

Anthony Sintes

## Aprenda

os principais conceitos e  
práticas da programação  
orientada a objetos em  
apenas 21 dias

## Aplique

seus conhecimentos  
ao mundo real

# Aprenda

# Programação Orientada a Objetos



MAKRON  
*Books*

em **21** dias

**PÁGINA EM BRANCO**

# Aprenda Programação Orientada a Objetos em 21 Dias

**Tony Sintes**

*Tradução*

**João Eduardo Nóbrega Tortello**

*Revisão Técnica*

**Ana Fernanda Gomes Ascencio**

Bacharel em Ciência da Computação pela PUC-SP

Especialista em Sistema de Informação pela UFSCAR e mestre em computação pela UFRGS



São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha  
Guatemala México Peru Porto Rico Venezuela

© 2002 by Pearson Education do Brasil

Título Original:

Object Oriented Programming in 21 Dias

© 2002 by Sams Publishing

Todos os direitos reservados

Editora: Gisélia do Carmo Costa

Gerente de Produção: Silas Camargo

Produtora Editorial: Sandra Cristina Pedri

Revisão: Jorge Abdalla Neto

Capa: Marcelo da Silva Françozo, sobre o projeto original

*Editoração Eletrônica: ERJ Informática Ltda.*

#### **Dados de Catalogação na Publicação**

---

Sintes, Tony

Aprenda Programação Orientada a Objetos em 21 Dias

Tradução: João Eduardo Nóbrega Tortello

Revisão Técnica: Ana Fernanda Gomes Ascencio

São Paulo: Pearson Education do Brasil, 2002

Título Original: Object Oriented  
Programming in 21 Dias

---

ISBN: 85.346.1461-X

---

#### **Índice para Catálogo Sistemático**

##### **1. Programação Orientada a Objetos**

2002

Direitos exclusivos para a língua portuguesa cedidos à

Pearson Education do Brasil,

uma empresa do grupo Pearson Education

Av. Ermano Marchetti, 1435

CEP 05038-001 – Lapa – São Paulo – SP

Tel: (11) 3613-1222 Fax: (11) 3611-0444

e-mail: vendas@pearsoned.com

# Sobre o autor

TONY SINTES trabalha com tecnologias orientadas a objetos há sete anos. Nesse tempo, Tony fez parte de muitos trabalhos de desenvolvimento orientados a objetos de larga escala. Atualmente, Tony trabalha na First Class Consulting, uma empresa que fundou para ajudar as grandes empresas a integrar seus diversos sistemas em uma estrutura unificada. Antes de iniciar a First Class Consulting, Tony trabalhou na BroadVision como consultor sênior, onde ajudou a construir alguns dos maiores sites Web do mundo. Atualmente, as principais responsabilidades de Tony são como arquiteto, líder técnico e mentor da equipe, ajudando a construir as habilidades em desenvolvedores menos experientes.

Tony é um autor técnico amplamente reconhecido, cujos trabalhos têm aparecido na *JavaWorld*, *Dr. Dobb's Journal*, *LinuxWorld*, *JavaOne Today* e *Silicon Prairie*, onde é co-autor de uma coluna mensal altamente respeitada sobre programação orientada a objetos. Atualmente, Tony escreve a coluna mensal de perguntas e respostas da *JavaWorld*. Você pode entrar em contato com ele no endereço [styoop@firstclassconsulting.net](mailto:styoop@firstclassconsulting.net).

# Agradecimentos

Escrever um livro é um processo como nenhum outro. A quantidade de pessoas que colaboram em um livro, e que são necessárias para produzir a cópia final que você está lendo agora, é simplesmente espantoso. Gostaria de estender minha gratidão à equipe inteira de editoriais da Sams. Sem seu trabalho árduo, este livro simplesmente não existiria.

Pelo nome, gostaria de agradecer a Michael Stephens, Carol Ackerman, Tiffany Taylor e George Nedeff. A liderança e toques sutis de Carol foram o que realmente fizeram este livro prosseguir até sua conclusão. A capacidade de Tiffany de estruturar e encadear o material técnico, clara e concisamente, é simplesmente espantosa. Não apenas as edições de Tiffany tornaram este livro mais legível, mas acho que seu trabalho me ensinou algumas lições valiosas sobre redação técnica. Também gostaria de agradecer a William Brown. William entrou em contato comigo a respeito do projeto STY OOP, no início de agosto de 2000. Confiar tal projeto a um autor relativamente desconhecido era arriscado e agradeço a William por me dar a chance de escrever este livro.

Agradeço especialmente aos editores técnicos, Mark Cashman e Richard Baldwin, que garantiram que o material apresentado fosse tecnicamente bom. Agradeço a vocês pela atuação técnica.

Aos meus colegas, muito obrigado. Gostaria de estender os agradecimentos especiais a David Kim e Michael Han. Eu comecei este livro enquanto estava na BroadVision e gostaria de agradecer a David Kim por me permitir aquelas férias de pânico, quando os prazos de entrega começaram a se aproximar. Também gostaria de agradecer a Michael Han, por suas idéias técnicas e por escrever o apêndice sobre Java deste livro.

Por último, mas não menos importante, tenho o privilégio de agradecer à minha maravilhosa esposa, Amy, por seu apoio firme, revisão e paciência. Agradeço ainda à minha família e amigos, que ofereceram seu apoio e que ouviram minhas reclamações.

# Diga-nos o que você acha!

Como leitor deste livro, *você* é nosso crítico e colaborador mais importante. Valorizamos sua opinião e queremos saber o que estamos fazendo corretamente, o que poderíamos fazer melhor, sobre quais áreas você gostaria de nos ver publicando e qualquer outra sugestão importante que deseje passar para nós.

Receberemos com satisfação seus comentários. Você pode enviar um fax, e-mail ou escrever diretamente, para que possamos saber o que gostou ou não neste livro — assim como o que podemos fazer para tornar nossos livros melhores.

Por favor, entenda que não podemos ajudá-lo em problemas técnicos relacionados ao assunto deste livro e que, devido ao grande volume de correspondência que recebemos, não possamos responder a todas as mensagens.

Quando você escrever, certifique-se de incluir o título e o autor deste livro, assim como seu nome e número de telefone ou fax. Examinaremos seus comentários cuidadosamente e os compartilharemos com os autores e editores que trabalharam no livro.

Fax: (11) 3611-9686

Fone: (11) 3613-1213

Endereço eletrônico: [clientes@makron.com.br](mailto:clientes@makron.com.br)

Endereço postal: Pearson Education do Brasil Ltda

Rua Emílio Goeldi, 747 – Lapa

São Paulo – SP – CEP: 05065-110

PÁGINA EM BRANCO

# Sumário

<b>Introdução</b>	<b>XXIII</b>
Sobre os exemplos .....	XXIV
O que você precisa saber para usar este livro .....	XXIV
<b>SEMANA 1 Definindo OO</b>	<b>1</b>
<b>Dia 1 Introdução à programação orientada a objetos</b>	<b>3</b>
Programação orientada a objetos em um contexto histórico .....	4
Precursors da POO .....	4
Programação orientada a objetos .....	6
Uma estratégia de POO para software usando objetos .....	6
O que é uma classe? .....	8
Reunindo tudo: classes e objetos .....	9
Fazendo os objetos trabalhar .....	11
Relacionamentos de objeto .....	13
Como a programação orientada a objetos fundamenta o passado .....	14
Vantagens e objetivos da OO .....	14
Natural .....	15
Confiável .....	15
Reutilizável .....	15
Manutenível .....	16
Extensível .....	16
Oportuno .....	16
Armadilhas .....	16
Armadilha 1: pensar na POO simplesmente como uma linguagem .....	17
Armadilha 2: medo da reutilização .....	17
Armadilha 3: pensar na OO como uma solução para tudo .....	17
Armadilha 4: programação egoísta .....	18
A próxima semana .....	18
Resumo .....	18
Perguntas e respostas .....	19
Workshop .....	19
Teste .....	19
Exercícios .....	20
<b>Dia 2 Encapsulamento: aprenda a manter os detalhes consigo mesmo</b>	<b>21</b>
Encapsulamento: o primeiro pilar .....	22
Um exemplo de interface e implementação .....	24

Público, privado e protegido .....	25
Por que você deve encapsular? .....	25
Abstração: aprendendo a pensar e programar de forma abstrata .....	26
O que é abstração? .....	26
Dois exemplos de abstração .....	27
Abstração eficaz .....	28
Guardando seus segredos através da ocultação da implementação .....	29
Protegendo seu objeto através do TAD (Abstract Data Type – Tipo Abstrato de Dados) .....	30
O que é um tipo? .....	30
Um exemplo de TAD .....	33
Protegendo outros de seus segredos, através da ocultação da implementação ..	34
Um exemplo real de ocultação da implementação .....	36
Divisão da responsabilidade: preocupando-se com seu próprio negócio .....	37
Dicas e armadilhas do encapsulamento .....	41
Dicas e armadilhas da abstração .....	41
Dicas e armadilhas do TAD .....	43
Dicas da ocultação da implementação .....	43
Como o encapsulamento atende os objetivos da programação orientada a objetos .....	44
Advertências .....	45
Resumo .....	45
Perguntas e respostas .....	45
Workshop .....	46
Teste .....	46
Exercícios .....	47
<b>Dia 3 Encapsulamento: hora de escrever algum código</b>	<b>49</b>
Laboratório 1: configurando o ambiente Java .....	49
Exposição do problema .....	50
Laboratório 2: classes básicas .....	50
Exposição do problema .....	53
Soluções e discussão .....	54
Laboratório 3: o encapsulamento .....	56
Exposição do problema .....	57
Soluções e discussão .....	57
Laboratório 4: estudo de caso — os pacotes de primitivas Java (opcional) .....	62
Exposição do problema .....	66
Soluções e discussão .....	66
Perguntas e respostas .....	67
Workshop .....	68
Teste .....	68
Exercícios .....	69

<b>Dia 4 Herança: obtendo algo para nada</b>	<b>71</b>
O que é herança? .....	71
Por que herança? .....	74
“É um” <i>versus</i> “tem um”: aprendendo quando usar herança .....	75
Aprendendo a navegar na teia emaranhada da herança .....	77
Mecânica da herança .....	79
Métodos e atributos sobrepostos .....	81
Novos métodos e atributos .....	84
Métodos e atributos recursivos .....	84
Tipos de herança .....	84
Herança para implementação .....	85
Problemas da herança da implementação .....	85
Herança para diferença .....	86
Especialização .....	87
Herança para substituição de tipo .....	90
Dicas para a herança eficaz .....	92
Resumo .....	94
Como a herança atende aos objetivos da OO .....	94
Perguntas e respostas .....	96
Workshop .....	96
Teste .....	97
Exercícios .....	97
<b>Dia 5 Herança: hora de escrever algum código</b>	<b>99</b>
Laboratório 1: herança simples .....	99
Exposição do problema .....	100
Soluções e discussão .....	101
Laboratório 2: usando classes abstratas para herança planejada .....	102
Exposição do problema .....	105
Soluções e discussão .....	105
Laboratório 3: conta em banco — praticando a herança simples .....	107
Uma conta genérica .....	107
A conta poupança .....	107
Uma conta com vencimento programado .....	107
Conta com cheques .....	107
Conta com cheque especial .....	108
Exposição do problema .....	108
Exposição estendida do problema .....	110
Soluções e discussão .....	111
Laboratório 4: estudo de caso — “é um”, “tem um” e java.util.Stack .....	117
Exposição do problema .....	118
Soluções e discussão .....	118
Resumo .....	119
Perguntas e respostas .....	119

