil a expansão das classes quando descobrirmos novos elementos. Embora tenha agrupado as caixas das classes para compactá-las nesse diagrama do livro, em um quadro branco eu as espalharia mais.

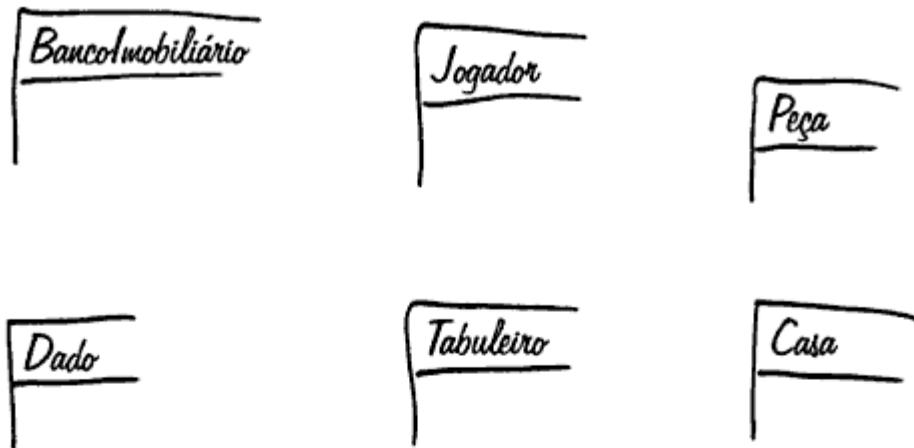


Figura 9.8 Modelo inicial do domínio Banco Imobiliário.

9.8 Diretriz: modelagem ágil – manter o modelo em uma ferramenta?

É normal esquecer classes conceituais significativas durante a modelagem de domínio inicial e descobri-las mais tarde, durante o esboço de projeto ou a programação. Se você estiver usando a abordagem de modelagem ágil, o objetivo de criar um modelo de domínio é para rapidamente entender e comunicar um esboço de aproximação dos conceitos-chave. Perfeição não é o objetivo e modelos ágeis geralmente são descartados rapidamente após a criação (embora se você tiver usado um quadro branco recomendando tirar um instantâneo digital). Sob esse ponto de vista, não há motivação para manter ou atualizar o modelo. Mas isso não significa que seja errado fazer isso.

Se alguém desejar que o modelo seja mantido e atualizado com novas descobertas, essa é uma boa razão para redesenhar o rascunho do quadro branco em uma ferramenta CASE UML ou fazer o desenho originalmente com uma ferramenta e um projetor acoplado ao computador (para que os outros vejam o diagrama facilmente). Mas pergunte a si mesmo: quem irá usar o modelo atualizado e por quê? Se não existir um motivo prático, não se preocupe. Frequentemente, a *camada de domínio* evolutiva do software indica a maioria dos termos dignos de nota, e um modelo de domínio da análise OO com vida longa não agrega valor.

9.9 Diretriz: objetos-relatório – incluir o “Recibo” no modelo?

Recibo é um termo digno de nota no domínio PDV. Mas talvez ele seja apenas um *relatório* de uma venda e seu pagamento e, assim, duplique informação. Ele deveria ser mostrado no modelo de domínio?

Temos alguns fatores a considerar:

- Em geral, mostrar um relatório sobre outra informação em um modelo de domínio não é útil porque toda a sua informação é derivada ou duplicada de outras fontes. Essa é uma razão para excluí-lo.

- Por outro lado, ele tem um papel especial em termos das regras de negócio: geralmente confere o direito ao portador do recibo (papel) de devolver os itens comprados. Essa é uma razão para mostrá-lo no modelo.

Como a devolução de itens não está sendo considerada nesta iteração, *Recibo* será excluído. Durante a iteração que lida com o caso de uso *Tratar Devoluções*, sua inclusão seria justificável.

9.10 Diretriz: pense como um cartógrafo; use termos do domínio

A estratégia do cartógrafo aplica-se tanto a mapas quanto a modelos de domínio.

Diretriz

Fazer um modelo de domínio com o mesmo espírito de como um cartógrafo ou construtor de mapas trabalha:

- Usar os nomes existentes no território. Por exemplo, se estiver desenvolvendo um modelo para uma biblioteca, denomine o cliente de “Usuário” ou “Leitor” – termos usados pelos funcionários da biblioteca.
- Excluir características irrelevantes ou fora do escopo. Por exemplo, no modelo de domínio do Banco Imobiliário para a iteração 1, cartões (como o cartão “Saia da Cadeia Livremente”) não são usados, assim, não mostre um Cartão no modelo nessa iteração.
- Não incluir coisas que não estão lá.

O princípio é semelhante à estratégia *Use o Vocabulário de Domínio* [Coad95].

9.11 Diretriz: como modelar o mundo irreal?

Alguns sistemas de software se destinam a domínios que têm pouca analogia com domínios naturais ou de negócios; softwares para telecomunicações são um exemplo. No entanto, ainda é possível criar um modelo de domínio nesses domínios. Isso requer um alto grau de abstração, bem como um afastamento do que nos acostumamos a considerar em projetos não OO mais habituais e ouvir com cuidado os conceitos e vocabulários-chave que os especialistas do domínio usam.

Por exemplo, temos aqui alguns candidatos a classes conceituais relacionados ao domínio de dispositivos de chaveamento em telecomunicações: *Mensagem*, *Conexão*, *Porta*, *Diálogo*, *Rota*, *Protocolo*.

9.12 Diretriz: um engano comum relativo a atributo versus classes

Talvez o engano mais comum na criação de um modelo de domínio seja representar algo como atributo quando ele deveria ser uma classe conceitual. Uma regra prática para ajudar a evitar esse engano é a seguinte:

Diretriz

Se não pensamos em alguma classe conceitual X como um número ou texto no mundo real, X provavelmente é uma classe conceitual, não um atributo.

Por exemplo, *loja* deve ser um atributo de *Venda* ou uma classe conceitual separada *Loja*?



No mundo real, uma loja não é considerada um número ou texto – o termo sugere uma entidade legal, uma organização e algo que ocupa espaço. Portanto, *Loja* deve ser uma classe conceitual.

Como outro exemplo, considere o domínio de reservas de passagens aéreas. *Destino* deveria ser um atributo de *Vôo* ou uma classe conceitual separada *Aeroporto*?



No mundo real, um aeroporto de destino não é considerado um número ou um texto – ele é uma coisa de grande porte que ocupa espaço. Portanto, *Aeroporto* deveria ser um conceito.

9.13 Diretriz: quando modelar com classes ‘descritivas’?

Uma **classe descritiva** contém informação que descreve algo mais. Por exemplo, uma *DescriçãoDoProduto* que registra o preço, figura e texto descritivo de um *Item*. Isso foi inicialmente denominado padrão *Descrição-Item* em [Coad92].

Motivação: por que usar classes ‘descritivas’?

A discussão a seguir pode, a princípio, parecer relacionada a um problema raro, altamente especializado. Entretanto, a necessidade de classes descritivas é comum em muitos modelos de domínio.

Suponha o seguinte:

- Uma instância de *Item* representa um item físico em uma loja; como tal, ela pode até mesmo ter um número de série.
- Um *Item* tem uma descrição, um preço e um idItem, que não estão registrados em qualquer outro lugar.
- Todos os que trabalham na loja sofrem de amnésia.
- Cada vez que um item físico real é vendido, uma instância de software correspondente de *Item* é excluída do “mundo do software”.

Com essas hipóteses, o que acontece no seguinte cenário?

Existe uma forte demanda para um hambúrguer vegetariano novo e muito apreciado – HambúrguerObjeto. A loja vende todo o seu estoque desse hambúrguer. Isso implica que todas as instâncias do *Item* HambúrguerObjeto são excluídas da memória do computador.

Agora, temos aqui um problema: se alguém pergunta, “quanto custam os Hambúrguer-Objeto?”, ninguém sabe responder, porque a memória do seu preço estava ligada às instâncias estocadas, as quais foram sendo excluídas à medida que eram vendidas.

Aqui há alguns problemas relacionados: o modelo, se implementado em software, similar ao do modelo de domínio, tem dados duplicados, é ineficiente em termos de espaço de armazenamento e propenso a erros (por causa da informação replicada), porque a descrição, o preço e o idItem são duplicados para cada instância de *Item* do mesmo produto.

O problema precedente ilustra a necessidade de objetos que sejam *descrições* (algumas vezes chamados de especificações) de outras coisas. Para resolver o problema de *Item*, é necessária uma classe *DescriçãoDoProduto* que armazene informações sobre itens. Uma *DescriçãoDoProduto* não representa um *Item*, mas uma descrição de informações sobre itens. Veja a Figura 9.9.

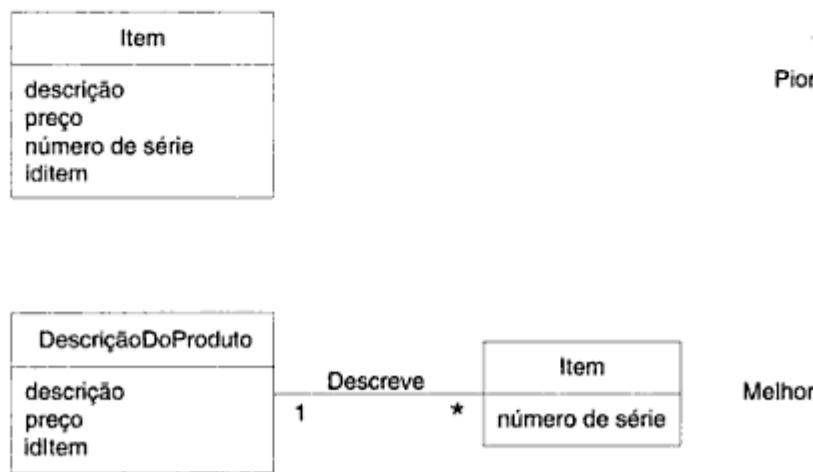


Figura 9.9 Descrições sobre outras coisas. O * significa uma multiplicidade de “muitos”. Indica que uma *DescriçãoDoProduto* pode descrever muitos (*) *Itens*.

Um *Item* em particular deve ter um número de série; ele representa uma instância física. Uma *DescriçãoDoProduto* não deve ter um número de série.

Mudando de uma perspectiva conceitual para a de software, observe que, mesmo se todos os itens registrados no estoque forem vendidos e suas instâncias de *Item* correspondentes em software forem excluídas, a *DescriçãoDoProduto* ainda permanecerá.

A necessidade de classes descritivas é comum nos domínios de vendas, produtos e serviços. Também é comum em manufatura, na qual é necessária uma descrição de uma coisa manufaturada, a qual é distinta da coisa em si.

Diretriz: quando as classes descritivas são úteis?

Diretriz

Acrescente uma classe descritiva (por exemplo, *DescriçãoDoProduto*) quando:

- Houver necessidade de uma descrição sobre um item ou serviço, independentemente da existência de algum exemplar desse item ou serviço.
- A exclusão de instâncias de coisas que elas descrevem (por exemplo, *Item*) resultar em uma perda de informação que precisa ser mantida, mas foi incorretamente associada com a coisa excluída.
- Ela reduzir informação redundante ou duplicada.

Exemplo: descrições em um domínio de linhas aéreas

Como outro exemplo, considere uma companhia de linhas aéreas que tenha um acidente fatal com um dos seus aviões. Suponha que todos os vôos sejam cancelados por seis meses, enquanto se aguarda o término da investigação. Imagine também que, quando os vôos são cancelados, os seus correspondentes objetos de software *Vôo* sejam excluídos da memória do computador. Portanto, após o acidente, todos os objetos de software *Vôo* são excluídos.

Se o único registro que mostra para qual aeroporto um vôo se destina estiver nas instâncias de software *Vôo*, as quais representam vôos específicos para uma data e hora específicas, então não existirá mais um registro sobre quais rotas de vôo a linha aérea tem.

O problema pode ser resolvido tanto de uma perspectiva puramente conceitual em um modelo de domínio quanto de uma perspectiva de software nos projetos de software, com uma *DescriçãoDeVôo* que descreva um vôo e sua rota, mesmo quando um determinado vôo não estiver programado (ver Figura 9.10).

Observe que o exemplo precedente é sobre um serviço (um vôo), em vez de um bem (como um hambúrguer vegetariano). As descrições de serviços ou planos de serviços geralmente são necessárias.

Como outro exemplo, uma companhia de telefonia móvel vende pacotes como “bronze”, “ouro”, etc. É necessário ter o conceito de uma descrição do pacote (um tipo de plano de serviços que descreve tarifas por minuto, o conteúdo de Internet sem

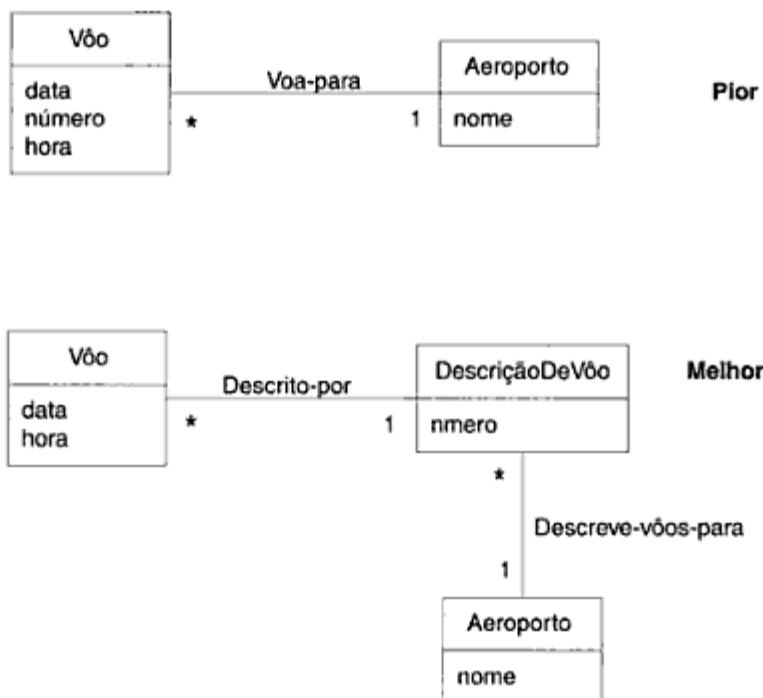


Figura 9.10 Descrições sobre outras coisas.

fio, o custo, etc.), separado do conceito de um pacote efetivamente vendido (como por exemplo “pacote de ouro vendido para Craig Larman, em 1º de janeiro de 2047, a US\$ 55 por mês”). O departamento de Marketing necessita definir e registrar esse plano de serviço ou *DescriçãoDePacoteDeComunicaçõesMóveis* antes que algum seja vendido.

9.14 Associações

É útil encontrar e mostrar as associações necessárias para satisfazer os requisitos de informação para os cenários em desenvolvimento, as quais auxiliam na compreensão do modelo de domínio.

Uma associação é um relacionamento entre classes (mais especificamente, entre instâncias dessas classes) que indica alguma conexão significativa e de interesse (ver Figura 9.11).

Na UML, as associações são definidas como “o relacionamento semântico entre dois ou mais classificadores, o que envolve conexões entre suas instâncias”.

Diretriz: quando mostrar uma associação?

As associações dignas de nota costumam implicar conhecimento de um relacionamento que necessita ser preservado por algum tempo – que pode ser de milisegundos ou de anos, dependendo do contexto. *Em outras palavras, entre quais objetos necessitamos ter alguma memória de um relacionamento?*

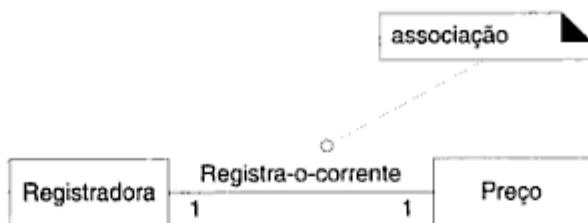


Figura 9.11 Associações.

Por exemplo, precisamos *lembra*r quais instâncias de *LinhaDeItemDeVenda* são associadas a uma instância de *Venda*? Sim, pois caso contrário não seria possível reconstruir a venda, imprimir um recibo ou calcular um total da venda.

E precisamos lembrar as *Vendas* completadas em um *LivroDiário* para fins legais e de contabilidade.

Como o modelo de domínio é uma perspectiva conceitual, essas afirmações sobre a necessidade de lembrar referem-se a uma necessidade em uma situação do mundo real, não a uma necessidade de software, embora durante a implementação muitas das mesmas necessidades surjam.

No domínio do jogo Monopólio, necessitamos lembrar em qual *Casa* uma *Peça* (ou *Jogador*) está – o jogo não funciona se isso não for lembrado. Do mesmo modo, precisamos lembrar qual *Peça* é propriedade de um *Jogador* específico. Precisamos lembrar quais *Casas* são partes de um *Tabuleiro* em particular.

Mas, por outro lado, não há necessidade de lembrar que o total de *Dado* (ou o plural, “dados”) indica a *Casa* para qual que se deve mover. É verdade, mas não é necessário ter uma memória atualizada daquele fato, depois que um movimento for feito. Do mesmo modo, um *Caixa* pode procurar uma *DescriçãoDoProduto*, mas não há necessidade de lembrar o fato de que um *Caixa* em particular procurou uma *DescriçãoDoProduto* específica.

Diretriz

Considere a inclusão das seguintes associações em um modelo de domínio:

- Associações para as quais o conhecimento do relacionamento precisa ser preservado por algum tempo (associações que “precisam ser conhecidas”).
- Associações derivadas da Lista de Associações Comuns.

Diretriz: por que devemos evitar adicionar muitas associações?

Precisamos evitar adicionar associações em excesso no modelo de domínio. Retornando às informações de nossos estudos de matemática discreta, você deve se lembrar de que, em um grafo com n nós, pode haver $(n \cdot (n-1))/2$ associações com outros nós – um número potencialmente muito grande. Um modelo de domínio com 20

classes poderia ter 190 linhas de associações! Muitas linhas no diagrama o obscurecem com “ruído visual”. Portanto, deve-se ter parcimônia quanto à adição de linhas de associação. Use as diretrizes criteriosas sugeridas neste capítulo e enfoque as associações que “precisam ser lembradas”.

Perspectivas: as associações serão implementadas em software?

Durante a modelagem de domínio, uma associação *não* é uma afirmação sobre fluxos de dados, nem um relacionamento via chave estrangeira em banco de dados, nem instância de variáveis ou conexão de objetos em uma solução de software; ela é uma afirmação de que um relacionamento é significativo em uma perspectiva puramente conceitual – no domínio real.

Dito isso, muitos desses relacionamentos *serão* implementados em software como caminhos de navegação e visibilidade (tanto no Modelo de Projeto quanto no Modelo de Dados). Mas o modelo de domínio *não* é um modelo de dados; associações são adicionadas para evidenciar nosso entendimento bruto dos relacionamentos dignos de nota, não para documentar objetos ou estruturas de dados.

Aplicação da UML: notação para associação

Uma associação é representada como uma linha entre classes, com o nome da associação começando com letra maiúscula. Ver Figura 9.12.

Os extremos de uma associação podem conter uma expressão de multiplicidade, indicando o relacionamento numérico entre as instâncias das classes.

A associação é inherentemente bidirecional, o que significa que o percurso lógico é possível a partir de uma instância de qualquer classe, em ambos os lados. Esse percurso é puramente abstrato; ele *não* é uma afirmação sobre conexões entre entidades de software.

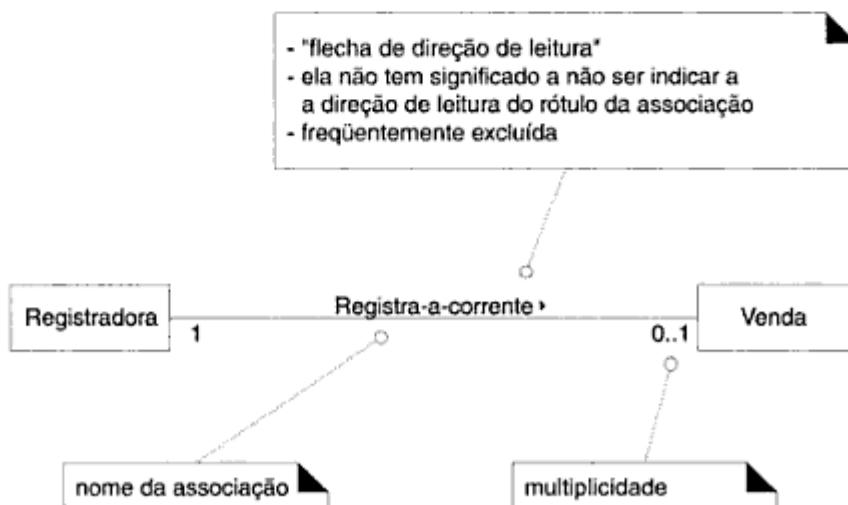


Figura 9.12 A notação UML para associações.