Workshop .....	120
Teste .....	120
Exercícios .....	120
<b>Dia 6 Polimorfismo: aprendendo a prever o futuro</b>	<b>121</b>
Polimorfismo .....	122
Polimorfismo de inclusão .....	126
Polimorfismo paramétrico .....	131
Métodos paramétricos .....	131
Tipos paramétricos .....	133
Sobreposição .....	134
Sobrecarga .....	135
Conversão .....	137
Polimorfismo eficaz .....	137
Armadilhas polimórficas .....	139
Armadilha 1: mover comportamentos para cima na hierarquia .....	140
Armadilha 2: sobrecarga de desempenho .....	140
Armadilha 3: vendas .....	141
Advertências .....	142
Como o polimorfismo atende os objetivos da OO .....	142
Resumo .....	143
Perguntas e respostas .....	144
Workshop .....	145
Teste .....	145
Exercícios .....	145
<b>Dia 7 Polimorfismo: hora de escrever algum código</b>	<b>147</b>
Laboratório 1: aplicando polimorfismo .....	147
Exposição do problema .....	154
Soluções e discussão .....	155
Laboratório 2: conta de banco — aplicando polimorfismo em um exemplo conhecido .....	156
Exposição do problema .....	157
Soluções e discussão .....	159
Laboratório 3: conta de banco — usando polimorfismo para escrever código à prova do futuro .....	160
Exposição do problema .....	162
Soluções e discussão .....	163
Laboratório 4: estudo de caso — estruturas condicionais Java e polimorfismo .....	165
Corrigindo uma estrutura condicional .....	167
Exposição do problema .....	169
Soluções e discussão .....	171
Resumo .....	172

Perguntas e respostas .....	173
Workshop .....	173
Teste .....	173
Exercícios .....	173
<b>SEMANA 1 Em revisão</b>	<b>174</b>
<b>SEMANA 2 Aprendendo a aplicar OO</b>	<b>175</b>
<b>Dia 8 Introdução à UML</b>	<b>177</b>
Introdução à Unified Modeling Language .....	177
Modelando suas classes .....	179
Notação básica de classe .....	179
Notação avançada de classe .....	181
Modelando suas classes de acordo com seus objetivos .....	181
Modelando um relacionamento de classe .....	183
Dependência .....	183
Associação .....	184
Agregação .....	186
Composição .....	187
Generalização .....	188
Reunindo tudo .....	189
Resumo .....	190
Perguntas e respostas .....	191
Workshop .....	191
Teste .....	191
Exercícios .....	192
<b>Dia 9 Introdução à AOO (Análise Orientada a Objetos)</b>	<b>193</b>
O processo de desenvolvimento de software .....	194
O processo iterativo .....	195
Uma metodologia de alto nível .....	197
AOO (Análise Orientada a Objetos) .....	198
Usando casos de estudo para descobrir o uso do sistema .....	199
Crie uma lista preliminar de casos de uso .....	201
Construindo o modelo de domínio .....	213
E agora? .....	214
Resumo .....	215
Perguntas e respostas .....	216
Workshop .....	216
Teste .....	216
Exercícios .....	217
<b>Dia 10 Introdução ao POO (Projeto Orientado a Objetos)</b>	<b>219</b>
POO (Projeto Orientado a Objetos) .....	220

Como você aplica POO (Projeto Orientado a Objeto)?	221
Passo 1: gere uma lista inicial de objetos	222
Passo 2: refine as responsabilidades de seus objetos	223
Passo 3: desenvolva os pontos de interação	231
Passo 4: detalhe os relacionamentos entre os objetos	232
Passo 5: construa seu modelo	233
Resumo	234
Perguntas e respostas	234
Workshop	235
Teste	235
Exercícios	236
<b>Dia 11 Reutilizando projetos através de padrões de projeto</b>	<b>237</b>
Reutilização de projeto	238
Padrões de projeto	238
O nome do padrão	239
O problema	239
A solução	239
As consequências	239
Realidades do padrão	240
Padrões por exemplo	240
O padrão Adapter	241
O padrão Proxy	245
O padrão Iterator	247
Apossando-se de um padrão	254
Resumo	255
Perguntas e respostas	255
Workshop	256
Teste	256
Exercícios	256
Respostas do teste	259
Respostas dos exercícios	260
<b>Dia 12 Padrões avançados de projeto</b>	<b>263</b>
Mais padrões por exemplo	263
O padrão Abstract Factory	264
O padrão Singleton	269
O padrão Typesafe Enum	275
Armadilhas do padrão	280
Resumo	281
Perguntas e respostas	281
Workshop	282
Teste	282
Exercícios	282

Respostas do teste .....	284
Respostas dos exercícios .....	285
<b>Dia 13 OO e programação da interface com o usuário</b>	<b>289</b>
POO e a interface com o usuário .....	289
A importância das UIs desacopladas .....	290
Como desacoplar a UI usando o padrão Model View Controller .....	293
O modelo .....	294
O modo de visualização .....	297
O controlador .....	301
Problemas com o MVC .....	303
Uma ênfase nos dados .....	304
Acoplamento forte .....	304
Ineficiência .....	305
Resumo .....	305
Perguntas e respostas .....	305
Workshop .....	306
Teste .....	307
Exercícios .....	307
<b>Dia 14 Construindo software confiável através de testes</b>	<b>313</b>
Testando software OO .....	314
Testes e o processo de desenvolvimento de software iterativo .....	314
Formas de teste .....	317
Teste de unidade .....	317
Teste de integração .....	318
Teste de sistema .....	318
Teste de regressão .....	319
Um guia para escrever código confiável .....	319
Combinando desenvolvimento e teste .....	319
Escrevendo código excepcional .....	335
Escrevendo documentação eficaz .....	336
Resumo .....	339
Perguntas e respostas .....	339
Workshop .....	340
Teste .....	341
Exercícios .....	341
<b>SEMANA 2 Em revisão</b>	<b>342</b>
<b>SEMANA 3 Reunindo tudo: um projeto OO completo</b>	<b>345</b>
<b>Dia 15 Aprendendo a combinar teoria e processo</b>	<b>347</b>
Jogo Vinte-e-um .....	347
Por quê vinte-e-um? .....	348

Declaração da visão .....	348
Requisitos de sobreposição .....	349
Análise inicial do jogo vinte-e-um .....	349
As regras do jogo vinte-e-um .....	350
Criando uma lista preliminar de casos de uso .....	353
Planejando as iterações .....	353
Iteração 1: jogo básico .....	354
Iteração 2: regras .....	355
Iteração 3: aposta .....	355
Iteração 4: interface com o usuário .....	355
Iteração 1: jogo básico .....	356
Análise do jogo vinte-e-um .....	356
Projeto do jogo vinte-e-um .....	360
A implementação .....	365
Resumo .....	380
Perguntas e respostas .....	381
Workshop .....	381
Teste .....	381
Exercícios .....	381
<b>Dia 16 Iteração 2 do jogo vinte-e-um: adicionando regras</b>	<b>383</b>
Regras do jogo vinte-e-um .....	383
Análise das regras .....	384
Projeto das regras .....	388
Implementação das regras .....	395
Teste .....	409
Resumo .....	409
Perguntas e respostas .....	410
Workshop .....	410
Teste .....	411
Exercícios .....	411
<b>Dia 17 Iteração 3 do jogo vinte-e-um: adicionando aposta</b>	<b>413</b>
Aposta no jogo vinte-e-um .....	413
Análise da aposta .....	414
Projeto da aposta .....	417
Implementação da aposta .....	420
A implementação de Bank (Banco) .....	421
Um pequeno teste: um objeto falsificado .....	427
Resumo .....	428
Perguntas e respostas .....	429
Workshop .....	429
Teste .....	429
Exercícios .....	429

<b>Dia 18 Iteração 4 do jogo vinte-e-um: adicionando uma GUI</b>	<b>431</b>
Apresentação do jogo vinte-e-um ······	431
Otimizações da linha de comando ······	432
Análise da GUI do jogo vinte-e-um ······	433
Casos de uso da GUI ······	433
Modelos visuais de GUI ······	436
Projeto da GUI do jogo vinte-e-um ······	437
Cartões CRC da GUI ······	437
Estrutura da GUI ······	438
Refazendo ······	439
Diagrama de classes da GUI ······	440
Implementação da GUI do jogo vinte-e-um ······	440
Implementando VCard, VDeck e CardView ······	440
Implementando PlayerView ······	444
Implementando OptionView e OptionViewController ······	445
Implementando GUIPlayer ······	445
Reunindo tudo com BlackjackGUI ······	448
Resumo ······	449
Perguntas e respostas ······	450
Workshop ······	450
Teste ······	450
Exercícios ······	451
<b>Dia 19 Aplicando uma alternativa ao MVC</b>	<b>453</b>
Uma GUI alternativa do jogo vinte-e-um ······	453
As camadas do PAC ······	454
A filosofia do PAC ······	454
Quando usar o padrão de projeto PAC ······	455
Analizando a GUI PAC do jogo vinte-e-um ······	455
Projetando a GUI PAC do jogo vinte-e-um ······	455
Identificando os componentes da camada de apresentação ······	456
Projetando os componentes da camada de abstração ······	457
Projetando a camada de controle ······	458
Usando o padrão Factory para evitar erros comuns ······	458
Implementando a GUI PAC do jogo vinte-e-um ······	460
Implementando VCard e VHand ······	460
Implementando VBettingPlayer ······	462
Implementando VBlackjackDealer ······	464
Implementando GUIPlayer ······	465
Reunindo tudo com o Controle ······	465
Resumo ······	468
Perguntas e respostas ······	468
Workshop ······	468