Uma seta opcional de “direção de leitura” indica a direção de leitura do nome da associação; ela não indica a direção de visibilidade ou de navegação. Na ausência de uma seta, convencionou-se que a associação deve ser lida da esquerda para a direita ou de cima para baixo, embora a UML não faça disso uma regra (ver Figura 9.12).

Cuidado

A flecha de direção de leitura não tem significado em termos de modelo, é apenas um auxílio para o leitor do diagrama.

Diretriz: como dar nome a uma associação em UML?

Diretriz

Nomeie uma associação com base no formato *NomeDaClasse-FraseComVerbo-NomeDaClasse*, no qual a frase com o verbo cria uma seqüência legível e significativa.

Nomes de associação simples como “Tem” ou “Usa” em geral são pobres, pois raramente melhoram o nosso entendimento do domínio.

Por exemplo,

- *Venda Paga-por PagamentoEmDinheiro*
 - Exemplo ruim (não melhora o significado): *Venda Usa PagamentoEmDinheiro*
- *Jogador Está-na Casa*
 - Exemplo ruim (não melhora o significado): *Jogador Tem Casa*

Os nomes das associações devem começar com uma letra maiúscula, uma vez que uma associação representa um classificador de ligações entre instâncias e, na UML, os classificadores devem começar com letra maiúscula. Dois formatos comuns e igualmente legais para compor um nome de associação são:

- *Registra-a-corrente*
- *RegistraACorrente*

Aplicação de UML: papéis

Cada extremo de uma associação é chamado de **papel**. Os papéis podem ter opcionalmente:

- expressão de multiplicidade
- nome
- navegabilidade

A multiplicidade é examinada a seguir.

Aplicação de UML: multiplicidade

A **multiplicidade** define quantas instâncias de uma classe *A* podem estar associadas a uma instância de uma classe *B* (ver Figura 9.13).

Por exemplo, uma única instância de uma *Loja* pode ser associada a “muitas” (zero ou mais, indicado pelo *) instâncias de *Item*.

Alguns exemplos de expressões de multiplicidade são mostrados na Figura 9.14.

O valor de multiplicidade informa quantas instâncias podem ser associadas corretamente com outra, em um determinado momento, em vez de ao longo do tempo.[‡] Por exemplo, é possível que um carro usado possa ser vendido de volta inúmeras vezes para um revendedor de carros usados ao longo do tempo. Mas, em um certo momento, o carro é *Estocado-por* somente um revendedor. O carro não é *Estocado-por* muitos revendedores naquele momento. Da mesma forma, em países com leis monogámi-

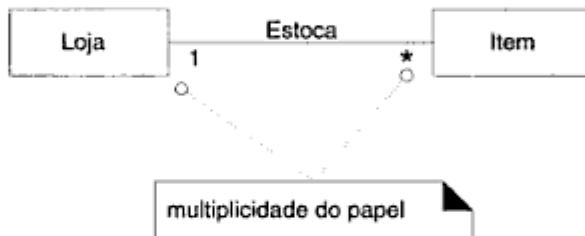


Figura 9.13 Multiplicidade em uma associação.



Figura 9.14 Valores de multiplicidade.

[‡] N. de R.T.: Por exemplo, uma multiplicidade de valor 1 significa que uma instância está associada, em um dado momento, a apenas uma instância de outra classe conceitual. Naturalmente, ao longo da vida da instância, ela poderá ser associada a muitas outras, porém somente a uma de cada vez.

cas, uma pessoa pode estar *Casada-com* somente uma outra pessoa em um determinado momento, ainda que, ao longo do tempo, ela possa ter estado casada com mais pessoas.

O valor da multiplicidade depende do nosso interesse como modeladores e desenvolvedores de software, visto que ele informa uma restrição do domínio que será (ou poderia ser) refletida no software. Veja na Figura 9.15 um exemplo e explicação.

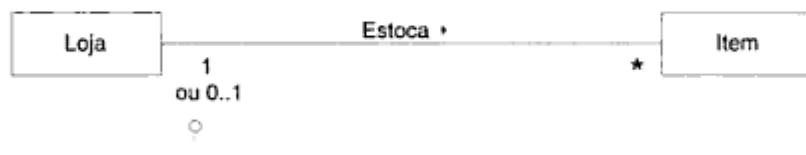
Rumbaugh dá um outro exemplo de *Pessoa* e *Empresa* na associação *Trabalha-para* [Rumbaugh91]. Indicar se uma instância de *Pessoa* trabalha para uma ou mais instâncias de *Empresa* depende do contexto do modelo; a Receita Federal está interessada em *muitos*; um sindicato, provavelmente em *um só*. A escolha depende, na prática, de por que estamos construindo o software.

Aplicação de UML: associações múltiplas entre duas classes

Duas classes podem ter múltiplas associações entre elas em um diagrama de classes UML, e isso não é raro. Não há exemplo importante disso nos estudos de caso PDV e Banco Imobiliário, mas um exemplo no domínio de linhas aéreas é o caso dos relacionamentos entre um *Vôo* (ou talvez, mais precisamente, uma *PernaDeVôo*) e um *Aeroporto* (ver Figura 9.16); As associações voando-para e voando-de são relacionamentos diferentes, que devem ser mostrados separadamente.

Diretriz: como encontrar associações com uma lista de associações comuns?

Comece a adição de associações usando a lista da Tabela 9.2. Ela contém categorias comuns que valem a pena considerar, especialmente para sistemas de informação comerciais. Os exemplos são tirados dos domínios 1) PDV, 2) Banco Imobiliário e 3) Reserva de Passagens Aéreas



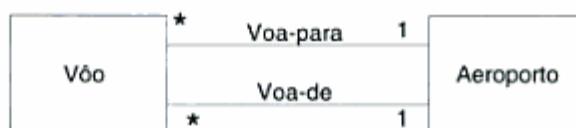
A multiplicidade deveria ser "1" ou "0..1"?

A resposta depende do nosso interesse em usar o modelo. Tipicamente e praticamente, a multiplicidade informa uma restrição do domínio que deve ser considerada para verificação pelo software – e esse relacionamento foi implementado ou refletido nos objetos de software ou em um banco de dados. Por exemplo, um determinado item pode estar esgotado ou ter sido descartado e, por isso, não existir mais estoque na loja. Sob esse ponto de vista, "0..1" faz sentido, mas ...

Devemos nos preocupar com esse ponto de vista? Se esse relacionamento fosse implementado no software, provavelmente desejariamos assegurar que uma instância de *Item* no software estivesse sempre relacionada com uma instância específica de *Loja*; caso contrário, teríamos um erro ou dados corrompidos.

Esse modelo parcial de domínio não representa objetos, mas as multiplicidades registram restrições cujo valor prático normalmente está relacionado com nosso interesse em construir software ou bancos de dados (que refletem nosso domínio no mundo real) com verificações de validade. Sob esse ponto de vista, o valor desejado pode ser "1".

Figura 9.15 A multiplicidade depende do contexto.

**Figura 9.16** Associações múltiplas.**Tabela 9.2** Lista de associações comuns

Categoria	Exemplos
A é uma transação relacionada à outra transação B	PagamentoEmDinheiro – Venda Cancelamento – Reserva
A é uma linha de item de uma transação B	LinhaDelItemDeVenda – Venda
A é um produto ou serviço para uma transação (ou linha de item) B	Item – LinhaDelItemDeVenda (ou Venda) Vôo – Reserva
A é um papel relacionado a uma transação B	Cliente – Pagamento Passageiro – Passagem
A é uma parte física ou lógica de B	Gaveta – Registradora Casa – Tabuleiro Assento – Avião
A está física ou logicamente contido em B	Registradora – Loja, Item – Prateleira Casa – Tabuleiro Passageiro – Avião
A é uma descrição para B	DescriçãoDoProduto – Item DescriçãoDeVôo – Vôo
A é conhecido/ registrado/ relatado/ capturado em B	Venda- Registradora Peça – Casa Reserva – ManifestoDeVôo
A é um membro de B	Caixa – Loja Jogador – BancoImobiliário Piloto – LinhaAérea
A é uma subunidade organizacional de B	Departamento – Loja Manutenção – LinhaAérea
A usa ou gerencia ou possui B	Caixa – Registradora Jogador – Parte Piloto – Avião
A é próximo a B	LinhaDelItemDeVenda – LinhaDelItemDeVenda Casa – Casa Cidade – Cidade

9.15 Exemplo: associações nos modelos de domínio

Estudo de caso: PDV ProxGer

O modelo de domínio da Figura 9.17 mostra um conjunto de classes conceituais e associações que são candidatas ao nosso modelo de domínio PDV. As associações são principalmente derivadas a partir do critério “precisa ser lembrada” para os requisitos dessa iteração e da Lista de Associações Comuns. Lendo a lista e mapeando os exemplos para o diagrama, podemos explicar as escolhas. Por exemplo:

- **Transações relacionadas a outra transação** – *Venda Paga-Por PagamentoEmDinheiro*.
- **Itens de linha de uma transação** – *Venda Contém ItemDeLinhaDeVenda*.
- **Produto de uma transação (ou item de linha)** – *LinhaDeItemDeVenda RegistraVenda-Doltem*.

Estudo de caso: Banco Imobiliário

Veja a Figura 9.18. Outra vez, as associações são principalmente derivadas a partir do critério “precisa ser lembrada” para os requisitos dessa iteração e da Lista de Associações. Por exemplo:

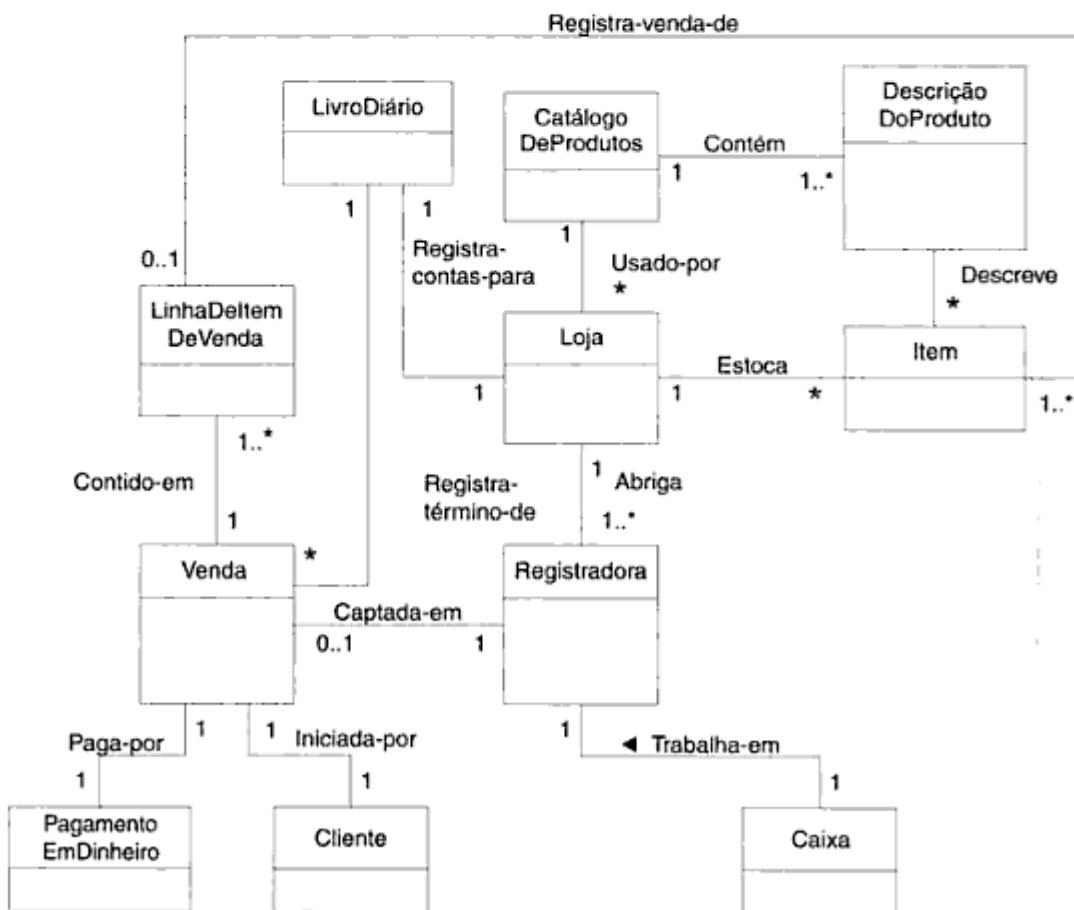


Figura 9.17 Um modelo parcial de domínio PDV ProxGer.

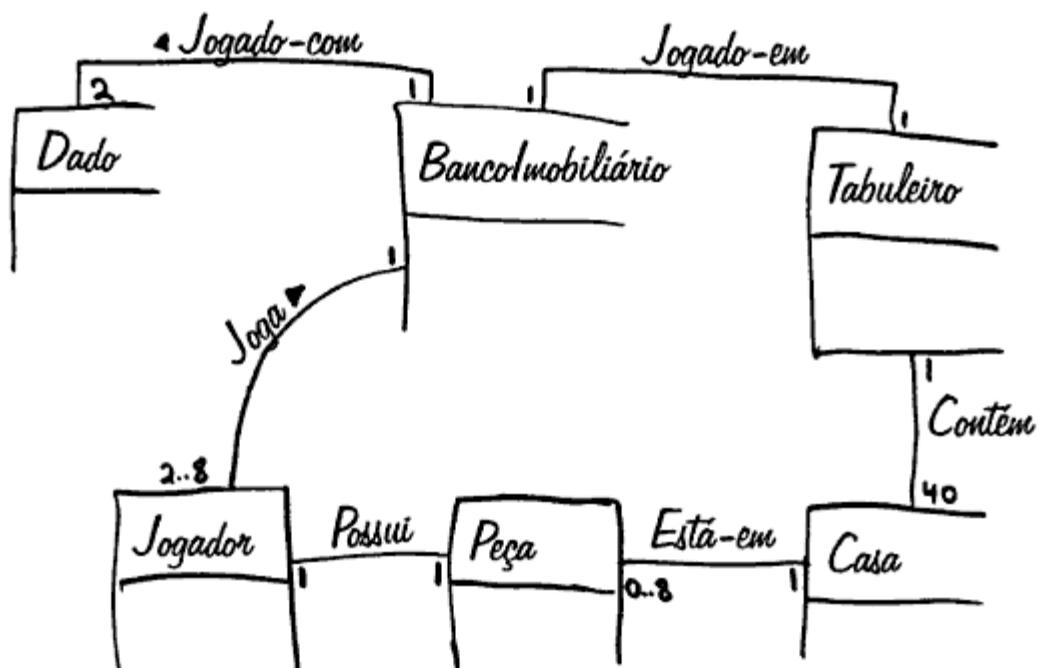


Figura 9.18 Modelo parcial de domínio do Banco Imobiliário.

- **A está contido em ou sobre B – Tabuleiro Contém Casa.**
- **A possui B – Jogadores Possuem Peças.**
- **A é conhecido de/sobre B – Peças Estão-na Casa.**
- **A é membro de B – Jogador É-Membro-de (ou Joga) BancaImobiliário.**

9.16 Atributos

É útil identificar os atributos das classes conceituais que são necessários para satisfazer aos requisitos de informação dos cenários em desenvolvimento. Um **atributo** é um valor de dados lógico de um objeto.

Diretriz: quando mostrar atributos?

Inclua atributos para os quais os requisitos (por exemplo, casos de uso) sugerem ou implicam a necessidade de memorizar informações.

Por exemplo, um recibo (que contém as informações de uma venda) no caso de uso *Processar Venda* normalmente inclui uma data e uma hora, o nome e endereço da loja e a ID do caixa, entre outras coisas.

Conseqüentemente,

- *Venda* precisa de um atributo *dataHora*.
- *Loja* precisa de um *nome* e *endereço*.
- *Caixa* precisa de uma *ID*.

Aplicação de UML: notação de atributos

Os atributos são mostrados no segundo compartimento da caixa que contém a classe (ver Figura 9.19). Opcionalmente, pode-se mostrar o seu tipo e outras informações.

Mais Notação

Notação de talhada UML para classe na pág. 266 e também na capa posterior interna do livro

A sintaxe completa de um atributo UML é:

visibilidade nome: tipo multiplicidade = default (cadeia-de caracteres-de-propriedades)

Alguns exemplos comuns são mostrados na Figura 9.20.

Como convenção, a maioria dos modeladores considerarão que os atributos têm visibilidade privada (-), a menos que mostrado de outro modo, assim, não costumo desenhar um símbolo explícito de visibilidade.

{SomenteLeitura} é provavelmente a cadeia de caracteres de propriedades mais comum para atributos.

A multiplicidade pode ser usada para indicar a presença opcional de um valor ou o número de objetos que pode preencher uma (coleção) de atributos. Por exemplo, muitos domínios exigem que um primeiro e último nome sejam conhecidos para uma pessoa, mas que um nome do meio seja opcional. A expressão *nomeDoMeio : [0..1]* indica um valor opcional – valores 0 ou 1 estão presentes.



Figura 9.19 Classes e atributos.

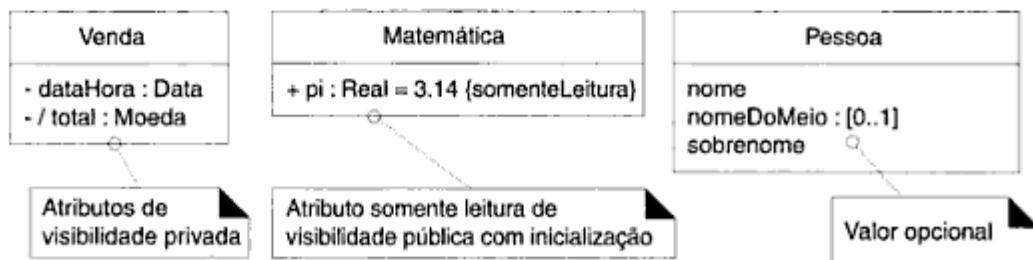


Figura 9.20 Notação UML para Atributo.

Diretriz: onde registrar requisitos de atributos?

Note que, sutilmente, *nomeDoMeio : [0..1]* é um requisito ou regra de domínio embutido no modelo de domínio. Embora esse seja apenas um modelo de domínio de perspectiva conceitual, ele provavelmente implica que a perspectiva do software deve permitir que falte o valor para *nomeDoMeio* na IU, nos objetos e no banco de dados. Alguns modeladores aceitam deixar tais especificações apenas no modelo de domínio, mas isso é propenso a erro e dispersivo, pois as pessoas tendem a não olhar o modelo de domínio em detalhe e nem a procurar orientação sobre requisitos. Sequer têm o costume de manter o modelo de domínio atualizado.

Em vez disso, sugiro colocar todos esses requisitos de atributo no Glossário do PU, que serve como dicionário de dados. Digamos que eu tenha gasto uma hora esboçando um modelo de domínio com um especialista em domínio; depois disso, posso gastar 15 minutos examinando-o e transferindo os requisitos implícitos de atributos para o Glossário.

Outra alternativa é usar uma ferramenta que integra modelos UML com um dicionário de dados; então todos os atributos vão aparecer automaticamente como elementos do dicionário.

Atributos derivados

O atributo *total* em *Venda* pode ser calculado ou derivado a partir da informação de *LinhaDeItemDeVenda*. Quando desejamos comunicar que 1) esse é um atributo importante, mas 2) ele é derivável, usamos a convenção UML: um símbolo / antes do nome do atributo.

Como outro exemplo, um caixa pode receber um grupo de itens iguais (por exemplo, seis pacotes de tofu), introduzir o *idItem* uma vez e depois entrar uma quantidade (por exemplo, seis). Conseqüentemente, uma *LinhaDeItemDeVenda* pode ser associada a mais de uma instância de um item.

A quantidade introduzida pelo caixa pode ser registrada como um atributo de *LinhaDeItemDeVenda* (Figura 9.21). No entanto, a quantidade pode ser calculada a partir do valor de multiplicidade real da associação, de modo que ele pode ser caracterizado como um atributo derivado – que pode ser derivado a partir de outra informação.

Diretriz: quais são os tipos de atributos adequados?

Enfoque em atributos de tipos de dados no modelo de domínio

Informalmente, a maioria dos tipos de atributo são os conhecidos como tipos de dados “primitivos”, tais como números e booleanos. O tipo de um atributo normalmente *não* deveria ser um conceito complexo do domínio, como uma *Venda* ou *Aeroporto*.

Por exemplo, o atributo *registradoraCorrente* na classe *Caixa*, na Figura 9.22, é indesejável porque o seu tipo significa uma *Registradora*, que não é um tipo simples de atributo (como *Número* ou *Cadeia*). A maneira mais vantajosa de dizer que um *Caixa* usa uma *Registradora* é com uma associação, não com um atributo.