Teste .....	468
Exercícios .....	469
<b>Dia 20 Divertindo-se com o jogo vinte-e-um</b>	<b>471</b>
Divertindo-se com o polimorfismo .....	471
Criando um jogador .....	471
O jogador seguro .....	472
Adicionando SafePlayer na GUI .....	472
Aperfeiçoamento .....	473
POO e simulações .....	474
Os jogadores do jogo vinte-e-um .....	474
Resumo .....	479
Perguntas e respostas .....	480
Workshop .....	480
Teste .....	480
Exercícios .....	480
<b>Dia 21 O último quilômetro</b>	<b>483</b>
Amarrando as pontas .....	483
Refazendo o projeto do jogo vinte-e-um para reutilização em outros sistemas .....	484
Identificando as vantagens que a POO trouxe para o sistema do jogo vinte-e-um .....	489
Realidades do setor e POO .....	491
Resumo .....	491
Perguntas e respostas .....	491
Workshop .....	492
Teste .....	492
Exercícios .....	492
<b>SEMANA 3 Em revisão</b>	<b>493</b>
<b>Apêndices</b>	<b>495</b>
<b>Apêndice A Respostas</b>	<b>497</b>
Dia 1 Respostas do teste .....	497
Respostas do teste .....	497
Dia 2 Respostas do teste e dos exercícios .....	499
Respostas do teste .....	499
Respostas dos exercícios .....	501
Dia 3 Respostas do teste e dos exercícios .....	501
Respostas do teste .....	501
Respostas dos exercícios .....	503
Dia 4 Respostas do teste e dos exercícios .....	505
Respostas do teste .....	505

Respostas dos exercícios .....	507
Dia 5 Respostas do teste .....	508
Respostas do teste .....	508
Dia 6 Respostas do teste e dos exercícios .....	508
Respostas do teste .....	508
Respostas dos exercícios .....	510
Dia 7 Respostas do teste .....	511
Respostas do teste .....	511
Dia 8 Respostas do teste e dos exercícios .....	512
Respostas do teste .....	512
Respostas dos exercícios .....	513
Dia 9 Respostas do teste e dos exercícios .....	515
Respostas do teste .....	515
Respostas dos exercícios .....	517
Dia 10 Respostas do teste e dos exercícios .....	517
Respostas do teste .....	517
Respostas dos exercícios .....	519
Dia 11 Respostas do teste e dos exercícios .....	520
Respostas do teste .....	520
Respostas dos exercícios .....	521
Dia 12 Respostas do teste e dos exercícios .....	523
Respostas do teste .....	523
Respostas dos exercícios .....	524
Dia 13 Respostas do teste e dos exercícios .....	528
Respostas do teste .....	528
Respostas dos exercícios .....	529
Dia 14 Respostas do teste e dos exercícios .....	531
Respostas do teste .....	531
Respostas dos exercícios .....	532
Dia 15 Respostas do teste e dos exercícios .....	533
Respostas do teste .....	533
Respostas dos exercícios .....	533
Dia 16 Respostas do teste e dos exercícios .....	534
Respostas do teste .....	534
Respostas dos exercícios .....	534
Dia 17 Respostas do teste e dos exercícios .....	536
Respostas do teste .....	536
Respostas dos exercícios .....	536
Dia 18 Respostas do teste e dos exercícios .....	540
Respostas do teste .....	540
Respostas dos exercícios .....	540
Dia 19 Respostas do teste e dos exercícios .....	543
Respostas do teste .....	543

Respostas dos exercícios .....	544
Dia 20 Respostas do teste e dos exercícios .....	548
Respostas do teste .....	548
Respostas dos exercícios .....	548
Dia 21 Respostas do teste e dos exercícios .....	553
Respostas do teste .....	553
Respostas dos exercícios .....	554
<b>Apêndice B Resumo do Java</b>	<b>555</b>
O Java Developer's Kit: J2SE 1.3 SDK .....	555
Configuração do ambiente de desenvolvimento .....	556
Panorama das ferramentas do SDK .....	557
Compilador Java: javac .....	557
Interpretador Java: java .....	558
Utilitário de compactação de arquivos Java: jar .....	558
Documentação Java e o gerador de documentação: javadoc .....	559
Cercadinho Java: seu primeiro programa Java .....	560
Compilando e executando .....	561
Criando um arquivo .jar .....	562
Gerando javadoc .....	563
Mecânica da linguagem Java .....	564
Classe Java simples .....	564
Tipo de Dados .....	565
Variáveis .....	566
Constantes .....	568
Operadores .....	568
Estruturas condicionais .....	570
Laços ou estruturas de repetição .....	571
Classes e interfaces — blocos de construção da linguagem Java .....	571
Usando classes já existentes .....	572
Criando suas próprias classes .....	572
Interfaces .....	575
Classes internas e classes internas anônimas .....	577
Resumo .....	579
<b>Apêndice C Referência da UML</b>	<b>581</b>
Referência da UML .....	581
Classes .....	581
Objeto .....	581
Visibilidade .....	581
Classes e métodos abstratos .....	582
Notas .....	582
Estereótipos .....	583
Relacionamentos .....	583

---

Dependência .....	583
Associação .....	584
Papéis .....	584
Multiplicidade .....	584
Agregação .....	585
Composição .....	585
Generalização .....	585
Diagramas de interação .....	586
Diagramas de colaboração .....	586
Diagramas de seqüência .....	586
<b>Apêndice D Bibliografia selecionada</b>	<b>587</b>
Análise, projeto e metodologias .....	587
Programação com C++ .....	588
Padrões de projeto .....	588
Princípios e teoria geral da OO .....	588
Teoria “Hard Core” (mas não deixe isso assustá-lo!) .....	588
Programação com Java .....	589
Miscelânea .....	589
Smalltalk .....	589
Teste .....	589
<b>Apêndice E Listagens do código do jogo vinte-e-um</b>	<b>591</b>
blackjack.core .....	592
blackjack.core.threaded .....	619
blackjack.exe .....	621
blackjack.players .....	627
blackjack.ui .....	635
blackjack.ui.mvc .....	636
blackjack.ui.pac .....	649
<b>Índice Remissivo</b>	<b>669</b>

**PÁGINA EM BRANCO**

# Introdução

Este livro adota uma estratégia prática para ensinar programação orientada a objetos (POO). Em vez de ensinar POO em um nível acadêmico, este livro apresenta lições e exemplos acessíveis e amigáveis, para permitir que você comece a aplicar POO imediatamente. Em vez de tentar ensinar cada detalhe teórico, este livro destaca os assuntos que você precisa conhecer para poder aplicar POO em seus projetos diários — e não perder seu tempo se degladiando em algum debate teórico.

O objetivo deste livro é fornecer a você uma base sólida sobre programação orientada a objetos. Após 21 dias, você deverá ter uma boa idéia dos conceitos básicos da POO. Usando essa base, você pode começar a aplicar conceitos de POO em seus projetos diários, assim como continuar a construir seus conhecimentos de OO, através de estudo adicional. Você não aprenderá tudo a respeito de POO em 21 dias — isso simplesmente não é possível. Entretanto, é possível construir uma base sólida e começar com o pé direito. Este livro o ajuda a fazer exatamente isso.

Dividimos este livro em três partes. A Semana 1 apresenta os três princípios da POO (também conhecidos como os três pilares da POO). Esses três princípios formam a base da teoria da orientação a objetos. O entendimento desses três princípios é absolutamente fundamental para entender POO. As lições da semana estão divididas entre apresentação da teoria e o fornecimento de experiência prática, através de laboratórios.

A Semana 2 apresenta o processo de desenvolvimento de software OO. Embora as lições do Capítulo 1, “Introdução à programação orientada a objetos”, sejam importantes, mandá-lo programar sem qualquer outra instrução é como dar a você madeira, uma serra, um martelo e alguns pregos, e dizer para construir uma casa. A Semana 2 mostra como aplicar as ferramentas apresentadas nas lições da Semana 1.

A Semana 3 o conduz em um estudo de caso completo, de um jogo de cartas OO. Esse estudo permitirá que você percorra um ciclo de desenvolvimento OO inteiro, do início ao fim, assim como pratique a escrita de algum código. Esperamos que esse estudo de caso ajude a esclarecer a teoria da OO e torne as coisas mais concretas.

Também existem vários apêndices no final do livro. De importância especial é o “Resumo do Java”, no Apêndice B, e a bibliografia selecionada, no Apêndice D. O Apêndice B serve como um guia excelente para a linguagem de programação Java. A bibliografia mostra os recursos que você desejará consultar, quando continuar seus estudos de POO. Esses recursos certamente foram valiosos para escrever este livro.

## **Sobre os exemplos**

Todos os exemplos de código-fonte foram escritos em Java. Alguma experiência em Java ajudará; entretanto, o Apêndice B deve ajudá-lo a ter velocidade, caso você esteja enferrujado ou nunca tenha visto a linguagem antes. Desde que você tenha algum conhecimento de programação, os exemplos são os mais acessíveis. Os recursos e truques especiais do Java foram particularmente evitados nos exemplos.

## **O que você precisa saber para usar este livro**

Este livro presume alguma experiência anterior em programação e não tenta ensinar programação básica. Este livro pega o conhecimento que você já tem e mostra como pode usá-lo para escrever software orientado a objetos. Ou escrever software orientado a objetos melhor. Isso não quer dizer que você precisa ser um guru de programação para ler e entender este livro — um curso de programação introdutório ou simplesmente a leitura de um livro de programação é todo o conhecimento que você deve precisar.

Para poder tirar total proveito dos exemplos e exercícios, você também precisará de um computador com acesso à Internet. A escolha do ambiente operacional e do editor ficam completamente por conta de seu gosto pessoal. O único requisito é que você possa fazer download, instalar e executar Java. O Apêndice B o conduz pelo processo de obtenção de um SDK Java.

Finalmente, você precisa de determinação, dedicação e uma mente aberta. A programação orientada a objetos não é fácil e você demorará mais de 21 dias para dominar, mas aqui você pode ter um início bom e sólido.

O mundo maravilhoso da POO está à espera....

# SEMANA 1

## Definindo OO

- 1 Introdução à programação orientada a objetos
- 2 Encapsulamento: aprenda a manter os detalhes consigo mesmo
- 3 Encapsulamento: hora de escrever algum código
- 4 Herança: obtendo algo para nada
- 5 Herança: hora de escrever algum código
- 6 Polimorfismo: aprendendo a prever o futuro
- 7 Polimorfismo: hora de escrever algum código

1

2

3

4

5

6

7

# Panorama

Os próximos sete dias fornecerão a você uma base sólida sobre programação orientada a objetos. O Dia 1 descreve os fundamentos da OO (orientado a objetos). Você aprende a respeito de orientação a objetos a partir de uma perspectiva histórica e vê como a OO evoluiu a partir de linguagens de programação existentes. Você também aprende a terminologia básica, assim como as vantagens e armadilhas da programação orientada a objetos.

Os dias 2, 4 e 6 apresentam os três pilares da programação orientada a objetos: *encapsulamento*, *herança* e *polimorfismo*. Esses capítulos não apenas explicam os fundamentos da programação orientada a objetos, mas como e quando usá-los, assim como os erros a serem evitados.