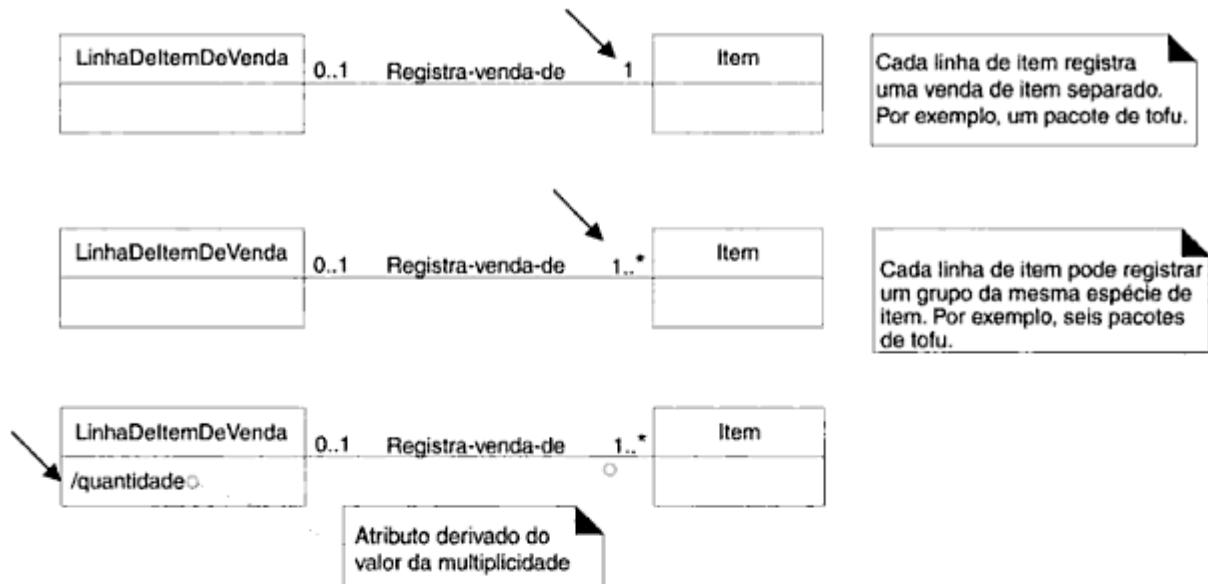


Figura 9.21 Registro da quantidade de itens vendidos em uma linha de item.

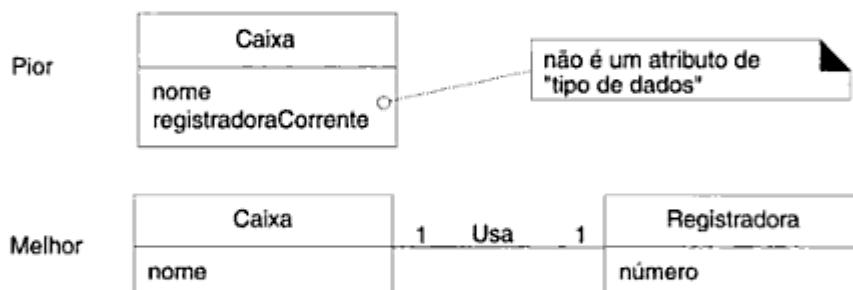


Figura 9.22 Relacione com associações, não com atributos.

Diretriz

De preferência, em um modelo de domínio, os atributos devem ser **tipos de dados**. Tipos muito comuns de dados são: *Booleano*, *Data*, *Número*, *Cadeia (Texto)*, *Hora*.

Outros tipos comuns incluem: *Endereço*, *Cor*, *Figuras Geométricas (Ponto, Retângulo)*, *Número de Telefone*, *Número do CPF*, *Código Universal de Produto (CUP)*, *SKU*, *ZIP ou CEP*, *tipos enumerados*.

Repetindo um exemplo anterior, uma confusão comum é modelar um conceito complexo do domínio como um atributo. Para ilustrar, observe que um aeroporto de destino não é, na realidade, uma cadeia de caracteres; ele é algo complexo, que ocupa muitos quilômetros quadrados de espaço. Portanto, *Vôo* deve estar relacionado com *Aeroporto* por uma associação, não com um atributo, como mostrado na Figura 9.23.

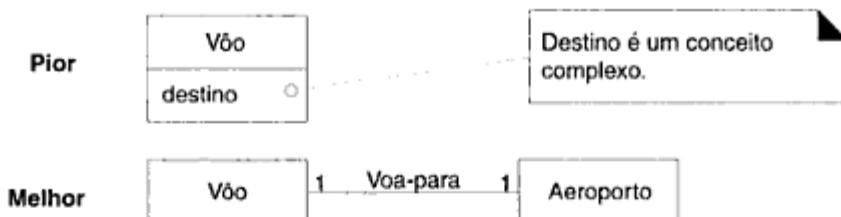


Figura 9.23 Não mostre conceitos complexos como atributos; use associações.

Diretriz

Relacione classes conceituais com uma associação, não com um atributo.

Tipos de dados

Como foi dito, os atributos no modelo de domínio deveriam ser **tipos de dados** (*data types*); informalmente, esses são tipos “primitivos” como número, booleano, caractere, cadeia e enumeração (por exemplo: Tamanho = {pequeno, grande}). Mais precisamente, esse é um termo da UML que significa um conjunto de valores para os quais uma identidade única não é significativa (no contexto de nosso modelo ou sistema) [RJB99]. Dito de outro modo, testes de igualdade *não* são baseados na identidade, mas no valor.⁶ Por exemplo, normalmente não faz sentido distinguir entre:

- Instâncias separadas do *Inteiro* 5.
- Instâncias separadas da *cadeia* “gato”.
- Instância separada da *Data* “13 de Novembro de 1990”.

Por outro lado, *faz* sentido distinguir (pela identidade) entre duas instâncias separadas de uma *Pessoa* cujos nomes são ambos “Jill Smith”, porque as duas instâncias podem representar indivíduos diferentes com o mesmo nome.

Também, valores de tipos de dados muitas vezes são imutáveis. Por exemplo, a instância ‘5’ de *Inteiro* é imutável; a instância “13 de Novembro de 1990” de *Data* é provavelmente imutável. Por outro lado, uma instância de *Pessoa* pode ter seu *Sobrenome* trocado por várias razões. De uma perspectiva de software, existem poucas situações em que alguém compararia os endereços de memória de instâncias de *Inteiro* (identidade) ou *Data*; somente são relevantes as comparações baseadas em valor. Por outro lado, é normal comparar os endereços de memória de instâncias de *Pessoa* e distingui-los, ainda que eles tenham os mesmos valores de atributo, porque a sua identidade única é importante.

Alguns livros de OO e UML também falam de **objetos de valor** (*value objects*), que são muito parecidos com os tipos de dados, mas com pequenas variações. Entretanto, considero essas distinções um tanto confusas e sutis e não as enfatizo.

⁶ Em Java, por exemplo, um teste de valor é feito com o método *equals* e um teste de identidade com o operador `==`.

Perspectivas: como ficam os atributos no código?

A recomendação de que atributos no modelo de domínio sejam principalmente tipos de dados não implica que atributos em C# ou Java devam ser somente de tipos de dados simples, primitivos. O modelo de domínio é uma perspectiva conceitual, não de software. No Modelo de Projeto, atributos podem ser de qualquer tipo.

Diretriz: quando definir novas classes de tipo de dados?

No sistema PDV ProxGer, um atributo *idItem* é necessário; ele é provavelmente um atributo de um *Item* ou *DescriçãoDoProduto*. Casualmente, ele parece justamente um número ou talvez uma cadeia. Por exemplo, *idItem: Integer* ou *idItem: String*.

Mas ele é mais do que isso (identificadores de item tem subpartes), e, de fato, é útil ter uma classe chamada *IdItem* (ou *IdentificadorDeItem*) no modelo de domínio e designar o tipo de atributo como tal. Por exemplo, *idItem: IdentificadorDeItem*.

A Tabela 9.3 fornece diretrizes de quando é útil modelar com tipos de dados.

Ao aplicar essas diretrizes aos atributos do modelo de domínio do PDV, chegamos à seguinte análise:

- O identificador de item é uma abstração de vários esquemas comuns de codificação, incluindo CUP-A, CUP-E, e a família de esquemas EAN. Esses esquemas numéricos de codificação têm subpartes que identificam o fabricante, o produto, o país (no caso do EAN) e dígitos de controle para validação. Portanto, deve haver uma classe para o tipo de dado *IdItem*, porque ela satisfaz a muitas das diretrizes abaixo.

Tabela 9.3 Diretrizes para modelar tipos de dados

Diretriz

Represente o que a princípio poderia ser considerado um número ou uma cadeia como uma nova classe de tipos de dados no modelo de domínio se:

- Ela for composta de seções separadas.
 - número de telefone, nome de pessoa.
- Existirem operações associadas a ela, como análise sintática ou validação.
 - número de CPF.
- Ela tiver outros atributos.
 - um preço promocional pode ter uma data de início (de entrada em vigor) e uma de fim.
- Ela for uma quantidade com uma unidade.
 - a quantia paga tem uma unidade monetária.
- Ela for uma abstração de um ou mais tipos com algumas dessas qualidades.
 - identificador de item no domínio de vendas é uma generalização de tipos, como Código Universal de Produto (CUP) e European Article Number (EAN).

- Os atributos *preço* e *quantia* devem ser uma classe do tipo de dados *Dinheiro*, porque são quantidades em uma unidade monetária.
- O atributo *endereço* deve ser uma classe do tipo de dados *Endereço* porque tem seções separadas.

Aplicação de UML: onde ilustrar essas classes de tipos de dados?

A classe *IdItem* deve ser mostrada como uma classe separada em um modelo de domínio? Isso depende do que você quer enfatizar no diagrama. Como *IdItem* é um **tipo de dado** (a identidade única das instâncias não é usada no teste de igualdade), pode ser mostrado apenas no compartimento de atributos da caixa da classe, como mostrado na Figura 9.24. Por outro lado, se *IdItem* é um novo tipo com seus próprios atributos e associações; mostrá-lo como uma classe conceitual na sua própria caixa pode ser informativo. Não há uma resposta correta; a solução depende de como o modelo de domínio está sendo usado como ferramenta de comunicação e da importância do conceito no domínio.

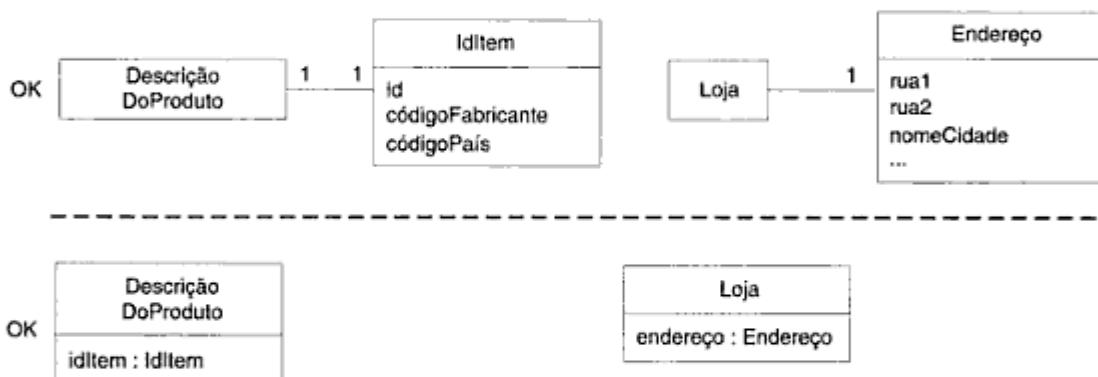


Figura 9.24 Dois modos de indicar uma propriedade de tipo de dados de um objeto.

Diretriz: nenhum atributo representando chave estrangeira

Os atributos *não* devem ser usados para relacionar classes conceituais no modelo de domínio. A violação mais comum desse princípio é acrescentar um tipo de **atributo chave estrangeira**, como normalmente é feito nos projetos de bancos de dados relacionais para associar dois tipos. Por exemplo, na Figura 9.25, o atributo *númeroDaRegistradoraCorrente* na classe *Caixa* é indesejável, visto que sua finalidade é relacionar o *Caixa* a um objeto *Registradora*. A melhor maneira de dizer que um *Caixa* usa uma *Registradora* é por uma associação, não com um atributo chave estrangeira. Lembre: relate tipos com uma associação, não com um atributo.

Existem muitas maneiras de relacionar objetos – as chaves estrangeiras são apenas uma – e postergaremos a forma de como implementar a relação até a fase de projeto para evitar a **infiltração no projeto**.

Diretriz: modelagem de quantidades e unidades

Muitas quantidades numéricas *não* devem ser representadas como números simples. Pense em preço ou peso. Dizer “o preço era 13” ou “o peso era 37”, não diz muito. Reais? Quilogramas?

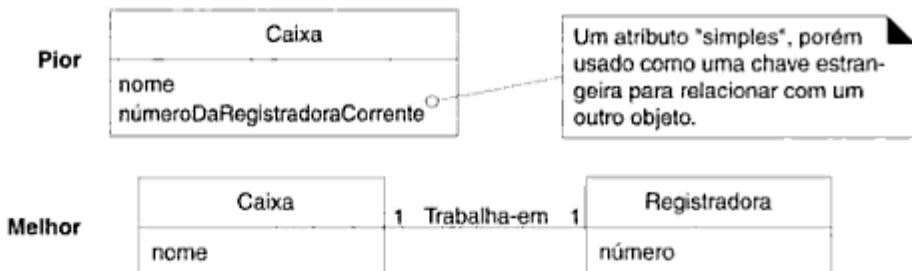


Figura 9.25 Não use atributos como chaves estrangeiras.

Essas são quantidades com unidades associadas e geralmente é necessário conhecer as unidades para que essas possam ser convertidas. O software PDV ProxGer destina-se ao mercado internacional e deve aceitar preços em diversas moedas. O modelo de domínio (e o software) devem modelar as quantidades de forma habilidosa.

Na maioria dos casos, a solução é representar *Quantidade* como uma classe distinta, com uma *Unidade* associada [Fowler96]. Também é comum mostrar especializações de *Quantidade*. *Dinheiro* é um tipo de quantidade cujas unidades são moedas. *Peso* é uma quantidade com unidades como quilos ou libras. Veja a Figura 9.26.

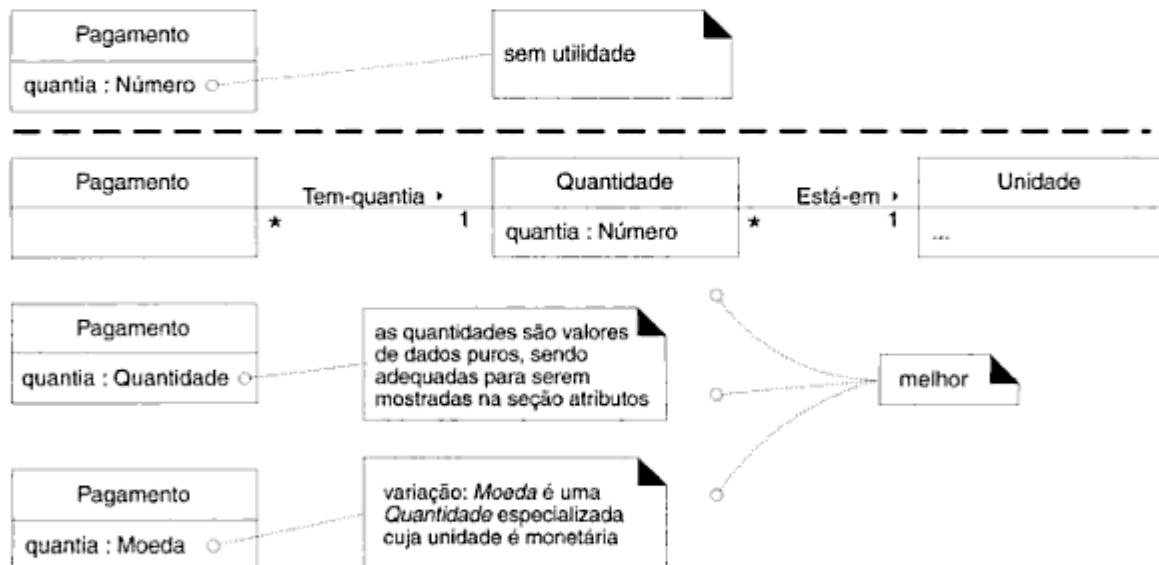


Figura 9.26 Modelagem de quantidades.

9.17 Exemplo: atributos em modelos de domínio

Estudo de caso: PDV ProxGer

Veja a Figura 9.27. Os atributos escolhidos refletem os requisitos de informação para essa iteração – os cenários de *ProcessarVenda* com pagamento em dinheiro dessa iteração. Por exemplo:

<i>PagamentoEmDinheiro</i>	<i>quantiaFornecida</i> – Para determinar se houve um pagamento suficiente e para calcular o troco, uma quantia (também conhecida como “quantia fornecida”) deve ser captada.
<i>DescriçãoDoProduto</i>	<i>descrição</i> – Para mostrar a descrição em uma tela ou em um recibo.
<i>Venda</i>	<i>idItem</i> – Para procurar uma <i>DescriçãoDoProduto</i> . <i>preço</i> – Para calcular o total da venda e mostrar o preço da linha de item.
<i>LinhaDeItemDeVenda</i>	<i>dataHora</i> – Um recibo normalmente mostra a data e o horário da venda, e isso é útil para análise de vendas.
<i>Loja</i>	<i>quantidade</i> – Para registrar a quantidade introduzida quando existir mais de um item em uma linha de item de venda (por exemplo, <i>cinco pacotes de tofu</i>). <i>endereço, nome</i> – O recibo requer o nome e o endereço da loja.

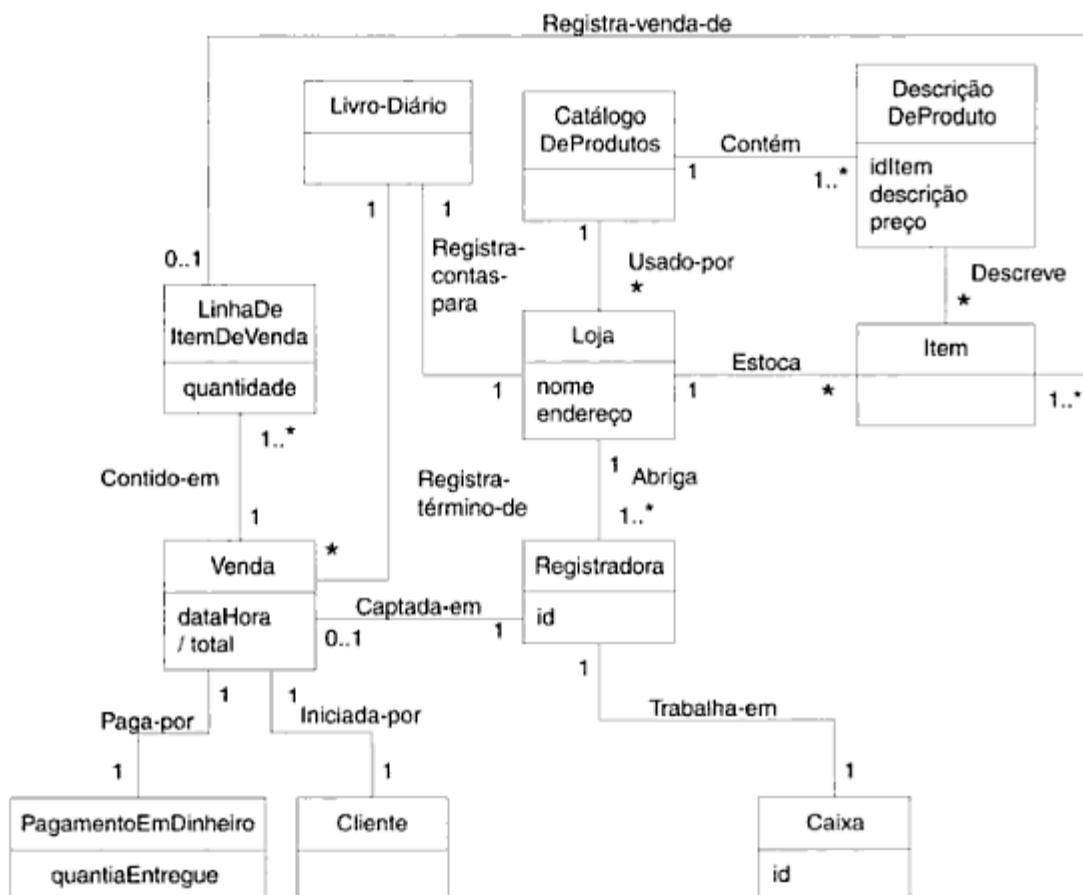


Figura 9.27 Um modelo parcial do domínio PDV ProxGer.