Os dias 3, 5 e 7 fornecem laboratórios correspondentes a cada um dos três pilares. Cada capítulo de laboratório fornece experiência prática que lhe permite se familiarizar com os pilares apresentados nos dias 2, 4 e 6.

Após concluir a primeira semana, você deverá ter um entendimento completo do quê constitui um programa orientado a objetos. Você deve poder identificar os três pilares da OO e aplicá-los em seu código.

Testes e exercícios seguem a lição de cada dia, para ajudá-lo a entender melhor os assuntos abordados. As respostas de cada pergunta dos testes e exercícios aparecem no Apêndice A.

# SEMANA 1

DIA 1

## Introdução à programação orientada a objetos

Embora as linguagens orientadas a objetos já existam desde a década de 1960, os últimos 10 anos têm visto um crescimento sem paralelo no uso e na aceitação de tecnologias de objeto, por todo o setor de software. Embora tenham começado como algo secundário, sucessos recentes, como Java, CORBA e C++, têm impulsionado as técnicas orientadas a objetos (OO) para novos níveis de aceitação. Isso não é por acaso. Após anos presa nos meios acadêmicos e tendo de lutar uma árdua batalha contra as práticas entrancheiradas, a programação orientada a objetos (POO) amadureceu até o ponto onde as pessoas são finalmente capazes de perceber as promessas que a técnica contém. No passado, você tinha de convencer seu chefe a permitir o uso de uma linguagem orientada a objetos. Hoje, muitas empresas obrigam seu uso. É seguro dizer que as pessoas estão finalmente ouvindo.

Se você está lendo este livro, finalmente foi convencido. Provavelmente é alguém com um nível intermediário de experiência em programação. Se você conhece C, Visual Basic ou FORTRAN, já estava nas imediações, mas decidiu que precisa dar uma séria olhada na programação orientada a objetos e torná-la parte de seu conjunto de habilidades.

Mesmo que você tenha alguma experiência com uma linguagem orientada a objetos, este livro pode ajudá-lo a solidificar seu entendimento de OO. Mas não entre em pânico, caso você não esteja familiarizado com uma linguagem OO. Embora este livro use Java para ensinar conceitos de

OO, um conhecimento prévio de Java não é necessário. Se você ficar confuso ou precisar de um lembrete da sintaxe, basta consultar o Apêndice B, “Resumo do Java”.

Se você precisa de OO para se manter no mercado, terminar seu projeto mais recente ou satisfazer sua própria curiosidade, então veio ao lugar certo. Embora nenhum livro possa ensinar tudo que há em relação a OO, este livro promete fornecer uma base sólida em OO. Com essa base, você pode começar a praticar POO. E, mais importante, a fundamentação fornecerá a base durável que você precisa para continuar a aprender e, finalmente, dominar esse paradigma de programação.

Hoje você aprenderá

- Programação orientada a objetos em um contexto histórico
- A base da programação orientada a objetos
- As vantagens e objetivos da programação orientada a objetos
- As falácias e armadilhas comuns associadas à programação orientada a objetos

## Programação orientada a objetos em um contexto histórico

Para entender o estado atual da POO, você deve conhecer um pouco da história da programação. Ninguém concebeu a POO da noite para o dia. Em vez disso, a POO é apenas outro estágio na evolução natural do desenvolvimento de software. Com o passar do tempo, se tornou mais fácil identificar as práticas que funcionam e as que comprovadamente falham. A POO combina práticas comprovadas e testadas o mais eficientemente possível.

### Novo TERMO

*OO* é a abreviatura de orientado a objetos. OO é um termo geral que inclui qualquer estilo de desenvolvimento que seja baseado no conceito de ‘objeto’ — uma entidade que exibe características e comportamentos. Você pode aplicar uma estratégia orientada a objetos na programação, assim como na análise e no projeto.

Você também pode dizer que OO é um estado da mente, uma maneira de ver o mundo todo em termos de objetos.

Simplesmente, a OO contém tudo que pode ser denominado como orientado a objetos. Você vai ver o termo OO muitas vezes neste livro.

## Precursors da POO

Atualmente, quando usa um computador, você tira proveito de 50 anos de refinamento. Antigamente, a programação era engenhosa: os programadores introduziam os programas diretamente na memória principal do computador, através de bancos de chaves (switches). Os programadores escreviam seus programas em linguagem binária. Tal programação em linguagem binária

era extremamente propensa a erros e a falta de estrutura tornou a manutenção do código praticamente impossível. Além disso, o código da linguagem binária não era muito acessível.

Quando os computadores se tornaram mais comuns, linguagens de nível mais alto e procedurais começaram a aparecer; a primeira foi FORTRAN. Entretanto, linguagens procedurais posteriores, como ALGOL, tiveram mais influência sobre a OO. As linguagens procedurais permitem ao programador reduzir um programa em procedimentos refinados para processar dados. Esses procedimentos refinados definem a estrutura global do programa. Chamadas seqüenciais a esses procedimentos geram a execução de um programa procedural. O programa termina quando acaba de chamar sua lista de procedimentos.

Esse paradigma apresentou diversas melhorias em relação à linguagem binária, incluindo a adição de uma estrutura de apoio: o procedimento. As funções menores não são apenas mais fáceis de entender, mas também são mais fáceis de depurar. Por outro lado, a programação procedural limita a reutilização de código. E, com muita freqüência, os programadores produziam código de espaguetti — código cujo caminho de execução se assemelhava a uma tigela de espaguetti. Finalmente, a natureza voltada aos dados da programação procedural causou alguns problemas próprios. Como os dados e o procedimento são separados, não existe nenhum encapsulamento dos dados. Isso exige que cada procedimento saiba como manipular corretamente os dados. Infelizmente, um procedimento com comportamento errôneo poderia introduzir erros se não manipulasse os dados corretamente. Uma mudança na representação dos dados exigia alterações em cada lugar que acessasse os dados. Assim, mesmo uma pequena alteração poderia levar a uma cascata de alterações, por todo o programa — em outras palavras, um pesadelo de manutenção.

A programação modular, com uma linguagem como Modula2, tenta melhorar algumas das deficiências encontradas na programação procedural. A programação modular divide os programas em vários componentes ou módulos constituintes. Ao contrário da programação procedural, que separa dados e procedimentos, os módulos combinam os dois. Um módulo consiste em dados e procedimentos para manipular esses dados. Quando outras partes do programa precisam usar um módulo, elas simplesmente exercitam a interface do módulo. Como os módulos ocultam todos os dados internos do restante do programa, é fácil introduzir a idéia de estado: um módulo contém informações de estado que podem mudar a qualquer momento.

**Novo TERMO** O *estado* de um objeto é o significado combinado das variáveis internas do objeto.

**Novo TERMO** Uma *variável interna* é um valor mantido dentro de um objeto.

Mas a programação modular sofre de duas deficiências próprias importantes. Os módulos não são extensíveis, significando que você não pode fazer alterações incrementais em um módulo sem abrir o código a força e fazer as alterações diretamente. Você também não pode basear um módulo em outro, a não ser através de delegação. E, embora um módulo possa definir um tipo, um módulo não pode compartilhar o tipo de outro módulo.

Nas linguagens modulares e procedurais, os dados estruturados e não estruturados têm um ‘tipo’. O tipo é mais facilmente pensado como o formato da memória para os dados. As linguagens fortemente tipadas exigem que cada objeto tenha um tipo específico e definido. Entretanto, os tipos não podem ser estendidos para criar outro tipo, exceto através de um estilo chamado ‘agregação’. Por exemplo, em C, podemos ter dois tipos de dados relacionados:

```
typedef struct
{
    int a;
    int b;
} a BaseType;

typedef struct
{
    a BaseType Base;
    int c;
} a DerivedType;
```

Nesse exemplo, a DerivedType é baseado em a BaseType, mas uma estrutura de a DerivedType não pode ser tratada diretamente como uma estrutura de a BaseType. Uma só pode fazer referência ao membro Base de uma estrutura a DerivedType. Infelizmente, essa organização leva a código que possui muitos blocos case e if/else, pois o aplicativo deve saber como manipular cada módulo que encontra.

Finalmente, a programação modular também é um híbrido procedural que ainda divide um programa em vários procedimentos. Agora, em vez de atuar em dados brutos, esses procedimentos manipulam módulos.

## Programação orientada a objetos

A POO dá o próximo passo lógico após a programação modular, adicionando herança e polimorfismo ao módulo. A POO estrutura um programa, dividindo-o em vários objetos de alto nível. Cada objeto modela algum aspecto do problema que você está tentando resolver. Escrever listas seqüenciais de chamadas de procedimento para dirigir o fluxo do programa não é mais o foco da programação sob a OO. Em vez disso, os objetos interagem entre si, para orientar o fluxo global do programa. De certa forma, um programa OO se torna uma simulação viva do problema que você está tentando resolver.

## Uma estratégia de POO para software usando objetos

Imagine que você tivesse de desenvolver um programa OO para implementar um carrinho de compras on-line ou um terminal de ponto de vendas. Um programa OO conterá os objetos item, carrinho de compras, cupom e caixa. Cada um desses objetos vai interagir uns com os outros para orientar o programa. Por exemplo, quando o caixa totalizar um pedido, ele verificará o preço de cada item.

Definir um programa em termos de objetos é uma maneira profunda de ver o software. Os objetos o obrigam a ver tudo, em nível conceitual, do que um objeto faz: seus comportamentos. Ver um objeto a partir do nível conceitual é um desvio da observação de como algo é feito: a implementação. Essa mentalidade o obriga a pensar em seus programas em termos naturais e reais. Em vez de modelar seu programa como um conjunto de procedimentos e dados separados (termos do mundo do computador), você modela seu programa em objetos. Os objetos permitem que você modele seus programas nos substantivos, verbos e adjetivos do domínio de seu problema.

**Novo TERMO**

A *implementação* define como algo é feito. Em termos de programação, *implementação* é o código.

**Novo TERMO**

O *domínio* é o espaço onde um problema reside. O domínio é o conjunto de conceitos que representam os aspectos importantes do problema que você está tentando resolver.

Quando recua e pensa nos termos do problema que está resolvendo, você evita se emaranhar nos detalhes da implementação. É claro que alguns de seus objetos de alto nível precisarão interagir com o computador. Entretanto, o objeto isolará essas interações do restante do sistema. (O Dia 2, “Encapsulamento: aprenda a manter os detalhes consigo mesmo”, explorará melhor essas vantagens.)



Em termos do carrinho de compras, *ocultação de implementação* significa que o caixa não vê dados brutos ao totalizar um pedido. O caixa não sabe procurar, em certas posições de memória, números de item e outra variável para um cupom. Em vez disso, o caixa interage com objetos item. Ele sabe perguntar quanto custa o item.

Neste ponto, você pode definir *objeto* formalmente:

**Novo TERMO**

Um *objeto* é uma construção de software que encapsula estado e comportamento. Os objetos permitem que você modele seu software em termos reais e abstrações.

Rigorosamente falando, um objeto é uma instância de uma classe. A próxima seção apresentará o conceito de *classe*.

Assim como o mundo real é constituído de objetos, da mesma forma o é o software orientado a objetos. Em uma linguagem de programação OO pura, tudo é um objeto, desde os tipos mais básicos, como inteiros e lógicos, até as instâncias de classe mais complexas; nem todas as linguagens orientadas a objeto chegam a esse ponto. Em algumas (como o Java), primitivas como int e float, não são tratadas como objetos.

## O que é uma classe?

Assim como os objetos do mundo real, o mundo da POO agrupa os objetos pelos seus comportamentos e atributos comuns.

A biologia classifica todos os cães, gatos, elefantes e seres humanos como mamíferos. Características compartilhadas dão a essas criaturas separadas um senso de comunidade. No mundo do software, as classes agrupam objetos relacionados da mesma maneira.

Uma classe define todas as características comuns a um tipo de objeto. Especificamente, a classe define todos os atributos e comportamentos expostos pelo objeto. A classe define a quais mensagens seus objetos respondem. Quando um objeto quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Freqüentemente, isso é referido como ‘envio de uma mensagem’.

**Novo TERMO** Uma *classe* define os atributos e comportamentos comuns compartilhados por um tipo de objeto. Os objetos de certo tipo ou classificação compartilham os mesmos comportamentos e atributos. As classes atuam de forma muito parecida com um cortador de molde ou biscoito, no sentido de que você usa uma classe para criar ou *instanciar* objetos.

**Novo TERMO** *Atributos* são as características de uma classe visíveis externamente. A cor dos olhos e a cor dos cabelos são exemplos de atributos.

Um objeto pode expor um atributo fornecendo um link direto a alguma variável interna ou retornando o valor através de um método.

**Novo TERMO** *Comportamento* é uma ação executada por um objeto quando passada uma mensagem ou em resposta a uma mudança de estado: é algo que um objeto faz.

Um objeto pode exercer o *comportamento* de outro, executando uma operação sobre esse objeto. Você pode ver os termos *chamada de método*, *chamada de função* ou *passar uma mensagem*, usados em vez de executar uma *operação*. O que é importante é que cada uma dessas ações omite o comportamento de um objeto.



*Passagem de mensagem, operação, chamada de método e chamada de função*; o quê você usa, freqüentemente depende de seus conceitos anteriores.

Pensar em termos de *passagem de mensagem* é uma maneira muito orientada a objetos de pensar. A *passagem de mensagem* é dinâmica. Conceitualmente, ela separa a *mensagem* do objeto. Tal mentalidade pode ajudar a pensar a respeito das interações entre objetos.

Linguagens como C++ e Java têm heranças procedurais, onde as chamadas de função são estáticas. Como resultado, essas linguagens freqüentemente se referem a um objeto realizando uma *chamada de método* a partir de outro objeto. Uma *chamada de método* está fortemente acoplada ao objeto.

Este livro normalmente usará *chamada de método*, devido à sua forte ligação com Java. Entretanto, podem existir ocasiões em que o termo *mensagem* é usado indistintamente.

## Reunindo tudo: classes e objetos

Pegue um objeto item, por exemplo. Um item tem uma descrição, id, preço unitário, quantidade e um desconto opcional. Um item saberá calcular seu preço descontado.

No mundo da POO, você diria que todos os objetos item são instâncias da classe Item. Uma classe Item poderia ser como segue:

```
public class Item {  
  
    private double unit_price;  
    private double discount; //uma porcentagem de desconto que se aplica ao preço  
    private int quantity;  
    private String description;  
    private String id;  
  
    public Item( String id, String description, int quantity, double price ) {  
        this.id = id;  
        this.description = description;  
  
        if( quantity >= 0 ) {  
            this.quantity = quantity;  
        }  
        else {  
            this.quantity = 0;  
        }  
  
        this.unit_price = price;  
    }  
  
    public double getAdjustedTotal() {  
        double total = unit_price * quantity;  
        double total_discount = total * discount;  
        double adjusted_total = total - total_discount;  
  
        return adjusted_total;  
    }  
  
    // aplica uma porcentagem de desconto no preço  
    public void setDiscount( double discount ) {  
        if( discount <= 1.00 ) {  
            this.discount = discount;  
        }  
        else {  
  
            this.discount = 0.0;  
        }  
    }  
}
```

```
}

public double getDiscount() {
    return discount;
}

public int getQuantity() {
    return quantity;
}

public void setQuantity( int quantity ) {
    if( quantity >= 0 ) {
        this.quantity = quantity;
    }
}

public String getProductID() {
    return id;
}

public String getDescription() {
    return description;
}
}
```

Métodos como

`public Item( String id, String description, int quantity, double price )`

são chamados *construtores*. Os construtores inicializam um objeto durante sua criação.

**Novo Termo**

*Construtores* são métodos usados para inicializar objetos durante sua instanciação. Você chama a criação de objetos de *instanciação* porque ela cria uma instância do objeto da classe.



No construtor e por todo o exemplo `Item`, você pode notar o uso de `this`. `this` é uma referência que aponta para a instância do objeto. Cada objeto tem sua própria referência para si mesmo. A instância usa essa referência para acessar suas próprias variáveis e métodos.

Métodos como `setDiscount()`, `getDescription()` e `getAdjustedTotal()` são todos comportamentos da classe `Item` que retornam ou configuram atributos. Quando um caixa quer totalizar o carrinho, ele simplesmente pega cada item e envia ao objeto a mensagem `getAdjustedTotal()`.

`unit_price`, `discount`, `quantity`, `description` e `id` são todas variáveis internas da classe `Item`. Esses valores compreendem o *estado* do objeto. O estado de um objeto pode variar com o tempo. Por exemplo, ao fazer compras, um consumidor pode aplicar um cupom ao item. Aplicar um cupom ao item mudará o estado do item, pois isso mudará o valor de `discount`.

Os métodos como `getAdjustedTotal()` e `getDiscount()` são chamados de *acessores*, pois eles permitem que você acesse os dados internos de um objeto. O acesso pode ser direto, como no caso de `getDiscount()`. Por outro lado, o objeto pode realizar processamento antes de retornar um valor, como no caso de `getAdjustedTotal()`.

**Novo Termo** Os *acessores* dão acesso aos dados internos de um objeto. Entretanto, os *acessores* ocultam o fato de os dados estarem em uma variável, em uma combinação de variáveis ou serem calculados. Os *acessores* permitem que você mude ou recupere o valor e têm ‘efeitos colaterais’ sobre o estado interno.

Os métodos como `setDiscount()` são chamados de *mutantes*, pois eles permitem que você altere o estado interno do objeto. Um mutante pode processar sua entrada como quiser, antes de alterar o estado interno do objeto. Pegue `setDiscount()`, por exemplo. `setDiscount()` garante que o desconto não seja maior que 100%, antes de aplicá-lo.

**Novo Termo** Os *mutantes* permitem que você altere o estado interno de um objeto.

Ao serem executados, seus programas usam classes como a `Item` para criar ou instanciar os objetos que compõem o aplicativo. Cada nova instância é uma duplicata da última. Entretanto, uma vez instanciada, a instância transporta comportamentos e controla seu estado. Então, o que inicia sua vida como clone poderia se comportar de maneira muito diferente durante sua existência.

Por exemplo, se você criar dois objetos `item` a partir da mesma classe `Item`, um objeto `item` poderá ter um desconto de 10%, enquanto o segundo pode não ter desconto. Alguns itens também são um pouco mais caros que outros. Um item poderia custar US\$1.000, enquanto outro poderia custar apenas US\$1,98. Assim, embora o estado de um item possa variar com o passar do tempo, a instância ainda é um objeto de `Item`. Considere o exemplo da biologia; um mamífero de cor cinza é tão mamífero quanto outro de cor marrom.

## Fazendo os objetos trabalhar

Considere o método `main()` a seguir:

```
public static void main( String [] args ) {  
    // cria os itens  
    Item milk   = new Item( "dairy-011", "1 Gallon Milk", 2,2.50 );  
    Item yogurt = new Item( "dairy-032", "Peach Yogurt", 4,0.68 );  
    Item bread   = new Item( "bakery-023", "Sliced Bread", 1,2.55 );  
    Item soap    = new Item( "household-21", "6 Pack Soap", 1,4.51 );
```

```

// aplica cupons
milk.setDiscount( 0.15 );
// obtém preços ajustados
double milk_price = milk.getAdjustedTotal();
double yogurt_price = yogurt.getAdjustedTotal();
double bread_price = bread.getAdjustedTotal();
double soap_price = soap.getAdjustedTotal();
// imprime recibo
System.out.println( "Thank You For Your Purchase." );
System.out.println( "Please Come Again!" );
System.out.println( milk.getDescription() + "\t $" + milk_price );
System.out.println( yogurt.getDescription() + "\t $" + yogurt_price );
System.out.println( bread.getDescription() + "\t $" + bread_price );
System.out.println( soap.getDescription() + "\t $" + soap_price );

// calcula e imprime total
double total = milk_price + yogurt_price + bread_price + soap_price;
System.out.println( "Total Price \t $" + total );
}

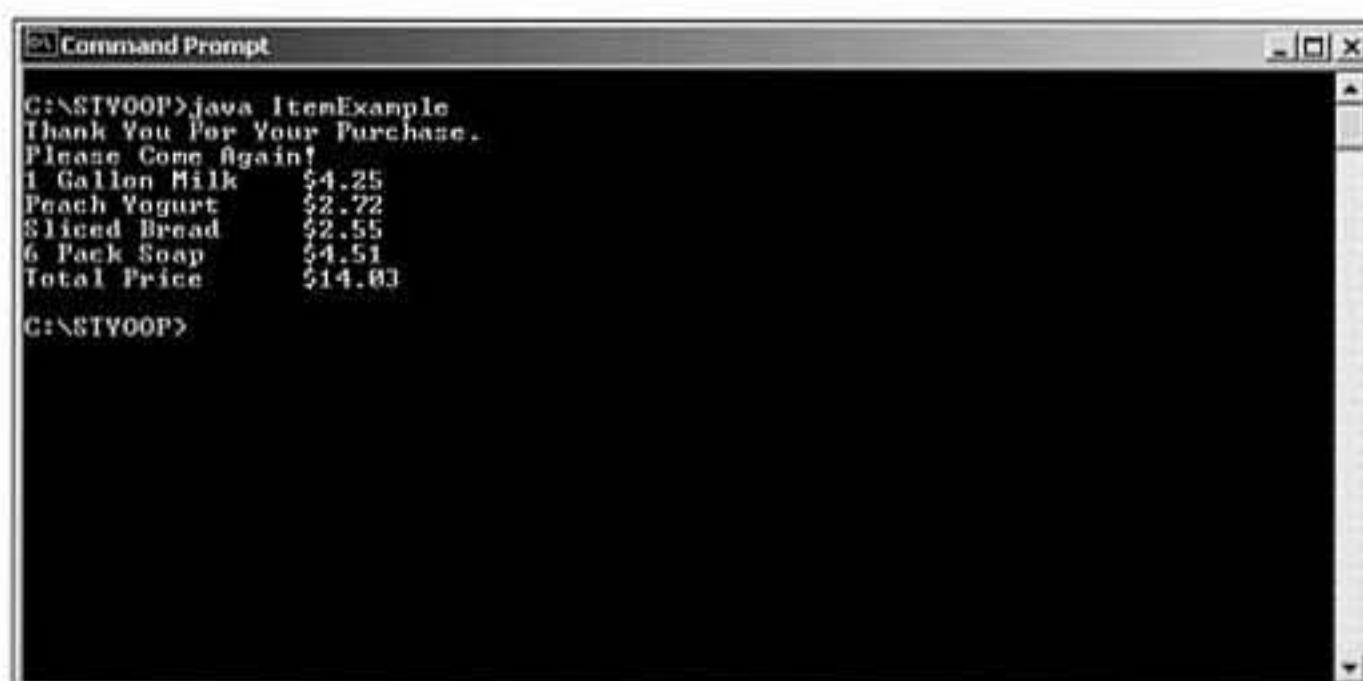
```