Estudo de caso: Banco Imobiliário

Veja a Figura 9.28. Os atributos escolhidos refletem os requisitos de informação para essa iteração – o cenário simplificado *Jogar Banco Imobiliário* dessa iteração. Por exemplo:

Dado	<i>valorDaFace</i> – Depois de rolar o dado, há necessidade de calcular a distância de um movimento
Casa	<i>nome</i> – Para imprimir a saída desejada de rastreamento.

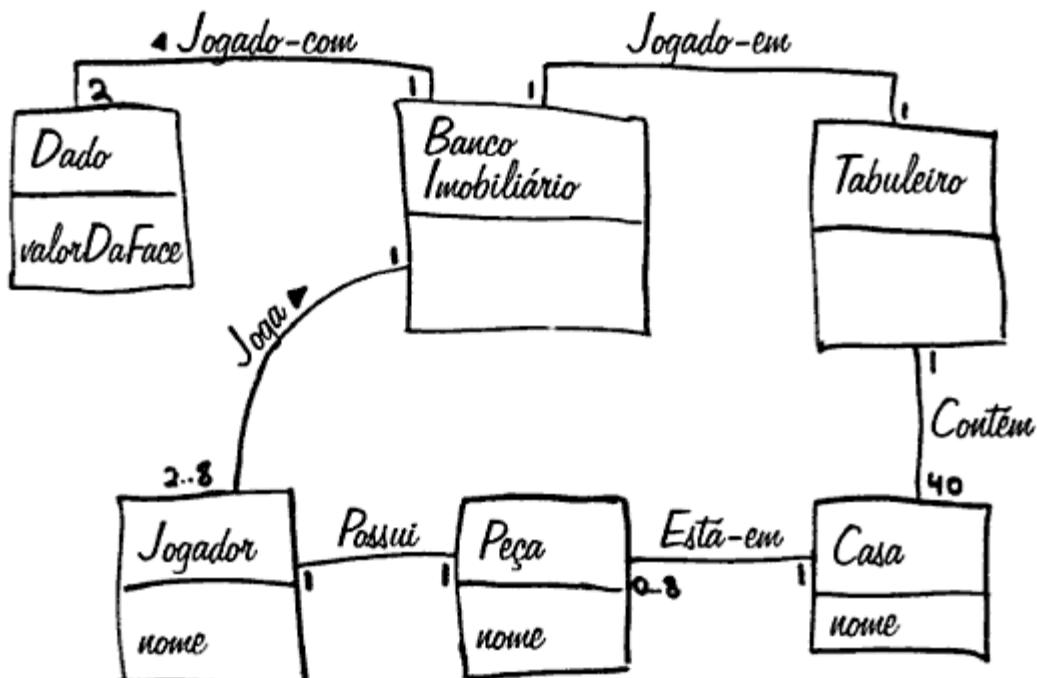


Figura 9.28 Um modelo parcial de domínio Banco Imobiliário.

9.18 Conclusão: o modelo de domínio está correto?

Não existe um modelo de domínio correto. Todos os modelos são aproximações do domínio que estamos tentando entender; o modelo de domínio é primeiramente uma ferramenta de entendimento e comunicação entre os membros de um grupo específico. Um modelo de domínio útil capta as abstrações e informações essenciais necessárias para entender o domínio no contexto dos requisitos correntes e ajuda as pessoas no entendimento do domínio – seus conceitos, terminologia e relacionamentos.

9.19 Processo: modelagem de domínio iterativa e evolutiva

Apesar de paradoxalmente um número significativo de páginas ter sido destinado a explicar a modelagem de domínio, em mãos experientes o desenvolvimento de um modelo (parcial, evolutivo) pode levar apenas 30 minutos para cada iteração. Isso por ser ainda mais diminuído pelo uso de padrões de análise predefinidos.

No desenvolvimento iterativo, evoluímos incrementalmente o modelo de domínio em diversas iterações. Em cada uma, o modelo de domínio é limitado aos cenários anteriores e atuais que estão sendo considerados, em vez de expandir um modelo em estilo cascata muito cedo na tentativa de captar todas as possíveis classes conceituais e relacionamentos. Por exemplo, essa iteração do PDV é limitada somente ao cenário simplificado de pagamento em dinheiro de *Processar Venda*; assim, o modelo parcial de domínio será criado para refletir apenas isso – nada mais.

E para reiterar a sugestão do início deste capítulo:

Diretriz

Evite uma preocupação excessiva com a modelagem em cascata para fazer um modelo de domínio completo ou “correto” – ele não vai ser nada disso e tais esforços de supermodelagem levam à *paralisia de análise*, com pouco ou nenhum retorno no investimento.

Limite a modelagem de domínio a não mais do que algumas horas por iteração.

Modelos de domínio dentro do PU

Como sugerido no exemplo da Tabela 9.4, o Modelo de Domínio do PU geralmente é iniciado e completado na fase de elaboração.

Tabela 9.4 Amostra de artefatos do PU e sua seqüência temporal; i = iniciar; r = refinar

Disciplina	Artefato Iteração →	Concepção C1	Elaboração E1..En	Construção Ct1..Ctn	Transição T1..T2
Modelagem de Negócios	Modelo de Domínio		i		
Requisitos	Modelo de Casos de Uso (DSSs)	i	r		
	Visão	i	r		
	Especificação Suplementar	i	r		
	Glossário	i	r		
Projeto	Modelo de Projeto		i	r	
	Documento de Arquitetura de Software		i		
	Modelo de Dados		i	r	

Concepção

Modelos de domínio não são fortemente motivados na concepção, pois o propósito da concepção não é fazer uma investigação profunda, mas sim decidir se o projeto merece investigação mais detalhada na fase de elaboração.

Elaboração

O Modelo de Domínio é criado durante as iterações da elaboração, quando há necessidade de entender melhor os conceitos importantes e mapear alguns deles para classes de software durante o trabalho de projeto.

O modelo de objetos de negócio do PU vs. modelo de domínio

O Modelo de Domínio do PU é uma variação oficial do menos comum Modelo de Objetos do Negócio do PU, em inglês Business Object Model (BOM). O BOM do PU – que não deve ser confundido com muitas outras definições de BOM – é uma espécie de modelo da empresa que descreve todo o negócio. Ele pode ser usado quando é feita a engenharia ou reengenharia do processo de negócio, independentemente de qualquer aplicação de software específica (como o PDV ProxGer). Para citar:

[O NOM do PU] serve como uma abstração de como os trabalhadores do negócio e as entidades do negócio precisam ser relacionados e de como eles precisam colaborar para realizar o negócio [RUP].

O BOM é representado com vários diagramas diferentes (classes, atividade e seqüência) que ilustram como toda a empresa funciona (ou deveria funcionar). Ele é mais útil se estiver sendo feita engenharia de processo de negócio, mas essa é uma atividade menos comum do que criar uma única aplicação de software.

Conseqüentemente, o PU define o Modelo de Domínio como o artefato mais comumente criado ou especializado no BOM. Para citar:

Você pode escolher desenvolver um modelo de objetos de negócio “incompleto”, com enfoque na explicação das “coisas” e produtos importantes para um domínio. [...] Isso é freqüentemente referido como o modelo de domínio [RUP].

9.20 Leituras recomendadas

Object-Oriented Methods: A Foundation, de Odell, fornece uma sólida introdução para modelagem de domínio conceitual. *Designing Object Systems*, de Cook e Daniel, também é útil.

Analysis Patterns, de Fowler, oferece padrões importantes no modelo de domínio e é definitivamente recomendado. Um outro bom livro que descreve padrões para modelo de domínio é *Data Model Patterns: Conventions of Thought*, de Hay. Sugestão de especialistas em modelagem de dados que entendem a distinção entre modelos conceituais puros e modelos de esquemas de banco de dados podem ser muito úteis para a modelagem de objetos do domínio.

Java Modeling in Color with UML [CDL99] tem muito mais sugestões de modelagem de domínio relevantes do que o título sugere. Os autores identificam padrões comuns em tipos relacionados e suas associações; o aspecto “cor” é realmente uma visualização de categorias comuns desses tipos, tais como *descrições* (azul), *papéis* (amarelo) e *intervalos de tempo* (rosa). A cor é usada para auxiliar a ver os padrões.

DIAGRAMAS DE SEQÜÊNCIA DO SISTEMA

Na teoria, não há diferença entre teoria e prática. Mas na prática há.
– Jan L. A. van de Snepscheut

Objetivos

- Identificar os eventos do sistema.
- Criar diagramas de seqüência do sistema para cenários de casos de uso.

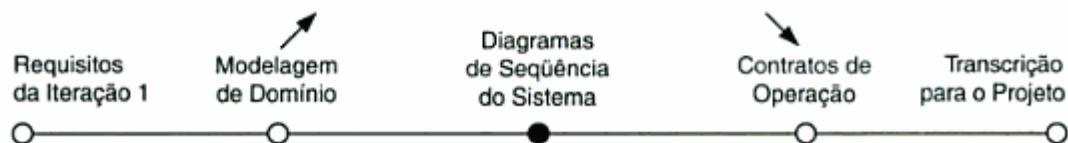
Introdução

Um diagrama de seqüência do sistema é um artefato criado rápida e facilmente que ilustra os eventos de entrada e saída relacionados com o sistema em discussão. Eles são entradas para contratos de operação e – mais importante – projeto de objetos.

A UML contém notação na forma de diagramas de seqüência para ilustrar eventos provenientes de atores externos ao sistema.

O que vem a seguir?

Explorada a modelagem de domínio, este capítulo identifica operações do sistema em um DSS. O seguinte toma essas operações e define o efeito nos objetos do modelo de domínio, usando uma notação de pré e pós-condição do contrato de operação.



A influência dos artefatos no PU enfatizando diagramas de seqüência de sistema é mostrada na Figura 10.1. O texto do caso de uso e seus eventos implícitos no sistema são entradas para a criação do DSS. As operações do DSS (tais como entrarItem) podem, por sua vez, ser analisadas nos contratos de operação, detalhadas no Glossário, e – mais importante – servir como ponto de partida para projetar objetos de colaboração.

10.1 Exemplo: DSS do ProxGer

Um DSS mostra, para uma seqüência específica de eventos dentro de um caso de uso, os atores externos que interagem diretamente com o sistema, o sistema (como uma caixa)

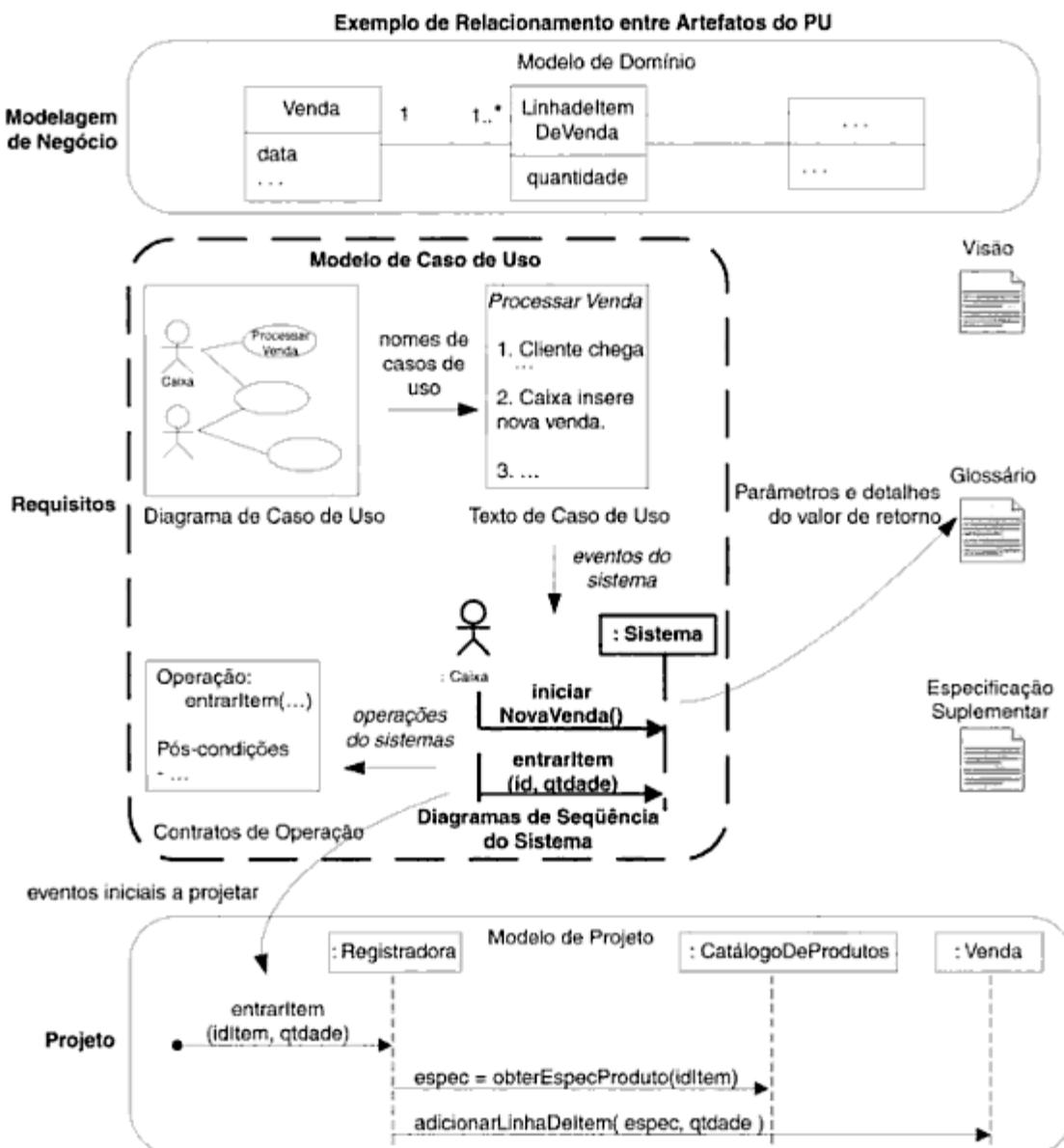


Figura 10.1 Amostra da influência dos artefato no PU.

xa-preta) e os eventos do sistema que os atores geram (ver Figura 10.2). O tempo corre de cima para baixo e a ordem dos eventos deve seguir sua ordem no cenário.

O exemplo da Fig. 10.2 é do cenário de sucesso principal de um cenário de Processar Venda. Ele indica que o caixa gera os eventos do sistema iniciarNovaVenda, entrarItem, finalizarVenda e fazerPagamento. Esses eventos são implícitos ou sugeridos lendo o texto do caso de uso.

10.2 O que são diagramas de seqüência do sistema?

Os casos de uso descrevem como os atores externos interagem com o sistema de software que estamos interessados em criar. Durante essa interação, um ator gera **eventos de sistema** para um sistema, geralmente solicitando alguma **operação de sistema** para tratar o evento. Por exemplo, quando um caixa entra com o ID de um item, ele está solicitando ao sistema PDV que registre a venda daquele item (o evento *entrarItem*). Esse evento inicia uma operação no sistema. O texto do caso de uso implica o evento *entrarItem*, e o DSS torna isso concreto e explícito.

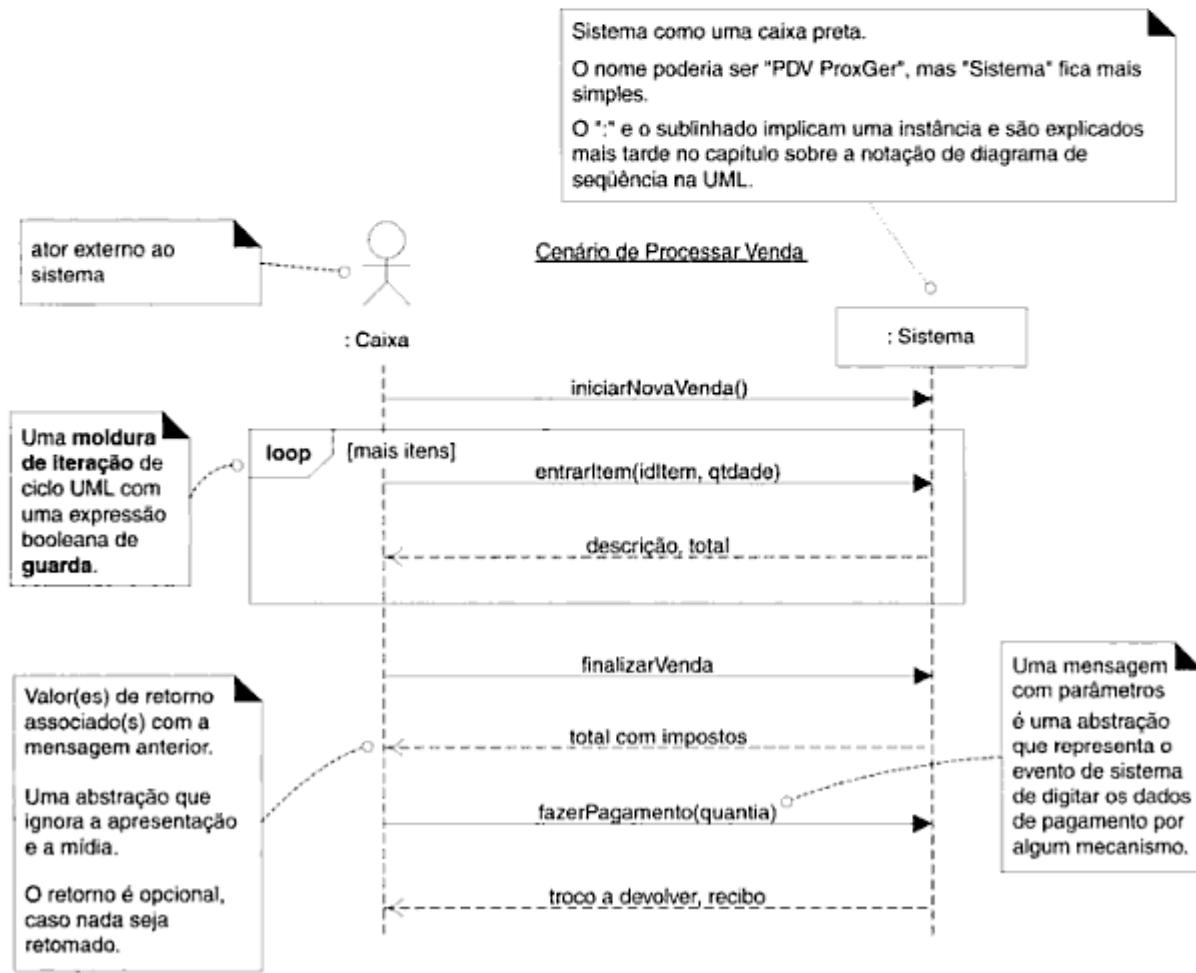


Figura 10.2 DSS para um cenário de Processar Venda.

A UML inclui **diagramas de seqüência**, como uma notação que pode ilustrar as interações de atores e as operações iniciadas por eles.

Um **diagrama de seqüência do sistema** é uma figura que mostra, para um cenário específico de um caso de uso, os eventos que os atores externos geram, sua ordem e os eventos entre sistemas. Todos os sistemas são tratados como uma caixa-preta; a ênfase do diagrama está nos eventos que cruzam a fronteira do sistema de atores para sistemas.

Diretriz

Desenhe um DSS para um cenário de sucesso principal de cada caso de uso e cenários alternativos freqüentes ou complexos.

10.3 Motivação: por que desenhar um DSS?

Uma questão interessante e útil no projeto de software é a seguinte: que eventos estão entrando no nosso sistema? Por quê? Porque temos que projetar o software para tratar esses eventos (desde o mouse, teclado, outro sistema,...) e executar uma resposta. Basicamente, um sistema de software reage a três coisas: (1) eventos externos de atores (seres humanos ou computadores), (2) eventos de tempo e (3) falhas ou exceções (que freqüentemente são de fontes externas).

Portanto, é útil saber quais, precisamente, são os eventos externos de entrada – os **eventos do sistema**. Eles são uma parte importante da análise do comportamento do sistema.

Você pode estar familiarizado com a idéia de identificar as mensagens que entram em um objeto de software. Mas esse conceito é útil em um nível mais alto de componentes, inclusive todo o sistema visto (abstratamente) como uma coisa ou objeto.

Antes de prosseguir para um projeto detalhado de como uma aplicação de software vai funcionar, é útil investigar e definir seu comportamento como uma “caixa-preta”. **Comportamento do sistema** é uma descrição do que um sistema faz, sem explicar como o faz. Uma parte dessa descrição é um diagrama de seqüência do sistema. Outras partes incluem os casos de uso e os contratos de operação do sistema (a serem discutidos posteriormente).

*Contratos
(pág. 203)*

10.4 Aplicação da UML: diagramas de seqüência

*Diagramas
de seqüên-
cia UML
(pág. 247)*

A UML não define alguma coisa denominada diagrama de seqüência do “sistema”, mas simplesmente um “diagrama de seqüência”. A qualificação é usada para enfatizar sua aplicação a sistemas como caixas-pretas. Posteriormente, diagramas de seqüência vão ser usados em outro contexto – para ilustrar o projeto de objetos de software que interagem para realizar trabalho.

Ciclos em diagramas de seqüência

Note na Fig. 10.2 como **molduras de iteração** são usadas para mostrar ciclos em diagramas de seqüência.

10.5 Qual é o relacionamento entre DSSs e casos de uso?

Um DSS mostra eventos do sistema *para um cenário de um caso de uso*; portanto, é gerado a partir da inspeção de um caso de uso (ver Figura 10.3).

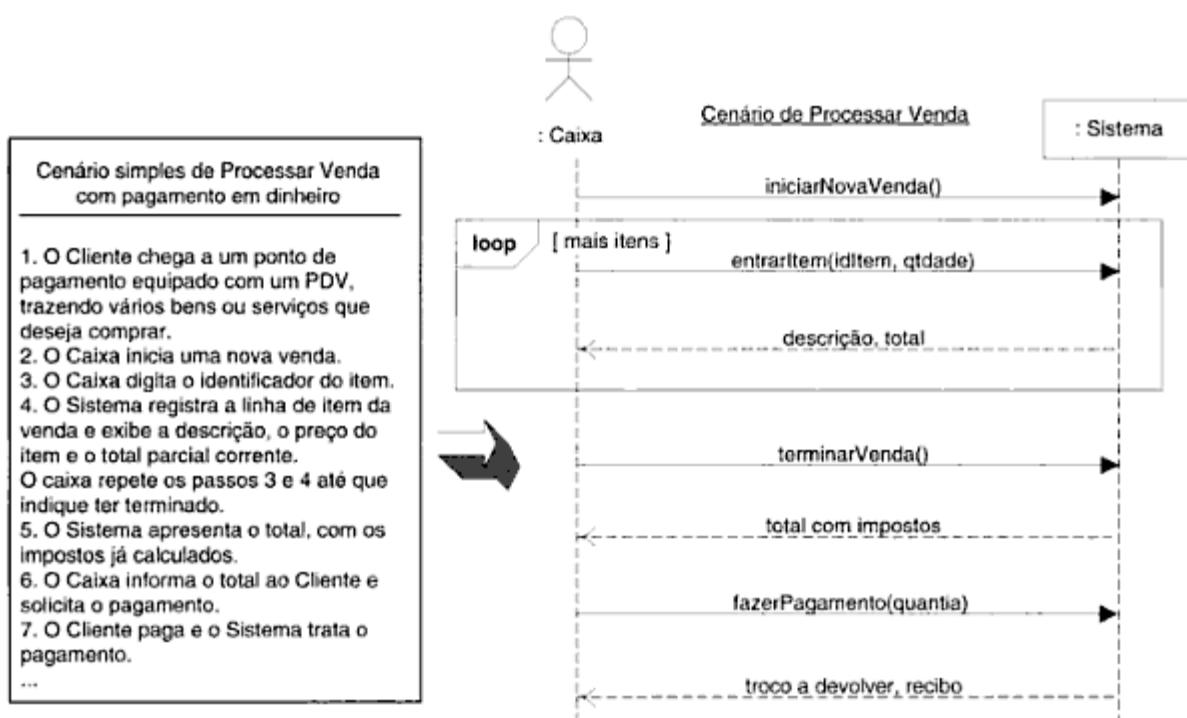


Figura 10.3 DSSs são derivados de casos de uso; eles mostram um cenário.

Aplicação da UML: deve-se mostrar o texto do caso de uso no DSS?

Geralmente não. Se você denomina o DSS adequadamente, pode indicar o caso de uso; por exemplo, Cenário de Processar Venda.

10.6 Como denominar eventos e operações do sistema?

O que é melhor, *escanear(idItem)* ou *entrarItem(idItem)*?

Eventos do sistema devem ser expressos no nível abstrato de intenção e não em termos do dispositivo físico de entrada.

Assim, “*entrarItem*” é melhor que “*escanear*” (isto é, usar o scanner a laser), porque capta a *intenção* da operação ao mesmo tempo em que permanece abstrato e sem

compromissos com relação a escolhas de projeto sobre qual interface é usada para captar o evento do sistema. Poderia ser por meio de scanner a laser, teclado, entrada de voz ou qualquer outra coisa.

Também melhora a clareza começar o nome de um evento de sistema com um verbo (adicionar..., entrar..., terminar..., efetuar...), como na Figura 10.4, pois isso enfatiza que se trata de comandos ou solicitações.

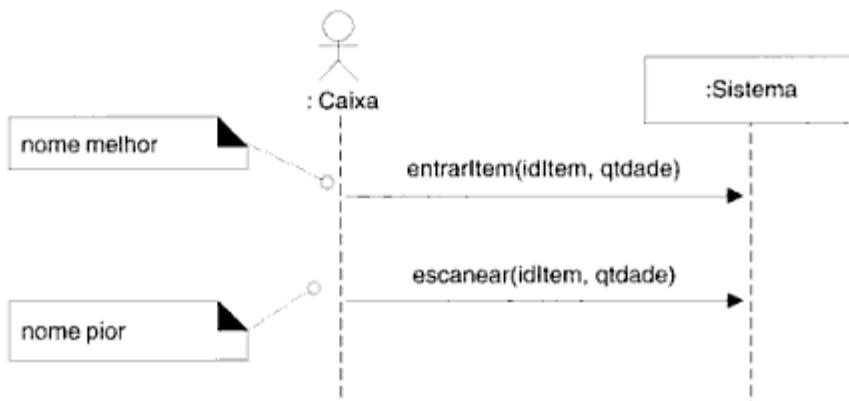


Figura 10.4 Escolher os nomes de eventos e de operações no nível abstrato.

10.7 Como modelar DSSs envolvendo outros sistemas externos?

DSSs inter-sistemas (págs. 412-413) Os DSSs também podem ser usados para ilustrar as colaborações entre sistemas, tais como entre o PDV ProxGer e o autorizador externo de pagamento a crédito. Entretanto, isso é adiado até uma iteração posterior no estudo de caso, uma vez que essa iteração não inclui a colaboração com sistemas remotos.

10.8 Que informação do DSS colocar no Glossário?

Os termos mostrados nos DSSs (nome de operação, parâmetros, dados de retorno) são resumidos. Eles podem necessitar de explicação adequada, de modo que, durante o projeto, fique claro o que está entrando e o que está saindo do sistema. O Glossário é excelente para esses detalhes.

Por exemplo, na Fig. 10.2 há uma linha de retorno contendo a descrição “*troco a devolver, recibo*”. Essa é uma descrição vaga sobre o recibo – um relatório complexo. Assim, o glossário do PU pode ter uma entrada *recibo* que mostra exemplos de recibos (talvez uma figura digital) e conteúdo e leiaute detalhados.

Diretriz

Em geral, para muitos artefatos, mostre detalhes no Glossário.

10.9 Exemplo: DSS do Banco Imobiliário

Texto do caso de uso (pág. 119) O caso de uso *Jogar Banco Imobiliário* é simples como o cenário principal. A pessoa que observa inicia com o número de jogadores e depois solicita a simulação do jogo, observando um rastreamento da saída até que haja um vencedor. Ver Figura 10.5.

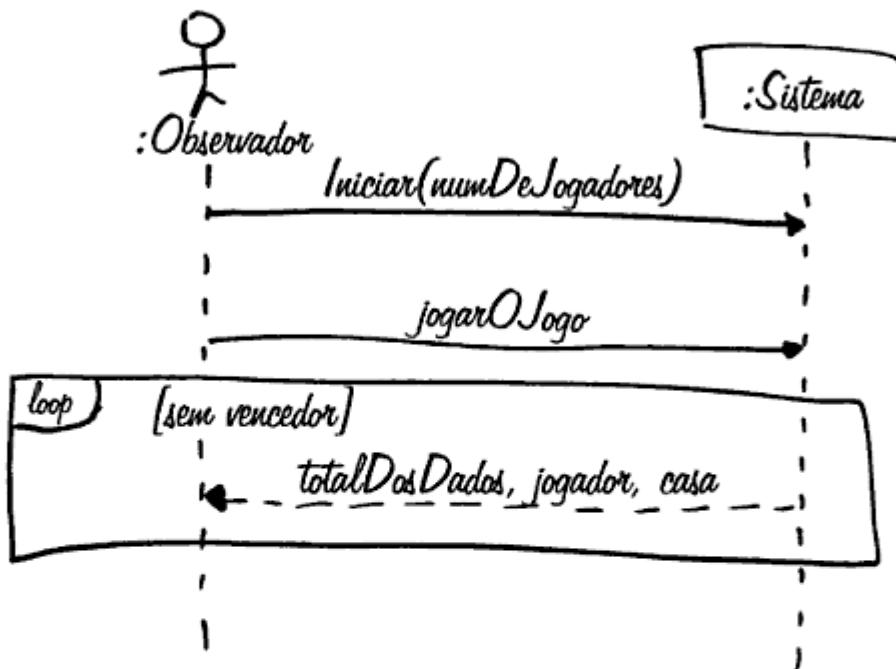


Figura 10.5 DSS para um cenário de *Jogar Banco Imobiliário*.

10.10 Processo: DSSs iterativos e evolutivos

Não crie DSSs para todos os cenários, a menos que você esteja usando uma técnica de estimativas (por exemplo, a contagem de pontos por função) que exija a identificação de todas as operações do sistema. Em vez disso, desenhe-os somente para os cenários escolhidos para a iteração seguinte. Além do mais, eles não devem demorar para ser esboçados – talvez alguns minutos ou meia hora.

DSSs são também muito úteis quando você deseja entender a interface e as colaborações de sistemas existentes ou documentar a arquitetura.

DSSs no PU

Os DSSs são parte do Modelo de Casos de Uso – uma visualização das interações implícitas nos cenários de casos de uso. DSSs não foram explicitamente mencionados na descrição original do PU, embora os criadores do PU estejam cientes da utilidade de tais diagramas. DSSs são um exemplo dos muitos possíveis artefatos e atividades

de análise e projeto ampla e habilidosamente usados que os documentos do PU ou do RUP não mencionam. Mas o PU, sendo muito flexível, encoraja a inclusão de todos e quaisquer artefatos e práticas que adicionam valor.

Fases do PU

Concepção – Os DSSs geralmente não se justificam na fase de concepção, a menos que você esteja fazendo uma estimativa grosseira (não espere que a estimativa na concepção seja confiável) envolvendo uma técnica baseada na identificação das operações do sistemas, como **pontos por função** ou **COCOMO II** (ver www.ifpug.org).

Elaboração – A maioria dos DSSs é criada durante a elaboração, quando então é útil para identificar detalhes dos eventos do sistema, esclarecer quais operações importantes devem ser projetadas para lidar com esses eventos, escrever contratos para as operações do sistema e, possivelmente, fundamentar estimativas (por exemplo, macroestimativas com pontos por função não ajustados e COCOMO II).

10.11 Histórico e leituras recomendadas

Identificar as operações públicas de um sistema de software é uma necessidade muito antiga, assim, variantes de diagramas de interface do sistema que ilustram os eventos de E/S para um sistema tratado como caixa preta têm tido amplo uso há várias décadas. Por exemplo, em telecomunicações eles têm sido chamados de diagramas de fluxo de chamada. Eles foram inicialmente popularizados em métodos OO no método Fusion [Coleman+94], o qual fornecia um exemplo detalhado do relacionamento entre os DSSs e as operações do sistema com outros artefatos de análise e projeto.

CONTRATOS DE OPERAÇÃO

Quando idéias falham, palavras ficam muito convenientes.
– Johann Wolfgang von Goethe

Objetivos

- Definir operações do sistema.
- Criar contratos para as operações do sistema.

Introdução

Casos de uso ou características do sistema são a principal maneira no PU para descrever o comportamento do sistema, e geralmente são suficientes. Às vezes uma descrição mais detalhada ou precisa do comportamento do sistema tem valor. Contratos de operação usam uma forma pré e pós-condição para descrever modificações detalhadas em objetos em um modelo de domínio, como resultado de uma operação do sistema. Um modelo de domínio é o modelo mais comum de análise OO, mas contratos de operação e modelos de estado (introduzidos na pág. 490) também podem ser artefatos úteis relacionados à AOO.

Contratos de operação podem ser considerados parte do Modelo de Casos de Uso do PU, porque fornecem mais detalhes de análise sobre o efeito das operações do sistema implícito nos casos de uso.

O que vem a seguir?

Explorados DSSs e operações do sistema, este capítulo torna as operações e define seu efeito nos objetos do modelo de domínio. Depois, tendo acabado a análise desta iteração, o capítulo seguinte resume a transição para o projeto, ciclo que se repete a cada iteração.



A influência dos artefatos do PU enfatizando os contratos de operação é mostrada na Fig. 11.1. As principais entradas para os contratos são as operações do sistema identificadas nos DSSs (por exemplo, *entrarItem*), o modelo de domínio e o conhecimento do domínio pelos especialistas. Os contratos podem, por sua vez, servir como entrada para o projeto de objetos, pois descrevem as modificações que são provavelmente exigidas nos objetos de software ou banco de dados.

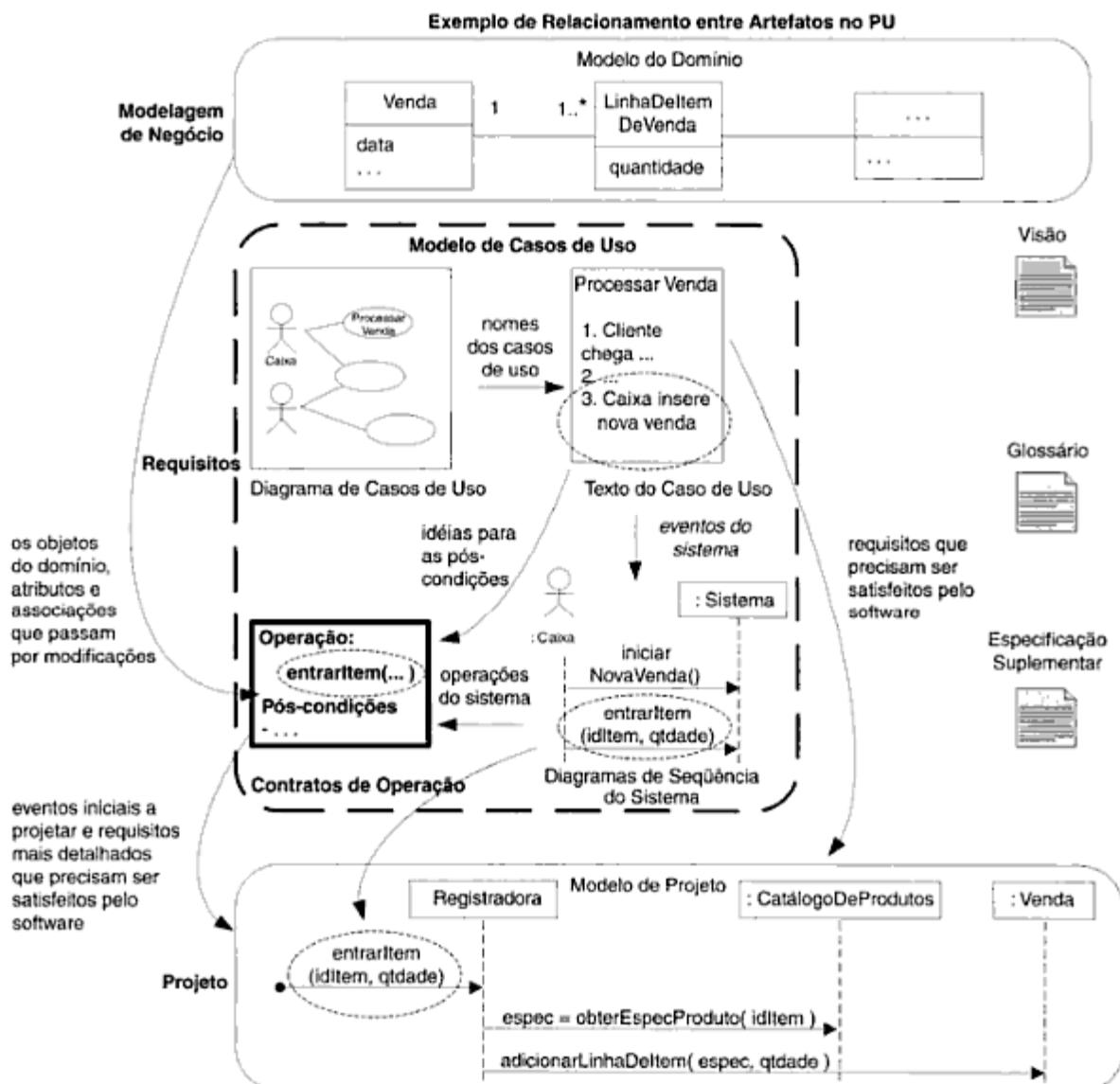


Figura 11.1 Amostra da influência dos artefatos no PU.

11.1 Exemplo

Segue na página seguinte, um contrato de operação para a operação do sistema *entrarItem*. O elemento crucial são as *pós-condições*; as outras partes são úteis, mas menos importantes.

Contrato CO2: entrarItem

Operação:	entrarItem (idItem: IdItem, qtdade: inteiro)
Referências Cruzadas:	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma venda em andamento.
Pós-Condições:	<ul style="list-style-type: none"> – Foi criada uma instância <i>liv</i> de LinhaDelItemDeVenda (<i>criação de instância</i>). – <i>liv</i> foi associada com a Venda corrente (<i>associação formada</i>). – <i>liv.qtdade</i> tornou-se quantidade (<i>modificação de atributo</i>). – <i>liv</i> foi associada a uma EspecificaçãoDeProduto, com base na correspondência de idItem (<i>associação formada</i>).

As categorizações, como “(*criação de instância*)”, são um apoio ao aprendizado, não propriamente parte do contrato.

11.2 Definição: o que são seções de um contrato?

O esquema a seguir mostra uma descrição de cada seção em um contrato.

Operação:	Nome da operação e parâmetros
Referências Cruzadas	Casos de uso nos quais essa operação pode ocorrer
Pré-Condições:	Hipóteses dignas de nota sobre o estado do sistema ou de objetos no Modelo de Domínio antes da execução da operação. São hipóteses não triviais que o leitor deveria saber que foram formuladas.
Pós-Condições:	É a seção mais importante. O estado dos objetos no Modelo de Domínio, depois de concluída a operação. Será discutida em detalhes em uma seção a seguir.

11.3 Definição: o que é uma operação do sistema

Contratos de operação podem ser definidos para **operações de sistema** – operações que o sistema, como um componente caixa-preta, oferece na sua interface pública. As operações do sistema podem ser identificadas enquanto se esboçam os DSSs, como na Figura 11.3. Para ser mais preciso, os DSSs mostram **eventos do sistema** – eventos ou mensagens de E/S relativas ao sistema. Os eventos de entrada do sistema implicam que o sistema tenha operações do sistema para tratar os eventos, tal como uma mensagem OO (uma espécie de evento ou sinal) é tratada por um método OO (uma espécie de operação).

O conjunto completo de operações do sistema, em todos os casos de uso, define a **interface pública do sistema**, vendo-o como um único componente ou classe. Na UML, todo o sistema pode ser representado como um objeto de uma classe denominada (por exemplo) *Sistema*.