Esse método mostra como poderia ser um pequeno programa que usa objetos de `Item`. Primeiro, o programa instancia quatro objetos de `Item`. Em um programa real, ele poderia criar esses itens quando um usuário navegassem em um catálogo on-line ou quando um caixa percorre os armazéns.

Esse programa cria vários itens, aplica descontos e, em seguida, imprime um recibo. O programa realiza toda a interação do objeto enviando várias mensagens para os itens. Por exemplo, o programa aplica um desconto de 15% no leite (`milk`), enviando a mensagem `setDiscount()` para o item. O programa totaliza os itens primeiro enviando a cada item a mensagem `getAdjustedTotal()`.

Finalmente, esse programa envia um recibo para a tela. A Figura 1.1 ilustra o exemplo de saída.

**FIGURA 1.1**  
*Imprimindo  
um recibo.*



É importante perceber que toda a programação foi feita em termos de objetos de `Item` e os comportamentos expostos pelos métodos de `Item`—os substantivos e verbos do domínio do carrinho de compras.

## Relacionamentos de objeto

O modo como os objetos se relacionam é um componente muito importante da POO. Os objetos podem se relacionar de duas maneiras importantes.

Primeiro, os objetos podem existir independentemente uns dos outros. Dois objetos de Item podem aparecer no carrinho de compras simultaneamente. Se esses dois objetos separados precisarem interagir, eles interagirão passando mensagens um para o outro.

1

**Novo Termo**

Os objetos se comunicam uns com os outros através de *mensagens*. As mensagens fazem com que um objeto realize algo.

‘Passar uma mensagem’ é o mesmo que chamar um método para mudar o estado do objeto ou para exercer um comportamento.

Segundo, um objeto poderia conter outros objetos. Assim, como os objetos compõem um programa em POO, eles podem compor outros objetos através da agregação. A partir do exemplo Item, você poderia notar que o objeto item contém muitos outros objetos. Por exemplo, o objeto item também contém uma descrição e uma id. A descrição e a id são ambas objetos de String. Cada um desses objetos tem uma interface que oferece métodos e atributos. Lembre-se de que, na POO, tudo é um objeto, mesmo as partes que compõem um objeto!

A comunicação funciona da mesma maneira entre um objeto e os objetos que ele contém. Quando os objetos precisarem interagir, eles farão isso enviando mensagens uns para os outros.

As mensagens são um importante conceito de OO. Elas permitem que os objetos permaneçam independentes. Quando um objeto envia uma mensagem para outro, geralmente ele não se preocupa com a maneira como o objeto escolhe transportar o comportamento solicitado. O objeto solicitante se preocupa apenas que o comportamento aconteça.

Você vai aprender mais a respeito de como os objetos se relacionam, na próxima semana.



**ALERTA**

A definição de objeto está aberta ao debate. Algumas pessoas não definem um objeto como uma instância de uma classe. Em vez disso, elas definem tudo em termos de ser um objeto: a partir desse ponto de vista, uma classe é um objeto que cria outros objetos. Tratar uma classe como um objeto é importante para conceitos como metaclasses.

Quando este livro encontrar uma discordância na terminologia, escolheremos uma definição e ficaremos com ela. Muito freqüentemente, a escolha será pragmática. Aqui, optamos por usar a definição de objeto como instância. Essa é a definição UML (*Unified Modeling Language*) e a mais encontrada no setor. (Você vai aprender mais a respeito de UML posteriormente.) Infelizmente, a outra é uma definição orientada a objetos mais pura. Entretanto, você não a encontrará muito freqüentemente e o conceito de metaclasses está fora dos objetivos deste livro.

## Como a programação orientada a objetos fundamenta o passado

Assim como outros paradigmas tentam acentuar as vantagens e corrigir as falhas dos paradigmas anteriores, a POO fundamenta a programação procedural e modular.

A programação modular estrutura um programa em vários módulos. Do mesmo modo, a POO divide um programa em vários objetos interativos. Assim como os módulos ocultam representações de dados atrás de procedimentos, os objetos encapsulam seu estado por trás de suas interfaces. A POO empresta esse conceito de encapsulamento diretamente da programação modular. O encapsulamento difere muito da programação procedural. A programação procedural não encapsula dados. Em vez disso, os dados são abertos para todos os procedimentos acessarem. Ao contrário da programação procedural, a programação orientada a objetos acopla fortemente dados e comportamentos ao objeto. Você vai aprender mais a respeito do encapsulamento nos dias 2 e 3.

Embora os objetos sejam conceitualmente semelhantes aos módulos, eles diferem de várias maneiras importantes. Primeiro, os módulos não suportam extensão prontamente. A programação orientada a objetos introduz o conceito de herança para eliminar essa deficiência. A herança permite que você estenda e melhore suas classes facilmente. A herança também permite que você classifique suas classes. Você vai aprender mais a respeito da herança no Dia 4, “Herança: obtendo algo para nada” e no Dia 5, “Herança: hora de escrever algum código”.

A POO também acentua o conceito de polimorfismo, que ajuda a construir programas flexíveis, que não resistem à mudança. O polimorfismo acrescenta essa flexibilidade limpando o sistema de tipagem limitado do módulo. Você vai aprender mais a respeito do polimorfismo no Dia 6, “Polimorfismo: aprendendo a prever o futuro” e no Dia 7, “Polimorfismo: hora de escrever algum código”.

A POO certamente não inventou o encapsulamento e o polimorfismo. Em vez disso, a POO combina esses conceitos em um só lugar. Pegue a noção da POO de objetos e você reunirá essas tecnologias de uma maneira jamais feita.

## Vantagens e objetivos da OO

A programação orientada a objetos define seis objetivos sobrepostos para desenvolvimento de software. A POO se esmera em produzir software que tenha as seguintes características:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível

5. Extensível
6. Oportunos

Vamos ver como ela funciona para atender cada um desses objetivos.

## Natural

A POO produz software natural. Os programas naturais são mais inteligíveis. Em vez de programar em termos de regiões de memória, você pode programar usando a terminologia de seu problema em particular. Você não precisa se aprofundar nos detalhes do computador enquanto projeta seu programa. Em vez de ajustar seus programas para a linguagem do mundo dos computadores, a OO o libera para que expresse seu programa nos termos de seu problema.

A programação orientada a objetos permite que você modele um problema em um nível funcional e não em nível de implementação. Você não precisa saber como um software funciona, para usá-lo: você simplesmente se concentra no que ele faz.

## Confiável

Para criar software útil, você precisa criar software que seja tão confiável quanto outros produtos, como geladeiras e televisões. Quando foi a última vez que seu microondas quebrou?

Programas orientados a objetos, bem projetados e cuidadosamente escritos são confiáveis. A natureza modular dos objetos permite que você faça alterações em uma parte de seu programa, sem afetar outras partes. Os objetos isolam o conhecimento e a responsabilidade de onde pertencem.

Uma maneira de aumentar a confiabilidade é através de testes completos. A OO aprimora os testes, permitindo que você isole conhecimento e responsabilidade em um único lugar. Tal isolamento permite que você teste e valide cada componente independentemente. Uma vez que tenha validado um componente, você pode reutilizá-lo com confiança.

## Reutilizável

Um construtor inventa um novo tipo de tijolo cada vez que constrói uma casa? Um engenheiro eletricista inventa um novo tipo de resistor cada vez que projeta um circuito? Então, por que os programadores continuam ‘reinventando a roda?’ Uma vez que um problema esteja resolvido, você deve reutilizar a solução.

Você pode reutilizar prontamente classes orientadas a objetos bem feitas. Assim como os módulos, você pode reutilizar objetos em muitos programas diferentes. Ao contrário dos módulos, a POO introduz a herança para permitir que você estenda objetos existentes e o polimorfismo, para que você possa escrever código genérico.

A OO não garante código genérico. Criar classes bem feitas é uma tarefa difícil que exige concentração e atenção à abstração. Os programadores nem sempre acham isso fácil.

Através da POO, você pode modelar idéias gerais e usar essas idéias gerais para resolver problemas específicos. Embora você vá construir objetos para resolver um problema específico, freqüentemente construirá esses objetos específicos usando partes genéricas.

## Manutenível

O ciclo de vida de um programa não termina quando você o distribui. Em vez disso, você deve manter sua base de código. Na verdade, entre 60% e 80% do tempo gasto trabalhando em um programa é para manutenção. O desenvolvimento representa apenas 20% da equação!

Um código orientado a objetos bem projetado é manutenível. Para corrigir um erro, você simplesmente corrige o problema em um lugar. Como uma mudança na implementação é transparente, todos os outros objetos se beneficiarão automaticamente do aprimoramento. A linguagem natural do código deve permitir que outros desenvolvedores também o entendam.

## Extensível

Assim como você deve manter um programa, seus usuários exigem o acréscimo de nova funcionalidade em seu sistema. Quando você construir uma biblioteca de objetos, também desejará estender a funcionalidade de seus próprios objetos.

A POO trata dessas realidades. O software não é estático. Ele deve crescer e mudar com o passar do tempo, para permanecer útil. A POO apresenta ao programador vários recursos para estender código. Esses recursos incluem herança, polimorfismo, sobreposição, delegação e uma variedade de padrões de projeto.

## Oportuno

O ciclo de vida do projeto de software moderno é freqüentemente medido em semanas. A POO ajuda nesses rápidos ciclos de desenvolvimento. A POO diminui o tempo do ciclo de desenvolvimento, fornecendo software confiável, reutilizável e facilmente extensível.