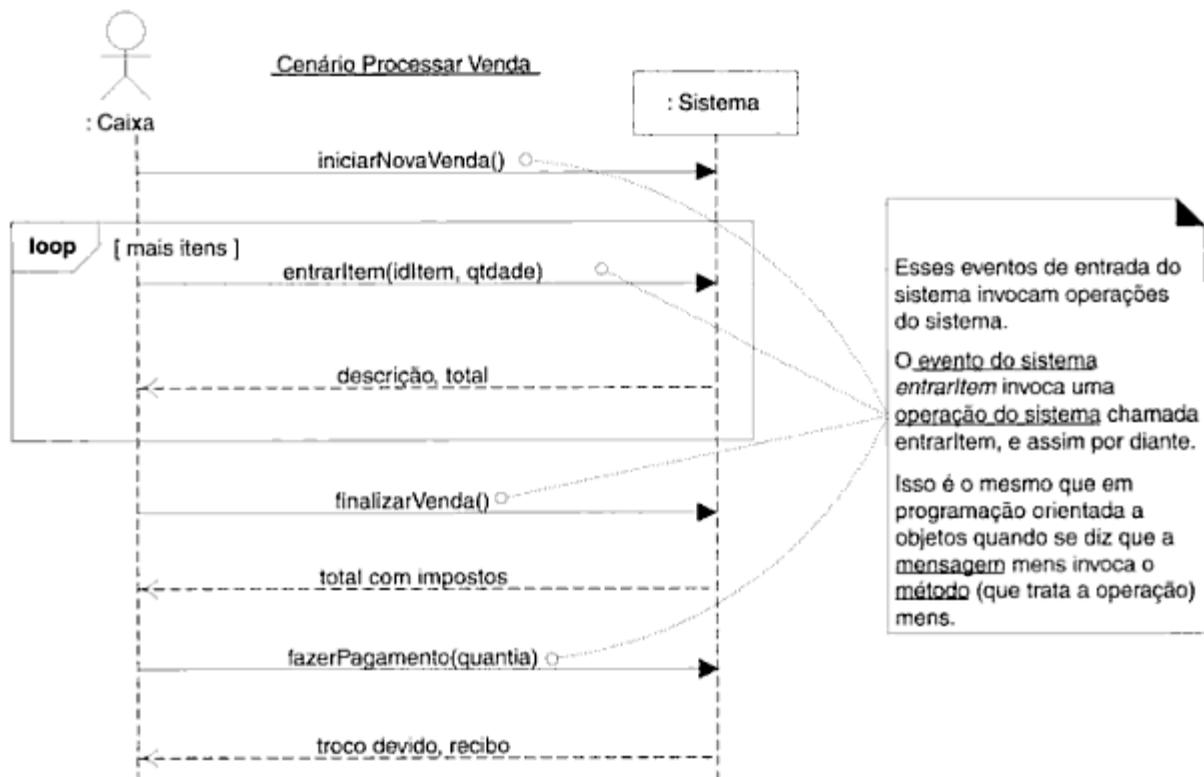


Figura 11.2 DSS. Operações do sistema tratam eventos de entrada do sistema.

11.4 Definição: pós-condições

Note que cada uma das pós-condições no exemplo *entrarItem* incluiu uma categorização de ajuda o aprendizado, por exemplo, *criação de instância* ou *associação formada*. Aqui, temos um ponto-chave:

Definição

As **pós-condições** descrevem modificações no estado dos objetos do modelo de domínio. Modificações no estado do modelo de domínio incluem instâncias criadas, associações formadas ou desfeitas e atributos modificados.

Pós-condições não são ações a serem executadas durante a operação; em vez disso, são *observações* sobre objetos do modelo de domínio que se tornam verdadeiras ao término da operação – *depois que a poeira baixou*.

Resumindo, as pós-condições se enquadram nas seguintes categorias:

- Criação e exclusão de instância
- Modificação de valor de atributo
- Associações (para ser preciso, *ligações UML*) formadas e desfeitas

Quebrar associações é raro. Mas, como exemplo, considere uma operação para permitir a exclusão de linhas de itens. A pós-condição poderia afirmar “a associação selecionada *LinhaDeItemDeVenda* com *Venda* foi desfeita”. Em outros domínios, quando um empréstimo é pago ou alguém cancela sua afiliação a algo, as associações são desfeitas.

As pós-condições relativas à exclusão de instâncias são raras, porque não costumamos nos preocupar em forçar a destruição de algo no mundo real. Como exemplo: em muitos países, depois que uma pessoa abriu falência e se passaram sete ou dez anos, todos os registros de sua abertura de falência devem ser destruídos por lei. Observe que essa é uma perspectiva conceitual, não de implementação. Essas não são declarações sobre liberar memória de computador ocupada por objetos de software.

Como as pós-condições são relacionadas ao modelo de domínio?

As pós-condições são expressas no contexto de objetos do Modelo de Domínio. Que instâncias podem ser criadas? As do Modelo de Domínio. Que associações podem ser formadas? As associações no Modelo de Domínio; e assim por diante.

Motivação: por que pós-condições?

Primeiro, elas nem sempre são necessárias. Freqüentemente, o efeito de uma operação do sistema é relativamente claro para os desenvolvedores devido à leitura do caso de uso, a conversas com especialistas ou a sua própria experiência. Mas, algumas vezes, mais detalhes e precisão são úteis. Contratos oferecem isso.

Note que as pós-condições apóiam detalhe de granularidade fina e precisão na declaração de qual deve ser o resultado da operação. É também possível expressar esse nível de detalhe nos casos de uso, mas indesejável – eles ficariam muito prolixos e detalhados em baixo nível.

Um contrato é uma excelente ferramenta de análise de requisitos ou AOO que descreve em muito detalhe as modificações exigidas por uma operação do sistema (em termos de objetos do modelo do domínio) sem ter que descrever *como* elas têm de ser conseguidas.

Em outras palavras, o *projeto* pode ser adiado e podemos focalizar a *análise* de *o quê* deve acontecer, em vez do *como* isso deve ser conseguido.

Considere as pós-condições:

Pós-Condições:

- Foi criada uma instância *liv* de *LinhaDeItemDeVenda* (criação de instância).
- *liv* foi associada à *Venda* corrente (associação formada).
- *liv.qtdade* tornou-se *quantidade* (modificação de atributo).
- *liv* foi associada a uma *DescriçãoDoProduto*, com base na correspondência de um *idItem* (associação formada).

Nenhum comentário é feito sobre o modo como uma instância de *LinhaDeItemDeVenda* é criada ou como ela é associada a uma *Venda*. Isso poderia ser uma instrução sobre escrever algo em pedaços de papel e grampeá-los, usar tecnologia Java para criar objetos de software e conectá-los ou inserir linhas em um banco de dados relacional.

Diretriz: como escrever uma pós-condição?

Expresse as pós-condições no pretérito perfeito para enfatizar que elas são *observações* sobre mudanças de estado que resultaram de uma operação, não uma ação a acontecer. É por isso que elas são chamadas **pós-condições**! Por exemplo:

- (melhor) Uma *LinhaDeItemDeVenda* foi criada.
em vez de
- (pior) Criar uma *LinhaDeItemDeVenda*, ou Uma *LinhaDeItemDeVenda* é criada

Analogia: o espírito das pós-condições: o palco e a cortina

Por que escrever pós-condições no pretérito perfeito? Pense nelas usando a seguinte imagem:

O sistema e seus objetos são apresentados no palco de um teatro.

1. Antes da operação, tire uma foto do palco.
2. Feche as cortinas do palco e aplique a operação de sistema (*barulho de batidas, gritos, guinchos, etc*).
3. Abra as cortinas e tire uma segunda foto.
4. Compare as fotos de antes e depois e expresse como pós-condições as mudanças no estado do palco (*foi criada uma LinhaDeItemDeVenda*).

Diretriz: quão completas as pós-condições devem ser?

Análise pesada x ágil

Contratos podem ser inúteis. Essa questão é discutida em uma seção subsequente. Entretanto, considerando que alguns são úteis, gerar um conjunto detalhado e completo de pós-condições para todas as operações do sistema não é provável – ou necessário. No espírito da modelagem ágil, trate sua criação como uma estimativa inicial, entendendo que os contratos não estarão completos e que especificações completas “perfeitas” raramente são possíveis ou críveis.

Mas entender que análise leve é realista e hábil não significa abandonar um pouco de investigação antes da programação – esse é outro extremo do mau entendimento.

11.5 Exemplo: pós-condições de entrarItem

A seção a seguir trata da motivação para as pós-condições da operação de sistema *entrarItem*.

Criação e exclusão de instância

Depois que o *idItem* e a *qtdade* de um item foram introduzidos, que novo objeto deve ter sido criado? Uma *LinhaDeItemDeVenda*. Assim:

- Foi criada uma instância *liv* de *LinhaDeItemDeVenda* (criação de instância).

Observe a denominação da instância. Esse nome simplificará as referências à nova instância em outras declarações de pós-condições.

Modificação de atributo

Depois que um *idItem* e a sua quantidade foram introduzidos pelo caixa, que atributos de objetos novos ou existentes devem ser modificados? A *quantidade* de *LinhaDelItemDeVenda* deve ter se tornado igual ao parâmetro *qtdade*. Dessa forma:

- *liv.quantidade* torna-se *qtdade* (modificação de atributo).

Associações formadas e desfeitas

Depois que *idItem* e *quantidade* de um item tiverem sido introduzidos pelo caixa, que associações entre objetos novos ou existentes devem ser formadas ou desfeitas? A nova *LinhaDelItemDeVenda* deve ter sido relacionada à sua venda e relacionada à sua *DescriçãoDoProduto*. Assim:

- *liv* foi associado à *Venda* atual (associação formada).
- *liv* foi associado a uma *DescriçãoDoProduto*, baseada em uma correspondência com o *idItem* (associação formada).

Observe a indicação informal de que ela forma um relacionamento com uma determinada *DescriçãoDoProduto* – aquela cujo *idItem* corresponde ao parâmetro. Abordagens de linguagem mais sofisticadas e formal são possíveis, tais como a *Object Constraint Language* (OCL). Recomendação: prefira a simplicidade.

11.6 Diretriz: devemos atualizar o modelo de domínio?

Durante a criação dos contratos, é comum descobrir a necessidade de registrar novas classes conceituais, atributos ou associações no modelo de domínio. Não fique limitado pela definição prévia do modelo de domínio; melhore-o à medida que fizer novas descobertas enquanto pensa nos contratos de operações.

Em métodos iterativos e evolutivos (e refletindo a realidade dos projetos de software) todos os artefatos de análise e projeto são considerados parciais e imperfeitos e evoluem em resposta a novas descobertas.

11.7 Diretriz: quando os contratos são úteis?

No PU, os casos de uso são o repositório principal de requisitos para o projeto. Eles podem fornecer a maioria ou todos os detalhes necessários para saber o que fazer no projeto, caso em que os contratos não são de muita valia. Entretanto, há situações em que os detalhes e a complexidade das mudanças de estado necessárias são inadequados ou muito detalhados para captar em casos de uso.

Por exemplo, considere um sistema de reserva de passagens aéreas e a operação de sistema *adicionarNovaReserva*. A complexidade é muito alta se levarmos em conta todos os objetos do domínio que devem ser modificados, criados e associados. Esses detalhes de granularidade fina *podem* ser escritos no caso de uso associado com essa operação, porém isso o tornará extremamente detalhado (por exemplo, anotando cada atributo em todos os objetos que devem ser modificados).

Observe que o formato de pós-condição oferece e incentiva o uso de uma linguagem muito precisa, analítica e rigorosa, que favoreça uma abrangência detalhada.

Se os desenvolvedores conseguirem compreender com relativa facilidade o que fazer sem eles, evite redigir contratos.

Esse estudo de caso mostra mais contratos do que o necessário por razões didáticas. Na prática, a maioria dos detalhes que eles registram são obviamente dedutíveis do texto do caso de uso. Entretanto, “óbvio” é um conceito muito escorregadio!

11.8 Diretriz: como criar e redigir contratos

Siga os seguintes conselhos ao criar contratos:

1. Identifique as operações do sistema a partir dos DSSs.
2. Para as operações do sistema complexas e talvez sutis em seus resultados ou que não estão claras no caso de uso, construa um contrato.
3. Para descrever pós-condições, use as seguintes categorias:
 - instâncias criadas e excluídas
 - atributos modificados
 - associações formadas e desfeitas

Redação de contratos

- Conforme mencionado, escreva as pós-condições de forma declarativa, no pretérito perfeito (*foi...*) para enfatizar a *observação* de uma modificação de estado, não de um projeto sobre como ela será alcançada. Por exemplo:
 - (melhor) Uma *LinhaDeItemDeVenda* **foi** criada.
 - (pior) Crie uma *LinhaDeItemDeVenda*.
- Lembre-se de estabelecer uma associação entre objetos existentes ou recém-criados. Por exemplo, não é suficiente que uma nova instância de *LinhaDeItemDeVenda* seja criada quando a operação *entrarItem* ocorre. Depois que a operação foi completada, também deve ser verdade que a instância recém-criada esteja associada com *Venda*; assim:
 - A *LinhaDeItemDeVenda* **foi** associada à *Venda* (associação formada).

Qual é o erro mais comum?

O problema mais comum é esquecer de incluir a *formação de associações*. Quando novas instâncias são criadas, é muito provável que associações com vários objetos precisem ser estabelecidas. Não se esqueça!

11.9 Exemplo: contratos PDV ProxGer

Operações de sistema do caso de uso Processar Venda

Contrato CO1: criarNovaVenda

Operação:	criarNovaVenda()
Referências Cruzadas:	Casos de Uso: Processar Venda
Pré-Condições:	Nenhuma.
Pós-Condições:	<ul style="list-style-type: none"> – Foi criada uma instância v de Venda (criação de instância). – v foi associada com Registradora (associação formada). – Os atributos de v foram iniciados.

Observe a descrição vaga na última pós-condição. Se for suficiente para ser entendida, tudo bem.

Em um projeto, todas essas pós-condições particulares são tão óbvias pelo caso de uso que o contrato para *criarNovaVenda* provavelmente não deve ser redigido.

Lembre-se de um dos princípios de processos saudáveis e do uso do PU: mantenha-o o mais leve possível e evite criar todos os artefatos, a menos que eles realmente acrescentem valor.

Contrato CO2: entrarItem

Operação:	entrarItem(idItem: IdItem, qtdade: inteiro)
Referências Cruzadas:	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma venda em andamento.
Pós-Condições:	<ul style="list-style-type: none"> – Foi criada uma instância liv da LinhaDelitemDeVenda (criação de instância). – liv foi associada com a venda corrente (associação formada). – liv.quantidade tornou-se quantidade (modificação de atributo). – liv foi associada a uma DescriçãoDoProduto com base na correspondência do idItem (associação formada).

Contrato CO3: finalizarVenda

Operação:	finalizarVenda()
Referências Cruzadas:	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma venda em andamento.
Pós-Condições	– Venda.estáCompleta tornou-se verdadeira (modificação de atributo).

Contrato CO4: fazerPagamento

Operação:	fazerPagamento(quantia: Moeda)
Referências Cruzadas:	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma venda em andamento.
Pós-Condições	– Foi criada uma instância p de Pagamento (criação de instância). – p.quantiaFornecida tornou-se quantia (modificação de atributo). – p foi associada com a Venda corrente (associação formada). – A Venda corrente foi associada com a Loja (associação formada); (para acrescentá-la ao registro histórico de vendas completadas).

Modificações no modelo de domínio PDV

Existe pelo menos um ponto sugerido por esses contratos que ainda não está representado no modelo de domínio: o término da entrada de itens para a venda. A especificação *finalizarVenda* modifica isso e é provavelmente uma boa idéia, durante o trabalho de projeto, fazer a operação *fazerPagamento* testá-lo para evitar pagamentos até que uma venda seja completada (o que quer dizer que não há mais itens a adicionar).

Uma maneira de representar essa informação é com um atributo *estáCompleta* em *Venda*:

Venda
estáCompleta: Boolean dataHora

Existem alternativas que devem ser consideradas especialmente durante o trabalho de projeto. Uma técnica é chamada **padrão Estado (State)**. Outra é o uso de objetos “sessão”, que rastreiam o estado de uma sessão, não permitindo operações fora de ordem; isso também será explorado adiante.

11.10 Exemplo: contratos do Banco Imobiliário

Vou usar esse estudo de caso para enfatizar que muitos artefatos de análise não são sempre necessários, inclusive contratos. O PU incentiva *evitar* criar um artefato a menos que ele se destine a um risco ou resolva um problema real. Pessoas que conhecem as regras do jogo da experiência quando criança ou adolescente (a maioria das pessoas, parece) podem implementá-lo sem ter que consultar muitos detalhes escritos.

11.11 Aplicação de UML: operações, contratos e a OCL

Quais são os relacionamentos entre contratos neste capítulo e a UML?

A UML formalmente define **operações**. Para citar:

Uma operação é uma especificação de uma transformação ou consulta que um objeto pode ser chamado para executar [RJB99].

Por exemplo, os elementos de uma interface são operações em termos UML. Uma operação é uma abstração, não uma implementação. Já um **método** (na UML) é a implementação de uma operação. Para citar:

[Um método é] a implementação de uma operação. Ele especifica o algoritmo ou procedimento associado com uma operação. [OMG03a]

No metamodelo UML, uma operação UML tem uma **assinatura** (nome e parâmetros) e, mais importante nesse contexto, está associada a um conjunto de restrições de objetos UML classificadas como pré e pós-condições que especificam a semântica da operação.

Para resumir: a UML define a semântica da operação por meio de restrições específicáveis no estilo de pré e pós-condições. Note que, como enfatizado neste capítulo, uma especificação de uma operação UML *não* pode mostrar um algoritmo ou solução, mas apenas as modificações de estado ou os efeitos da operação.

Além de usar contratos para especificar operações públicas de todo o *Sistema* (isto é, operações do *sistema*), os contratos podem ser aplicados a operações em qualquer nível de granularidade: às operações públicas (ou interface) de um subsistema, a um componente, a uma classe abstrata, etc. Por exemplo, operações podem ser definidas para uma única classe de software, como *Pilha*. As operações de granularidade grossa discutidas neste capítulo pertencem a uma classe *Sistema*, representando o sistema total como um componente caixa-preta, mas, na UML, operações podem pertencer a qualquer classe ou interface, todas com pré e pós-condições.

Contratos de operações expressos com a OCL

O formato de pré e pós-condição neste capítulo é linguagem natural informal – perfeitamente aceitável na UML e desejável para ser facilmente entendido.

Mas, também associada à UML, existe uma linguagem formal e rigorosa chamada *Object Constraint Language (OCL)* [WK99], que pode ser usada para expressar restrições de operações UML.

Diretriz

A menos que haja uma boa razão prática para exigir que o pessoal aprenda e use a OCL, mantenha a simplicidade e use linguagem natural. Embora eu esteja certo de que há aplicações reais – e úteis –, nunca vi um projeto que usou OCL, embora visite muitos clientes e projetos.

A OCL define um formato oficial para especificar pré e pós-condições para operações, como mostrado neste fragmento:

```
System::criarNovaVenda( )
  pre: <comandos em OCL>
  post: ...
```

Mais detalhes da OCL estão fora do escopo desta introdução.

11.12 Processo: contratos de operações no PU

Um contrato com pré e pós-condições é um estilo bem conhecido de especificar uma operação na UML. Na UML, existem muitos níveis de operações, desde *Sistema* até classes de granularidade fina, como *Venda*. Os contratos de operações no nível de *Sistema* são parte do Modelo de Casos de Uso, embora não fossem formalmente destacados na documentação original do RUP ou do PU; a sua inclusão nesse modelo foi verificada em conjunto com os autores do RUP.¹

Fases

Concepção – Contratos não são justificados durante a concepção – eles são detalhados demais.

Elaboração – Caso os contratos sejam usados, a maioria será redigida durante a elaboração, quando a maior parte dos casos de uso também é redigida. Somente redija contratos para as operações de sistema mais complexas e sutis.

11.13 Histórico

Os contratos de operação surgiram na área de especificações formais em ciência da computação, originalmente do engenheiro Tony Hoare. Hoare estava trabalhando na

¹ Comunicação particular.

indústria em meados da década de 1960 para desenvolver um compilador ALGOL 60 e leu Bertrand Russell, *Introduction to Mathematical Philosophy*, que apresentou a ele a idéia da teoria axiomática e asserções. Ele constatou que programas de computador podem ser expressos com asserções (pré e pós-condições) relativas aos resultados esperados no lançamento e no término de um programa. Em 1968, ele se uniu à academia e sua idéia se expandiu, juntamente com outras teorias de pesquisadores de especificações formais.

Em 1974, no Laboratório da IBM em Viena, um compilador PL/1 estava sendo desenvolvido e os pesquisadores desejavam uma especificação formal não ambígua da linguagem. Em decorrência dessa necessidade, a VDL –*Vienna Definition Language* – foi criada por Peter Lucas. A VDL emprestou a forma de asserções de pré e pós-condição inicialmente explorada por Hoare e Russel. A VDL posteriormente evoluiu para a linguagem usada no *Vienna Definition Method* (VDM), um método que aplica especificação formal em contratos de operação e teoria rigorosa de prova [BJ78].

Nos anos 1980, Bertrand Meyer – não surpreendentemente outro escritor de compiladores (para a linguagem OO Eiffel) – começou a promover o uso de asserções de pré e pós-condições como elementos de primeira classe na sua linguagem Eiffel para ser aplicada à A/POO. Ele contribuiu fortemente para uma percepção muito mais ampla de especificações formais e contratos de operação em seu popular livro *Object-Oriented Software Construction*, em que também propôs a abordagem como um método chamado de **Projeto por contrato** (*Design by Contract* – DBC). Em DBC, contratos são escritos para operações de fina granularidade de classes de software, não especificamente as operações públicas do “sistema” global. Além disso, DBC promove uma seção *invariante*, comum nas especificações de contrato. Invariantes definem elementos que não devem mudar de estado antes e depois de a operação ser executada. Invariantes não foram usadas neste capítulo por causa da simplicidade.