O software natural simplifica o projeto de sistemas complexos. Embora você não possa ignorar o projeto cuidadoso, o software natural pode otimizar os ciclos de projeto, pois você pode se concentrar no problema que está tentando resolver.

Quando você divide um programa em vários objetos, o desenvolvimento de cada parte pode ocorrer em paralelo. Vários desenvolvedores podem trabalhar nas classes independentemente. Tal desenvolvimento em paralelo leva a tempos de desenvolvimento menores.

## Armadilhas

Quando você aprende OO pela primeira vez, existem quatro armadilhas que precisa evitar.

## Armadilha 1: pensar na POO simplesmente como uma linguagem

Freqüentemente, as pessoas equiparam linguagens orientadas a objetos com a POO. O erro surge ao supor que você está programando de maneira orientada a objetos simplesmente porque usa uma linguagem orientada a objetos. Nada poderia estar mais distante da realidade.

A POO é muito mais do que simplesmente usar uma linguagem orientada a objetos ou conhecer certo conjunto de definições. Você pode escrever código horrivelmente não orientado a objetos em uma linguagem orientada a objetos. A verdadeira POO é um estado da mente que exige que você veja seus problemas como um grupo de objetos e use encapsulamento, herança e polimorfismo corretamente.

Infelizmente, muitas empresas e programadores supõem que, se simplesmente usarem uma linguagem orientada a objetos, se beneficiarão de todas as vantagens que a POO oferece. Quando falham, elas tentam culpar a tecnologia e não o fato de que não treinaram seus funcionários corretamente, ou que agarraram um conceito de programação popular sem entender realmente o que ele significava.

## Armadilha 2: medo da reutilização

Você deve aprender a reutilizar código. Aprender a reutilizar sem culpa freqüentemente é uma das lições mais difíceis de aprender, quando você escolhe a POO pela primeira vez. Três problemas levam a essa dificuldade.

Primeiro, os programadores gostam de criar. Se você olhar a reutilização de modo errado, ela parecerá afastar algumas das alegrias da criação. Entretanto, você precisa lembrar que está reutilizando partes para criar algo maior. Pode não parecer interessante reutilizar um componente, mas isso permitirá que você construa algo ainda melhor.

Segundo, muitos programadores sofrem do sentimento de ‘não escrito aqui’— significando que eles não confiam no software que não escreveram. Se um software é bem testado e atende sua necessidade, você deve reutilizá-lo. Não rejeite um componente porque você não o escreveu. Lembre-se de que reutilizar um componente o liberará para escrever outro software maravilhoso.

## Armadilha 3: pensar na OO como uma solução para tudo

Embora a POO ofereça muitas vantagens, ela não é a solução para tudo no mundo da programação. Existem ocasiões em que você não deve usar OO. Você ainda precisa usar bom senso na escolha da ferramenta correta para o trabalho a ser feito. Mais importante, a POO não garante o sucesso de seu projeto. Seu projeto não terá sucesso automaticamente, apenas porque você usa uma linguagem OO. O sucesso aparece somente com planejamento, projeto e codificação cuidadosos.

## Armadilha 4: programação egoísta

Não seja egoísta quando programar. Assim como você deve aprender a reutilizar, também deve aprender a compartilhar o código que cria. Compartilhar significa que você encorajará outros desenvolvedores a usarem suas classes. Entretanto, compartilhar também significa que você tornará fácil para outros reutilizarem essas classes.

Lembre-se dos outros desenvolvedores quando você programar. Faça interfaces limpas e inteligíveis. Mais importante, escreva a documentação. Documente suposições, parâmetros de métodos, documente o máximo que você puder. As pessoas não reutilizarão o que não podem encontrar ou entender.

## A próxima semana

Na próxima semana, você continuará sua introdução a POO, aprendendo a respeito dos três pilares que formam a base da teoria da POO: *encapsulamento, herança e polimorfismo*.

Cada pilar será dividido em duas lições. A primeira lição apresentará o pilar e a teoria subjacente. A segunda lição fornecerá experiência prática com os conceitos apresentados no dia anterior. Essa estratégia espelha a estratégia de preleção/laboratório, usada com sucesso por muitas universidades e escolas.

Você completará todos esses laboratórios usando a linguagem de programação Java da Sun Microsystems. Você pode obter gratuitamente todas as ferramentas usadas neste livro, através da World Wide Web. O Dia 3, assim como o Apêndice B, “Resumo do Java”, no final do livro, o conduzirão na obtenção e configuração de seu ambiente de desenvolvimento.



### NOTA

#### Por que Java?

Existem dois motivos para se usar Java como ferramenta de ensino. Primeiro, o Java abstrai perfeitamente dos detalhes da máquina e do sistema operacional. Em vez de ter de se preocupar com alocação e desalocação de memória, você pode simplesmente se concentrar no aprendizado dos objetos. Finalmente, aprender boas práticas orientadas a objetos em Java é prático. Você pode pegar o conhecimento e fazer um trabalho. Algumas linguagens são mais orientadas a objetos do que o Java. Entretanto, é fácil fazer o Java funcionar.

## Resumo

Hoje, você fez um passeio pela programação orientada a objetos. Você começou vendo a evolução dos principais paradigmas de programação e aprendeu alguns dos fundamentos da POO. Agora, você deve entender as idéias conceituais por trás da OO, como o que é uma classe e como os objetos se comunicam.

Definições são importantes, mas nunca devemos perder o rumo do que estamos tentando fazer usando OO, nos atendo ao ‘como’ do que estivermos fazendo. As seis vantagens e objetivos resumem o que a programação orientada a objetos espera cumprir:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível
6. Oportuna

1

Você nunca deve perder esses objetivos de vista.

## Perguntas e respostas

**P** **O que posso fer para dominar a POO?**

**R** Livros como este são uma boa maneira de começar em sua jornada para dominar a OO. É importante construir uma base sólida; uma que você possa desenvolver.

Uma vez que tenha uma base, você precisa começar a praticar OOativamente. O verdadeiro domínio só vem com a experiência. Comece como um desenvolvedor em um projeto OO. Conheça profundamente o assunto. Quando você se tornar mais familiarizado com a OO, comece a se envolver na análise e no projeto de seus trabalhos.

Também ajuda encontrar um mentor. Encontre alguém que esteja desejoso de passar algum tempo compartilhando sabedoria. Instruir-se com os outros é a maneira melhor e mais rápida de aprender POO.

Finalmente, continue seu estudo pessoal. Leia livros, artigos, participe de conferências. Você sempre pode absorver mais informações.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. O que é programação procedural?
2. Qual vantagem a programação procedural tem em relação à programação não estruturada?
3. O que é programação modular?

4. Quais vantagens a programação modular tem em relação à programação procedural?
5. Liste uma deficiência da programação procedural e da programação modular.
6. O que é programação orientada a objetos?
7. Quais são as seis vantagens e objetivos da programação orientada a objetos?
8. Explique um dos objetivos da programação orientada a objetos.
9. Defina os seguintes termos:  
Classe  
Objeto  
Comportamento
10. Como os objetos se comunicam entre si?
11. O que é um construtor?
12. O que é um acessor?
13. O que é um mutante?
14. O que é this?

## Exercícios

Alegre-se! Para hoje, você não tem exercícios escritos. Em vez disso, dê um passeio.

# SEMANA 1

## DIA 2

### Encapsulamento: aprenda a manter os detalhes consigo mesmo

Esperamos que o Dia 1, “Introdução à programação orientada a objetos”, tenha aguçado seu interesse e que você provavelmente tenha muitas perguntas. Conforme você pode imaginar, existe muito mais na programação orientada a objetos do que algumas definições simples. Ao adotar uma estratégia de OO para desenvolvimento de software, você não pode simplesmente sair programando. Em vez disso, você deve ter um planejamento cuidadoso e uma boa base nas importantes teorias existentes por trás da POO. Infelizmente, não há uma maneira prática de se tornar um especialista em POO em poucos anos, imagine em 21 dias! Em vez disso, você precisa voltar e perguntar, “o que estou tentando fazer?” Você está tentando se tornar um especialista teórico ou um profissional prático? Você sabe, é preciso ser um pouco mais prático se quiser aprender OO suficientemente para fazer seu trabalho. Felizmente, você não precisa de um doutorado para entender e aplicar OO em seus projetos de software eficientemente. O que você precisa é de uma mente aberta e o desejo de aprender — ou desaprender, em muitos casos.

Hoje e no restante da semana, você dará uma olhada prática nas teorias subjacentes a POO: as ferramentas de OO. Essas teorias devem lhe dar uma base suficiente para começar a experimentar a POO. Entretanto, o domínio não virá rapidamente. Assim como qualquer outra habilidade, seus conhecimentos de POO melhorarão e aumentarão apenas com estudo e prática.

Hoje você vai aprender:

- Sobre os três pilares da programação orientada a objetos
- Como aplicar encapsulamento eficazmente
- Como programar de forma abstrata
- Como os tipos abstratos de dados formam a base do encapsulamento
- A diferença entre interface e implementação
- Sobre a importância da responsabilidade
- Como o encapsulamento atinge os objetivos da OO

## Os três pilares da programação orientada a objetos

Para edificar seu entendimento e domínio de OO, você deve primeiro ter uma base sólida a partir da qual possa expandir sua compreensão. Primeiro, você precisará identificar, definir e explorar os conceitos básicos da OO. Somente quando você tiver uma boa base das teorias básicas de OO é que poderá aplicá-la corretamente no software que escrever. Tal discussão o leva naturalmente aos três conceitos que devem estar presentes para que uma linguagem seja considerada realmente orientada a objetos. Esses três conceitos são freqüentemente referidos como os *três pilares* da programação orientada a objetos.

**Novo TERMO**

Os *três pilares* da programação orientada a objetos são: *encapsulamento*, *herança* e *polimorfismo*.

Como a POO é baseada neles, os três pilares são semelhantes a uma torre de blocos: remova o bloco inferior e tudo mais virá abaixo. O encapsulamento, que você abordará hoje, é uma peça extremamente importante do quebra-cabeça, pois ele forma a base da herança e do polimorfismo.

## Encapsulamento: o primeiro pilar

Em vez de ver um programa como uma única entidade grande e monolítica, o encapsulamento permite que você o divida em várias partes menores e independentes. Cada parte possui implementação e realiza seu trabalho independentemente das outras partes. O encapsulamento mantém essa independência, ocultando os detalhes internos ou seja, a implementação de cada parte, através de uma interface externa.