No início dos anos 1990, Grady Booch discutiu brevemente a aplicação de contratos em operações de objetos em seu **Método de Booch**. Também Derek Coleman e colegas do Laboratório da HP emprestaram a idéia de contratos de operação e aplicaram-na em AOO e modelagem de domínio, tornando-os parte do influente **método Fusion** para A/POO [Coleman+94].

Linguagem de Programação como Apoio para Contratos

Algumas linguagens, como Eiffel, têm apoio de primeira classe para invariantes e pré e pós-condições. Usando atributos, *tags Javadoc*, ou pré compiladores, facilidades análogas podem ser fornecidas em Java e C#, por exemplo.

11.14 Leituras recomendadas

Muitos exemplos de contratos de operação de sistemas OO podem ser encontrados em *Object-Oriented Development: The Fusion Method*, de Coleman, et al. *Object-Oriented Software Construction*, de Meyer, mostra muitos exemplos de contrato para pro-

gramas em Eiffel. Na UML, os contratos de operação também podem ser especificados de forma mais rigorosa pela *Object Constraint Language* (OCL), sobre a qual o livro *The Object Constraint Language: Precise Modeling with UML*, de Warmer e Kleppe, é recomendado.

DOS REQUISITOS PARA O PROJETO – ITERATIVAMENTE

Hardware, subst.: as partes de um sistema de computador que podem ser chutadas.
– anônimo

Objetivos

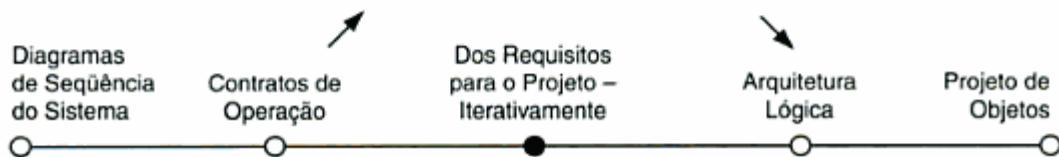
- Motivar rapidamente a transição para as atividades de projeto.
- Contrastar a importância das habilidades de projeto de objetos *versus* o conhecimento da notação UML.

Introdução

Até o presente momento, estudos de caso têm enfatizado a análise dos requisitos e objetos. Seguindo as diretrizes do PU, talvez 10% dos requisitos sejam investigados na concepção, e uma investigação um pouco mais profunda tem início na primeira iteração da elaboração. Os próximos capítulos mudam a ênfase para o projeto de uma solução para essa iteração em termos de objetos de software que colaboram entre si.

O que vem a seguir?

Explorados os contratos de operação, este capítulo conclui o trabalho de análise e resume a transição para o projeto, ciclo que se repete a cada iteração. O capítulo seguinte introduz uma arquitetura de projeto lógico para os estudos de caso, com base no padrão de Camadas.



12.1 Fazer a coisa certa, fazer certo a coisa, iterativamente

Os requisitos e a análise orientada a objetos focalizaram o aprendizado em *fazer a coisa certa*, isto é, entender alguns dos objetivos mais importantes dos estudos de caso e regras e restrições relacionadas. Em contraposição, o trabalho de projeto seguinte estará concentrado em *fazer certo a coisa*, isto é, projetar de maneira competente uma solução para satisfazer os requisitos dessa iteração.

No desenvolvimento iterativo, em cada iteração ocorre a transição de um foco direcionado principalmente aos requisitos e à análise para um foco direcionado principalmente ao projeto e à implementação. As iterações iniciais vão empregar relativamente mais tempo em atividades de análise. À medida que a visão e as especificações começam a se estabilizar com base na programação, teste e realimentação iniciais, nas iterações posteriores é comum que a análise diminua; há maior foco em apenas construir a solução.

12.2 Provocação das modificações iniciais

É natural e saudável descobrir e modificar alguns requisitos durante o trabalho de projeto e implementação, especialmente nas iterações iniciais. Métodos iterativos e evolutivos “acolhem modificações” – embora tentemos provocar essa modificação inevitável nas iterações *iniciais*, de modo que tenhamos um objetivo mais estável (e orçamento e cronograma) para as iterações finais. Programação, teste e demos iniciais ajudam a *provocar* as inevitáveis modificações logo no começo. Lembre! Essa idéia simples é fundamental para entender por que o desenvolvimento iterativo funciona.

A descoberta de que especificações mudam vai tanto esclarecer o objetivo do trabalho de projeto desta iteração quanto refinar o entendimento dos requisitos para as iterações futuras. Ao longo dessas primeiras iterações de elaboração, a descoberta de requisitos deve se estabilizar, de modo que ao final da elaboração talvez 80% dos requisitos estejam definidos de maneira segura – definidos e refinados como resultado de realimentação, programação e teste iniciais, em vez de especulação, como ocorre em um método em cascata.

12.3 Toda aquela análise e modelagem não demorou semanas para ser feita?

Após muitos capítulos de discussão detalhada, com certeza parece que a modelagem prévia exigiu semanas de trabalho. Não é bem assim. Quando nos sentimos à vontade na redação de casos de uso, de modelagem de domínio, etc., toda a modelagem examinada até aqui, para ser realista, dura *poucas* horas ou dias.

Entretanto, isso não significa que se passaram poucos dias desde o início do projeto. Muitas outras atividades, como a programação para demonstração de conceitos, a busca de recursos (pessoas, software, etc.), o planejamento, a preparação do ambiente, etc., podem consumir algumas semanas de preparação.

ARQUITETURA LÓGICA E DIAGRAMA DE PACOTES UML

0x2B | ~0x2B.
— Hamlet

Objetivos

- Introduzir uma arquitetura lógica usando camadas.
- Ilustrar a arquitetura lógica usando diagramas de pacotes UML.

Introdução

Primeiro, para estabelecer o nível de expectativa: esta é uma *introdução* muito breve do tópico de arquitetura lógica, um tópico razoavelmente grande. Aprenda mais a partir da pág. 560.

Agora que fizemos a transição do trabalho orientado à análise para o projeto de software, vamos começar em larga escala. Neste nível, o projeto de um sistema OO típico é baseado em várias camadas arquiteturais, tais como uma camada de IU, uma camada de aplicação lógica (ou “domínio”), etc. Este capítulo explora resumidamente uma camada lógica de arquitetura e a notação UML relacionada.

O que vem a seguir?

Feita a transição para o projeto desta iteração, este capítulo introduz uma arquitetura lógica de projeto. O capítulo seguinte resume o projeto de objetos, modelagem estática e dinâmica e a importância relativa da capacidade de projeto sobre a habilidade com a notação UML.



A influência de artefatos de PU, enfatizando a arquitetura lógica (AL), é mostrada na Figura 13.1. Diagramas de pacotes UML podem ilustrar a AL como parte do modelo de projeto – e também ser resumidos como uma visão no Documento de Arquitetura do Software. A principal entrada são as forças arquiteturais captadas na Especificação Suplementar. A AL define os pacotes dentro dos quais as classes do software são definidas.

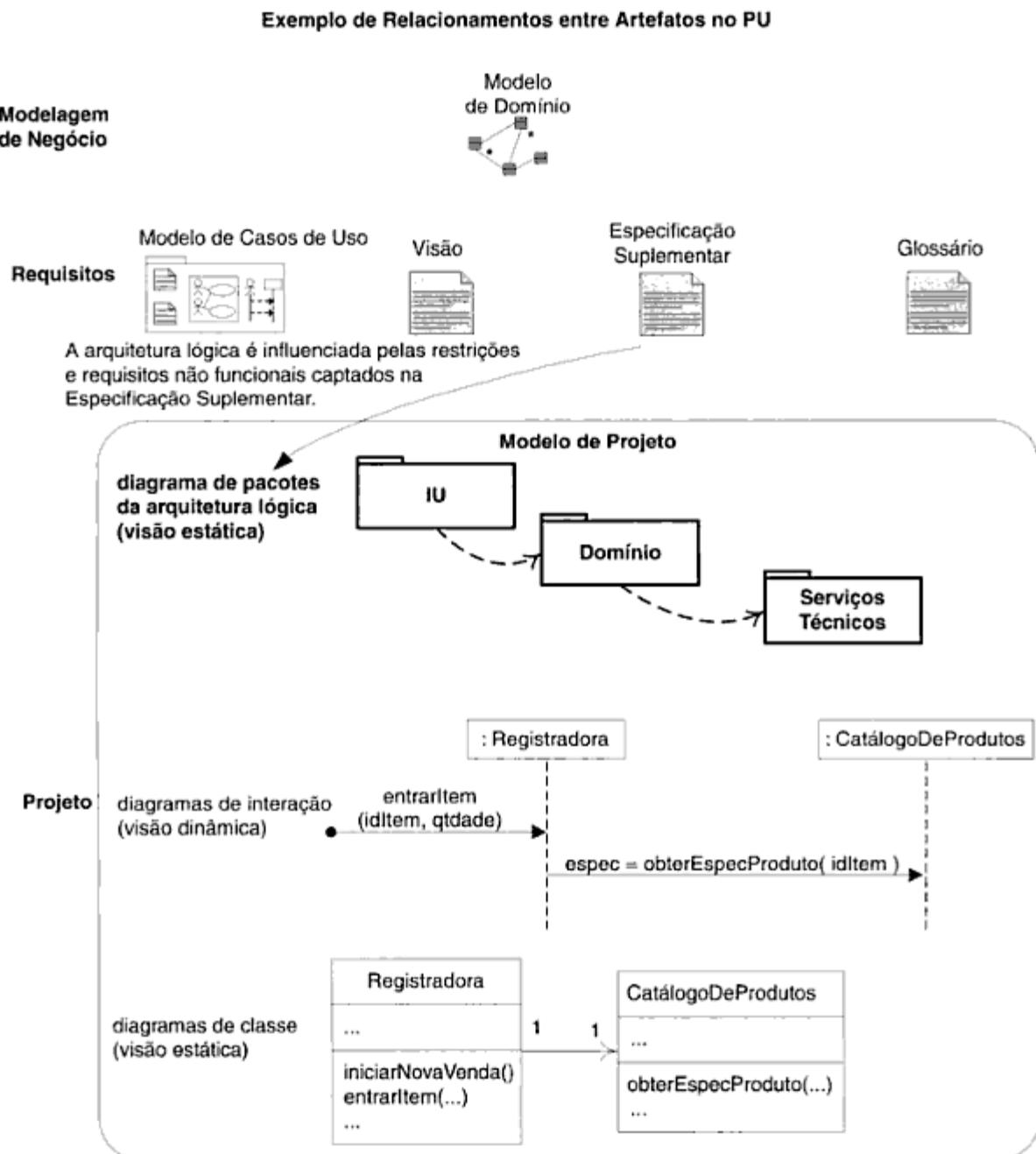


Figura 13.1 Amostra da influência de artefatos do PU.

13.1 Exemplo

A Figura 13.2 mostra uma arquitetura lógica em camadas parcial desenhada com a notação de diagrama de pacotes UML.

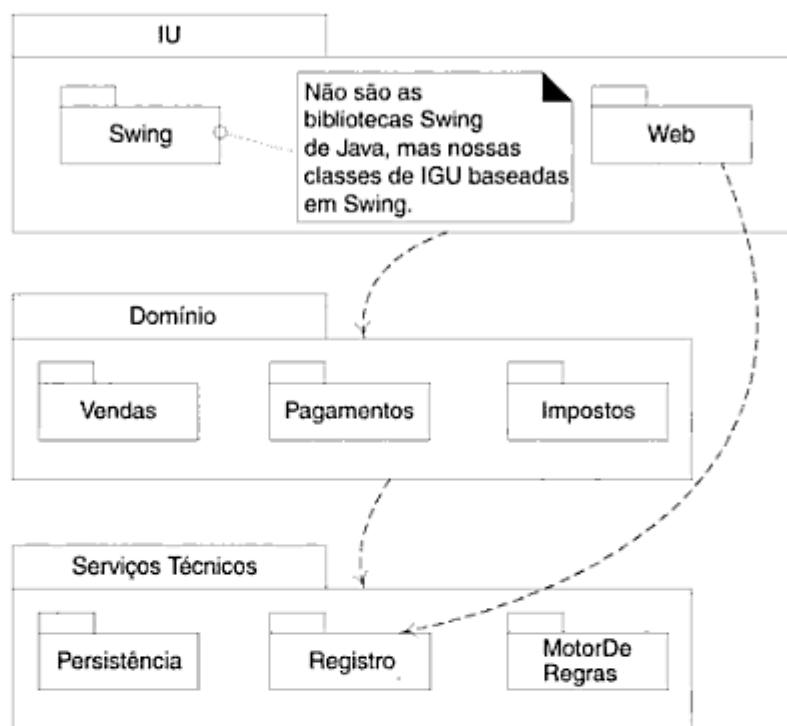


Figura 13.2 Camadas mostradas com notação de diagrama de pacotes UML.

13.2 Qual é a arquitetura lógica? E as camadas?

A **arquitetura lógica** é a organização em larga escala das classes de software em pacotes (ou espaços de nomes), subsistemas e camadas. É chamada de arquitetura *lógica* porque não há decisão sobre como esses elementos são implantados pelos diferentes processos do sistema operacional ou pelos computadores físicos em uma rede (essas decisões posteriores são parte da **arquitetura de implantação**).

Uma **camada** é um agrupamento de granularidade muito grossa de classes, pacotes ou subsistemas que tem responsabilidade coesiva sobre um tópico importante do sistema. Também, as camadas são organizadas de modo que as “mais altas” (como a camada de IU) solicitem serviços das “mais baixas” mas normalmente não vice-versa. Camadas em um sistema OO incluem:

- **Interface com o Usuário**
- **Lógica da Aplicação e Objetos do Domínio** – objetos de software representando conceitos do domínio (por exemplo, uma classe de software *Venda*) que satisfazem requisitos da aplicação, como calcular um total de venda.

- **Serviços Técnicos** – objetos de propósito geral e subsistemas que fornecem serviços técnicos de apoio, como interfaceamento com um banco de dados ou registro de erros. Esses serviços geralmente são independentes da aplicação e reusáveis entre diversos sistemas.

Em uma **arquitetura em camadas estrita**, uma camada apenas solicita serviços da camada diretamente inferior a ela. Esse projeto é comum em pilhas de protocolo de rede, mas não em sistemas de informação, que costumam ter uma **arquitetura em camadas relaxada**, na qual uma camada mais alta solicita informações de várias camadas mais baixas. Por exemplo, a camada de IU pode solicitar da sua camada de lógica da aplicação diretamente subordinada e também de elementos de uma camada de serviços técnicos mais baixa para registro, e assim por diante.

Uma arquitetura lógica não precisa ser organizada em camadas, mas isso é *muito* comum e, portanto, abordado a seguir.

13.3 Que camadas são o foco dos estudos de caso?

Estudos de caso
(pág. 69)

Para reiterar um ponto estabelecido quando os estudos de caso foram introduzidos:

Apesar da tecnologia OO poder ser aplicada em todos os níveis, esta introdução à A/POO enfoca a *camada de lógica da aplicação principal* (ou “*domínio*”), com um pouco de discussão secundária das outras camadas.

A exploração do projeto das outras camadas (por exemplo, a camada de IU) vai enfocar o projeto de sua interface com a camada de lógica da aplicação.

A discussão na pág. 69 explica, resumidamente, porque as outras camadas tendem a ser muito dependentes da tecnologia (por exemplo, muito específicas de Java ou .NET) e, em qualquer caso, as lições de projeto OO aprendidas no contexto da camada de lógica da aplicação (domínio) são aplicáveis a todas as outras camadas ou componentes.

13.4 O que é arquitetura do software?

Mencionei as arquiteturas lógica e de implantação, assim, agora é uma boa hora para introduzir uma definição de **arquitetura do software**. Eis uma delas:

Uma arquitetura é um conjunto de decisões significativas sobre a organização de um sistema de software, a seleção dos elementos estruturais e suas interfaces pelas quais o sistema é composto, juntamente com seu comportamento, como especificado nas colaborações entre esses elementos, a composição desses elementos estruturais e comportamentais em subsistemas progressivamente maiores, e o estilo arquitetural que dirige essa organização – esses elementos e suas interfaces, suas colaborações e sua composição. [BRJ99]

Independentemente da definição (e há muitas), o tema comum a todas as definições de arquitetura de software é o de que ela tem a ver com a grande escala – as Grandes Idéias nas motivações, restrições, organização, padrões, responsabilidades e conexões de um sistema (ou sistema de sistemas).

13.5 Aplicação de UML: diagramas de pacotes

Diagramas de pacotes UML freqüentemente são usados para ilustrar a arquitetura lógica de um sistema – as camadas, subsistemas, pacotes (no significado de Java), etc. Uma camada pode ser modelada como um pacote UML; por exemplo, a camada de IU modelada como um pacote denominado IU.

Um diagrama de pacotes UML fornece um modo de agrupar elementos. Um pacote UML pode agrupar qualquer coisa: classes, outros pacotes, casos de uso, etc. Aninhar pacotes é muito comum. Um pacote UML é um conceito mais geral do que simplesmente um pacote Java ou espaço de nomes .NET, assim, um pacote UML pode representar esses – e muitos outros.

O nome do pacote pode ser colocado na orelha se o pacote mostrar membros internos, ou na pasta principal, se não for o caso.

É comum desejar mostrar dependência (acoplamento) entre pacotes, de modo que os desenvolvedores possam ver o acoplamento em larga escala do sistema. A **linha de dependência UML** é usada para isso, uma linha tracejada com seta em que a seta aponta para o pacote dependente.

Um pacote UML representa um **espaço de nomes** de modo que, por exemplo, uma classe *Data* pode ser definida em dois pacotes. Se você precisa fornecer **nomes plenamente qualificados**, a notação UML é, por exemplo, *java::util::Data*, para o caso em que há um pacote externo chamado “java” com um pacote aninhado chamado “util” contendo uma classe Data.

A UML fornece notações alternativas para ilustrar pacotes aninhados externos e internos. Algumas vezes fica estranho desenhar uma caixa de pacote externo em volta do pacote interno. Alternativas são mostradas na Fig. 13.3.

Ferramentas UML: fazer engenharia reversa para obter diagramas de pacotes a partir do código

Durante o início do desenvolvimento, podemos rascunhar um diagrama de pacotes UML e depois organizar nosso código de acordo com esses rascunhos de pacotes. Com o passar do tempo, a base de código aumenta e gastamos mais tempo programando e menos modelando ou fazendo diagramas UML. Nesse ponto, um excelente uso para uma ferramenta CASE UML é fazer a engenharia reversa do código fonte e gerar um diagrama de pacotes automaticamente.

Essa prática é aperfeiçoadas se usarmos as convenções de denominação da pág. 226 sugeridas para pacotes de código.

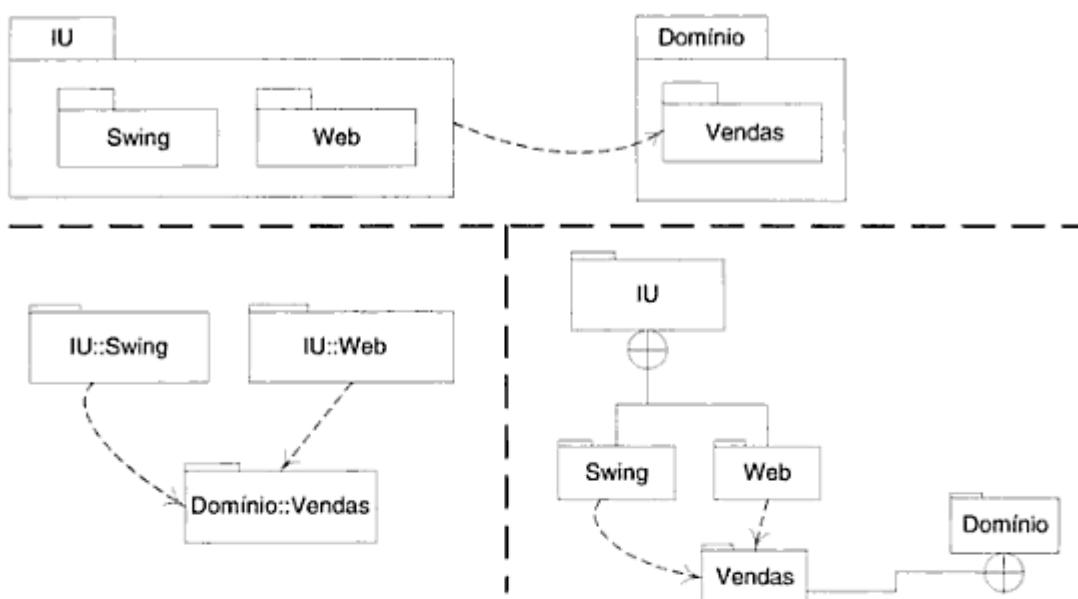


Figura 13.3 Abordagens alternativas UML para mostrar aninhamento de pacotes usando pacotes embutidos, nomes UML plenamente qualificados e o símbolo círculo-cruz.

13.6 Diretriz: projeto em camadas

As idéias essenciais de usar camadas [BMRSS96] são simples:

- Organizar a estrutura lógica de larga escala de um sistema em camadas discretas de responsabilidades distintas relacionadas, com uma separação clara e coesa de interesses, de modo que as camadas “inferiores” sejam de baixo nível e serviços gerais e camadas superiores sejam mais específicas da aplicação.
- Colaboração e acoplamento ocorrem das camadas superiores para as inferiores; um acoplamento das camadas inferiores para as superiores é evitado.

Alguns outros tópicos de projeto são abordados posteriormente, começando nas pág. 560. A idéia é descrita por meio do **Padrão de Camadas** [BMRSS96] e produz uma **arquitetura em camadas**. Ele tem sido aplicado e tem se escrito sobre ele com tanta freqüência como um padrão que o *Pattern Almanac 2000* [Rising00] lista mais de 100 padrões que são variantes do ou relacionados ao padrão de Camadas.

Usar camadas ajuda a tratar diversos problemas:

- Modificações do código fonte se espalham por todo o sistema – muitas partes do sistema são altamente acopladas.
- A lógica da aplicação está entrelaçada com a interface com o usuário, de modo que não pode ser reusada com uma interface diferente ou distribuída para outro nó de processamento.
- Serviços técnicos potencialmente gerais ou lógica de negócios estão entrelaçados com lógica mais específica da aplicação, de modo que não podem ser reusa-

dos, distribuídos para outro nó ou facilmente substituídos por uma implementação diferente.

- Existe alto acoplamento entre diferentes áreas de interesse. Assim, fica difícil dividir o trabalho com fronteiras claras entre diferentes desenvolvedores.

A finalidade e o número de camadas varia entre aplicações e domínios de aplicação (sistemas de informação, sistemas operacionais, etc.). Aplicadas a sistemas de informação, camadas típicas são ilustradas e explicadas na Figura 13.4.

A camada de *Aplicação* da Figura 13.4 é discutida nas págs. 567-568.

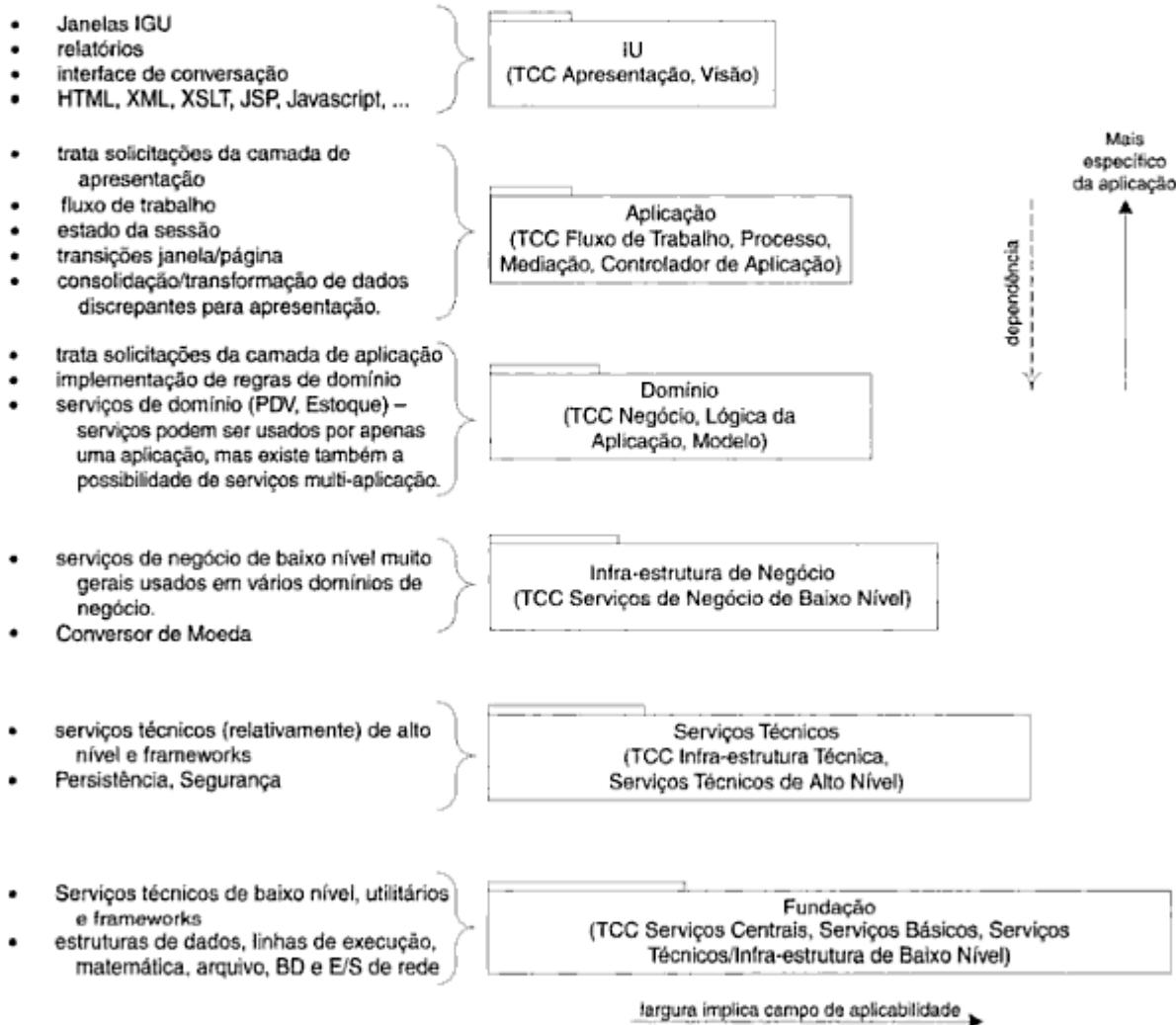


Figura 13.4 Camadas comuns na arquitetura lógica de um sistema de informação.¹

¹ A largura do pacote é usada para informar o campo de aplicabilidade nesse diagrama, mas essa não é uma prática geral da UML. TCC significa Também Conhecida Como (*Also Known As* – AKA).

Benefícios do uso de camadas

- Em geral, existe uma separação de interesses, uma separação de serviços de alto para baixo nível e uma separação de serviços de aplicação gerais para específicos. Isso reduz o acoplamento e as dependências, melhora a coesão, aumenta o potencial de reuso e aumenta a clareza.
- A complexidade relacionada é encapsulada e decomponível.
- Algumas camadas podem ser substituídas por nova implementação. Isso é geralmente impossível para as camadas de Serviço Técnico de baixo nível ou Fundação (por ex, *Java.util*), mas pode ser possível para as camadas de IU, Aplicação e Domínio.
- As camadas inferiores contêm funções reusáveis.
- Algumas camadas (principalmente de Domínio e de Serviços Técnicos) podem ser distribuídas.
- O desenvolvimento em equipes é facilitado por causa da segmentação lógica.

Diretriz: responsabilidades coesivas; mantenha uma separação de interesses

As responsabilidades dos objetos em uma camada devem estar fortemente relacionadas umas com as outras e não devem ser misturadas com responsabilidades de outras camadas. Por exemplo, objetos da camada IU devem estar focados no trabalho de IU, como a criação de janelas e dispositivos, captação de eventos do mouse e do teclado, etc. Objetos da camada de lógica da aplicação ou “domínio” devem enfocar a lógica da aplicação, por exemplo, calcular um total de venda ou impostos ou mover uma peça em um tabuleiro de jogo.

Objetos de IU não devem executar a lógica da aplicação. Por exemplo, um objeto Java Swing *JFrame* (janela) não deve conter lógica para calcular impostos ou mover uma peça de jogo. E, por outro lado, classes da lógica da aplicação não devem captar eventos de IU, mouse ou teclado. Isso violaria uma clara **separação de interesses** e a manutenção de uma **alta coesão** – princípios arquiteturais básicos.

*Alta coesão
(págs. 329-330)*

*Modelo-visão
(págs. 229-231)*

Capítulos posteriores vão explorar esses importantes princípios, além do **Princípio de Separação Modelo-Visão**, com mais detalhes.

Código: mapeamento da organização do código para camadas e pacotes UML

As linguagens OO mais populares (Java, C#, C++, Python,...) fornecem apoio a pacotes (chamados espaços de nomes em C# e C++).

Aqui está um exemplo, usando Java, para mapear pacotes UML para o código. As camadas e pacotes ilustrados na Figura 13.2 podem ser mapeados para nomes de pacotes Java como mostrado a seguir. Note que o nome da camada é usado como uma seção do nome de pacote Java.

```
// --- Camada de IU
com[minhaempresa].proxger.iu.swing
com[minhaempresa].proxger.iu.web

// --- Camada de DOMÍNIO
// pacotes específicos para o projeto ProxGer
com[minhaempresa].proxger.dominio.vendas
com[minhaempresa].proxger.dominio.pagamentos

// --- Camada de SERVIÇOS TÉCNICOS
// nossa camada de acesso de persistência (banco de dados)
// desenvolvida internamente
com[minhaempresa].servico.persistencia

// terceirizado
org.apache.log4j
org.apache.soap.rpc

// --- Camada de FUNDAÇÃO
// pacotes de fundação que nossa equipe cria
com[minhaempresa].utils
```

Note que, para apoiar reuso entre projetos, evitamos usar um qualificador específico da aplicação (“proxger”) no nome do pacote, a menos que seja necessário. Os pacotes de IU são relacionados à aplicação PDV ProxGer, assim, são qualificados com o nome da aplicação `com[minhaempresa].proxger.iu`*. Mas os utilitários que escrevemos poderiam ser compartilhados entre vários projetos, assim, o nome do pacote é `com[minhaempresa].utils` e não `com[minhaempresa].proxger.utils`.

Ferramentas UML: fazer engenharia reversa para obter diagramas de pacotes a partir do código

Como já mencionado, um excelente uso para uma ferramenta CASE UML é fazer a engenharia reversa do código fonte e gerar um diagrama de pacotes automaticamente. Essa prática é aperfeiçoada se você usa no código as convenções de denominação recomendadas. Por exemplo, incluindo o nome parcial “.iu.” em todos os pacotes da camada de IU, a ferramenta CASE UML vai automaticamente agrupar e aninhar subpacotes sob um pacote “iu” e você poderá ver a arquitetura em camadas tanto no código quanto no diagrama de pacotes.

*Definição: Camada de Domínio vs Camada da Lógica da Aplicação;
Objetos de Domínio*

Esta seção descreve um conceito simples,
mas muito importante em projeto OO!

Um sistema de software típico tem lógica da IU e lógica da aplicação, como a criação de um dispositivo de IGU e cálculos de impostos. Eis uma questão-chave:

Como projetamos a lógica da aplicação com objetos?

Poderíamos criar uma classe chamada *XYZ* e colocar nela todos os métodos para toda a lógica necessária. Isso poderia funcionar tecnicamente (apesar de ser um pesadelo para entender e manter), mas não é a abordagem recomendada pelo raciocínio OO.

Então, qual é a abordagem recomendada? Resposta: criar objetos de software com nomes e informação semelhantes ao domínio do mundo real e atribuir responsabilidades da lógica da aplicação a eles. Por exemplo, no mundo real de PDV há vendas e pagamentos. Assim, no software criamos as classes *Venda* e *Pagamento* e lhes damos responsabilidades da lógica da aplicação. Essa espécie de objeto de software é chamada de **objeto de domínio**. Representa uma coisa no espaço do domínio do problema e tem lógica relacionada de aplicação ou de negócio, por exemplo, um objeto *Venda* sendo capaz de calcular o seu total.

Projetar objetos desse modo implica que a camada de lógica da aplicação seja mais precisamente chamada de **camada de domínio** da arquitetura – a camada que contém objetos de domínio para tratar do trabalho da lógica da aplicação.

Qual é o relacionamento entre a camada de domínio e o modelo de domínio?

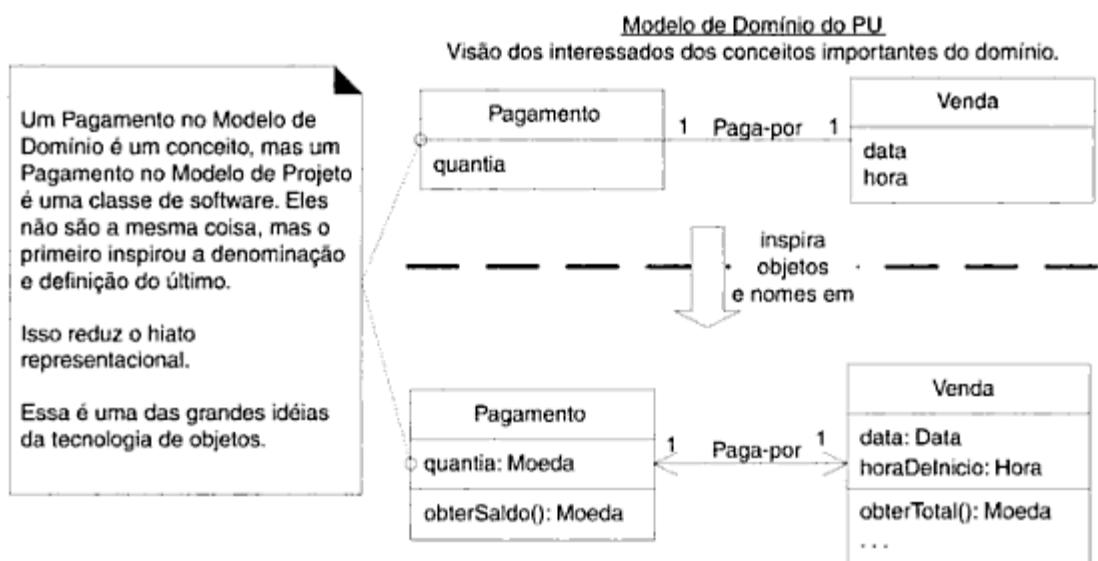
Esse é um outro ponto-chave: há um relacionamento entre o modelo de domínio e a camada de domínio. Olhamos para o modelo de domínio (que é a visualização dos conceitos importantes do domínio) para ter inspiração quanto aos nomes das classes na camada de domínio. Veja a Figura 13.5.

A camada de domínio é parte do software e o modelo de domínio é parte da análise da perspectiva-conceitual – eles não são a mesma coisa. Mas, ao criar a camada de domínio com inspiração no modelo de domínio, alcançamos um **baixo hiato representacional** entre o domínio do mundo real e o nosso projeto de software. Por exemplo, uma *Venda* no Modelo de Domínio do PU ajuda a nos inspirar quanto à criação de uma classe de software *Venda* na camada de domínio do Modelo de Projeto do PU.

Definição: fileiras, camadas e partições

A noção original de uma **fileira** (tier) em arquitetura era uma camada lógica, não um nó físico, mas a palavra tornou-se amplamente usada para um nó de processamento físico (ou um grupo de nós), por exemplo, o “nó cliente” (o computador cliente). Este livro vai evitar o termo em prol da clareza, mas tenha isso em mente quando estudar a literatura sobre arquitetura.

As **camadas** de uma arquitetura são usadas para representar as fatias verticais, enquanto que **partições** representam uma divisão horizontal de subsistemas relativamente paralelos de uma camada. Por exemplo, a camada de *Serviços Técnicos* pode ser dividida em partições como *Segurança* e *Relatórios* (Figura 13.6).



Camada de Domínio da arquitetura no Modelo de Projeto do PU
O desenvolvedor orientado a objetos foi inspirado pelo domínio do mundo real na criação de classes de software.

Assim, o hiato representacional entre como os interessados concebem o domínio e sua representação no software foi diminuído.

Figura 13.5 Relacionamento entre a camada de domínio e o modelo de domínio.

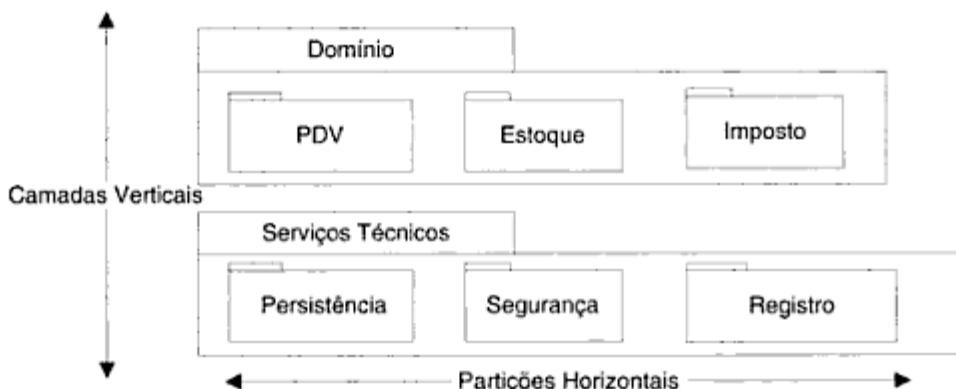


Figura 13.6 Camadas e Partições.

Diretriz: não mostre recursos externos como a camada inferior

A maioria dos sistemas depende de recursos ou serviços externos, tais como de um banco de dados de estoque em MySQL e um serviço de denominação e diretório LD-PA Novell. Esses são componentes *físicos* de implementação, não uma camada na arquitetura lógica.

Mostrar recursos externos, como um banco de dados específico em uma camada “abaixo” da camada de Fundação (por exemplo) mistura a visão lógica e a visão de implantação da arquitetura.

Uma nova edição do número 1 entre os livros de introdução à análise orientada a objetos e ao projeto com UML

Totalmente atualizada para a UML 2 e as práticas iterativas/ágiles mais recentes

Inclui um novo estudo de caso que ilustra os conceitos-chave do livro

Utilizando UML e Padrões é o livro número 1 – tanto na prática profissional quanto acadêmica – de introdução ao “pensamento orientado a objetos” e à utilização desse conhecimento em situações reais. Depois de duas aclamadas edições, Craig Larman atualizou esta obra para abordar o novo padrão UML 2 e para auxiliar no projeto de objetos, promover práticas de modelagem iterativas, ágeis e de alto impacto.

Esta edição auxilia desenvolvedores e estudantes a dominar a análise e o projeto orientado a objetos por meio de três iterações de dois estudos de caso coesos e completos, que gradualmente introduzem as técnicas fundamentais de A/POO, a notação UML e as melhores práticas. Você não apenas aprenderá o que são diagramas UML – saberá como aplicá-los no contexto do desenvolvimento de software OO.

Calcado em sua experiência como professor e consultor, Larman ajuda a entender conceitos como requisitos evolutivos e casos de uso, modelagem de objetos do domínio, projeto guiado por responsabilidades, projeto OO essencial, arquitetura em camadas, padrões de projeto da “Gangue dos Quatro”, GRASP, métodos iterativos, uma abordagem ágil ao Processo Unificado e muito mais.

Esta edição inclui:

- ▶ Aprimoramento do ensino da A/POO por meio de estudos de caso que demonstram os princípios-chave de OO e padrões, ao mesmo tempo em que aplicam a UML
- ▶ Uma nova abordagem de UML 2, Modelagem Ágil, Desenvolvimento Dirigido por Teste e refatoração
- ▶ Novas dicas sobre como combinar desenvolvimento iterativo e evolutivo e A/POO
- ▶ Atualizações para facilitar o estudo, incluindo novos recursos de aprendizagem e gráficos
- ▶ Novos recursos de ensino para professores universitários
- ▶ Diretrizes sobre a aplicação do PU com um espírito leve, ágil, complementar a outros métodos iterativos, como XP e Scrum
- ▶ Técnicas para a aplicação da UML na documentação de arquiteturas
- ▶ Um novo capítulo sobre requisitos evolutivos e muito mais

Utilizando UML e Padrões, 3^a edição, é uma introdução lúcida e prática a como pensar e projetar com objetos – e criar sistemas bem arquitetados, robustos e sustentáveis.

“Esta edição contém a precisão e profundidade habitual de Larman. É um livro muito bom que está ainda melhor.”

Alistair Cockburn, autor de *Escrevendo Casos de Uso Eficazes* e *Surviving OO Projects*

“Poucas pessoas têm jeito para explicar as coisas. Muito poucas conhecem análise e projeto orientados a objetos. Craig Larman reúne as duas qualidades.”

John Vlissides, autor de *Padrões de Projeto* e *Pattern Hatching*

ISBN 978-85-60031-52-8



artmed®
EDITORAS
RESPEITO PELO CONHECIMENTO



www.bookman.com.br

Material com direitos autorais