**Novo TERMO**

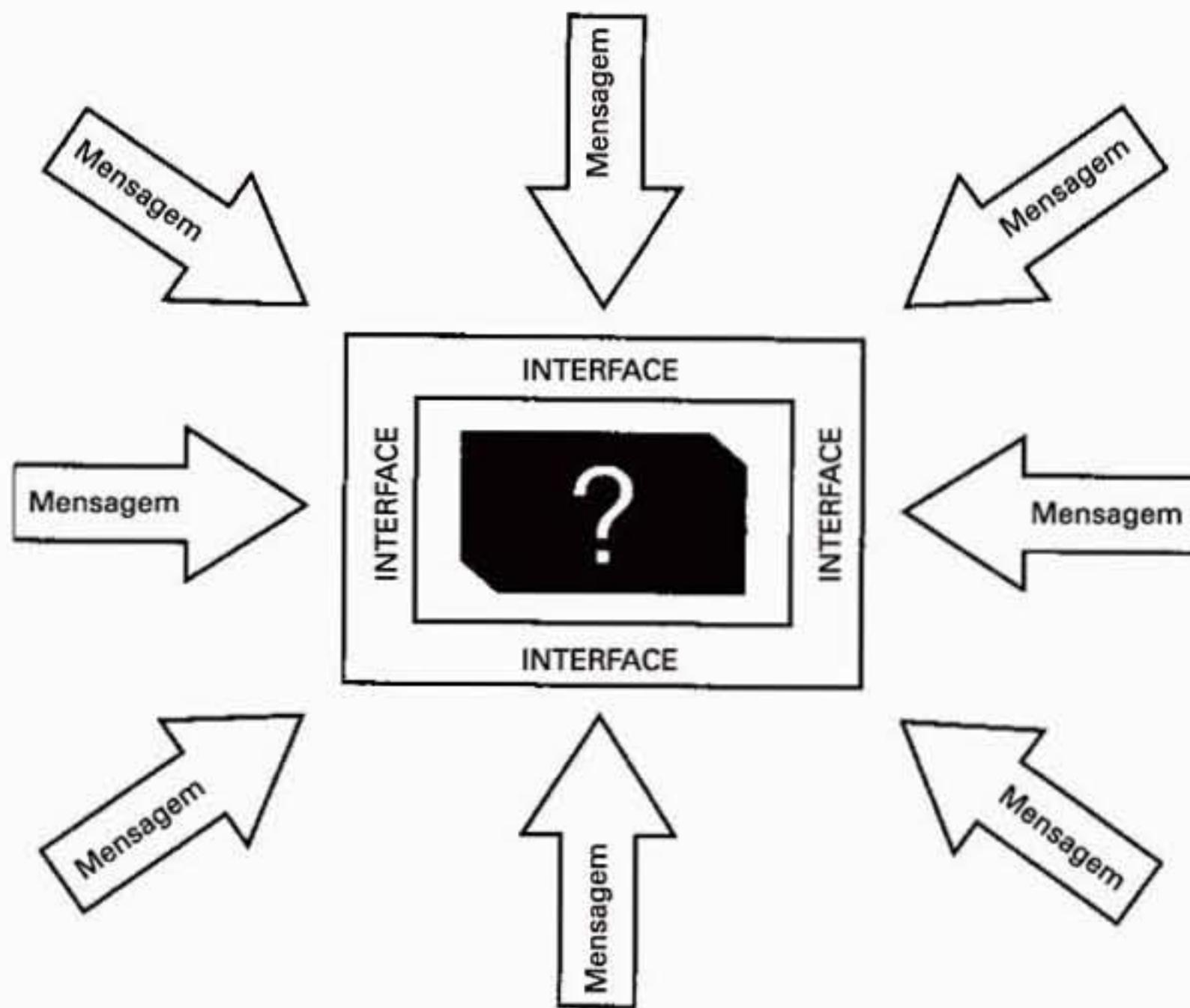
Encapsulamento é a característica da OO de ocultar partes independentes da implementação. O encapsulamento permite que você construa partes ocultas da implementação do software, que atinjam uma funcionalidade e ocultam os detalhes de implementação do mundo exterior.

**NOTA**

Se você não estiver familiarizado com o termo *encapsulamento*, talvez reconheça os termos *módulo*, *componente* ou *bean*. Você pode usar esses termos em vez de ‘software encapsulado’.

Uma vez encapsulado, você pode ver uma entidade de software como uma caixa preta. Você sabe o que a caixa preta faz, pois conhece sua interface externa. Conforme a Figura 2.1 ilustra, você simplesmente envia mensagens para a caixa preta. Você não se preocupa com o que acontece dentro da caixa; você só se preocupa com o fato de que isso aconteça.

2

**FIGURA 2.1***Uma caixa preta.***NOVO TERMO**

Uma *interface* lista os serviços fornecidos por um componente. A interface é um contrato com o mundo exterior, que define exatamente o que uma entidade externa pode fazer com o objeto. Uma interface é o painel de controle do objeto.

**NOTA**

Uma interface é importante, pois ela diz o que você pode fazer com o componente. O mais interesse é o que uma interface *não* informa: como o componente fará seu trabalho. Em vez disso, a interface oculta a implementação do mundo exterior. Isso libera o componente para alterações na sua implementação a qualquer momento. As mudanças na implementação não mudam o código que usa a classe, desde que a interface permaneça inalterada. As alterações na interface necessitarão de mudanças no código que exerce essa interface.



Talvez você esteja familiarizado com o termo de programação API (Interface de Programa Aplicativo). Uma *interface* é semelhante a API para um objeto. A interface lista todos os métodos e argumentos que o objeto entende.

**Novo TERMO**

A implementação define como um componente realmente fornece um serviço. A implementação define os detalhes internos do componente.

## Um exemplo de interface e implementação

Considere a classe Log a seguir:

```
public class Log {  
  
    public void debug( String message ) {  
        print( "DEBUG ", message );  
    }  
  
    public void info( String message ) {  
        print( "INFO ", message );  
    }  
  
    public void warning( String message ) {  
        print( "WARNING ", message );  
    }  
  
    public void error( String message ) {  
        print( "ERROR ", message );  
    }  
  
    public void fatal( String message ) {  
        print( "FATAL ", message );  
        System.exit( 0 );  
    }  
  
    private void print( String message, String severity ) {  
        System.out.println( severity + ": " + message );  
    }  
}
```

A classe Log contém objetos para relatar mensagens de depuração, informativas, de alerta e de erro, durante a execução. A interface da classe Log é constituída de todos os comportamentos disponíveis para o mundo exterior. Os comportamentos disponíveis para o mundo exterior são conhecidos como interface pública. A interface pública da classe Log inclui os seguintes métodos:

```
public void debug( String message )  
public void info( String message )  
public void warning( String message )  
public void error( String message )  
public void fatal( String message )
```

Tudo mais na definição da classe, além dessas cinco declarações de método, é implementação. Lembre-se de que a implementação define como algo é feito. Aqui, o ‘como’ é o fato de que Log imprime na tela. Entretanto, a interface oculta completamente o ‘como’. Em vez disso, a interface define um contrato com o mundo exterior. Por exemplo, `public void debug( String message )` é uma maneira de dizer ao mundo exterior que, se você passar uma String, ela reportará uma mensagem de depuração.

O que é importante notar é o que a interface não diz. `debug()` não diz que imprimirá na tela. Em vez disso, o que é feito com a mensagem é deixado para a implementação. A implementação poderia escrever na tela, descarregar em um arquivo, gravar em um banco de dados ou enviar uma mensagem para um cliente monitorado pela rede.

2

## Público, privado e protegido

Você pode ter notado que a interface pública não inclui

```
private void print( String message, String severity ).
```

Em vez disso, a classe Log restringe o acesso a `print()`.

O que aparece e o que não aparece na interface pública é governado por diversas palavras-chave. Cada linguagem OO define seu próprio conjunto de palavras-chave, mas fundamentalmente essas palavras-chave acabam tendo efeitos semelhantes.

A maioria das linguagens OO suporta três níveis de acesso:

- PÚBLICO — Garante o acesso a todos os objetos.
- PROTEGIDO — Garante o acesso à instância, ou seja, para aquele objeto, e para todas as subclasses (mais informações sobre subclasses no Dia 4, “Herança: obtendo algo para nada”).
- PRIVADO — Garante o acesso apenas para a instância, ou seja, para aquele objeto.

O nível de acesso que você escolhe é muito importante para seu projeto. Todo comportamento que você queira tornar visível para o mundo, precisa ter acesso público. Tudo que você quiser ocultar do mundo exterior precisa ter acesso protegido ou privado.

## Por que você deve encapsular?

Quando usado cuidadosamente, o encapsulamento transforma seus objetos em componentes plugáveis. Para que outro objeto use seu componente, ele só precisa saber como usar a interface pública do componente. Tal independência tem três vantagens importantes:

- Independência significa que você pode reutilizar o objeto em qualquer parte. Quando você encapsular corretamente seus objetos, eles não estarão vinculados a nenhum programa em particular. Em vez disso, você pode usá-los sempre que seu uso fizer sentido. Para usar o objeto em qualquer lugar, você simplesmente exerce sua interface.
- O encapsulamento permite que você torne transparentes as alterações em seu objeto. Desde que você não altere sua interface, todas as alterações permanecerão transparentes para

aqueles que estiverem usando o objeto. O encapsulamento permite que você atualize seu componente, forneça uma implementação mais eficiente ou corrija erros — tudo isso sem ter de tocar nos outros objetos de seu programa. Os usuários de seu objeto se beneficiarão automaticamente de todas as alterações que você fizer.

- Usar um objeto encapsulado não causará efeitos colaterais inesperados entre o objeto e o restante do programa. Como o objeto tem implementação independente, ele não terá nenhuma outra interação com o restante do programa, além de sua interface.

Agora, você está em um ponto onde podemos falar sobre algumas generalidades a respeito do encapsulamento. Você viu que o encapsulamento permite escrever componentes de software com implementações independentes. As três características do encapsulamento eficaz são:

- Abstração
- Ocultação da implementação
- Divisão de responsabilidade

Vamos ver mais profundamente cada característica, para aprender a melhor maneira de obter o encapsulamento.

## **Abstração: aprendendo a pensar e programar de forma abstrata**

Embora as linguagens OO estimulem o encapsulamento, elas não o garantem. É fácil construir código dependente e frágil. O encapsulamento eficaz vem apenas com um projeto cuidadoso, abstração e experiência. Um dos primeiros passos para o encapsulamento eficaz é aprender como abstrair software e os conceitos subjacentes eficientemente.

### **O que é abstração?**

Abstração é o processo de simplificar um problema difícil. Quando começa a resolver um problema, você não se preocupa com cada detalhe. Em vez disso, você o simplifica, tratando apenas dos detalhes pertinentes a uma solução.

Imagine que você tenha de escrever um simulador de fluxo de tráfego. É possível que você modele classes para sinais de trânsito, veículos, condições da pista, auto-estradas, ruas de mão dupla, ruas de mão única, condições climáticas etc. Cada um desses elementos afetaria o fluxo do tráfego. Entretanto, você não modelaria insetos e pássaros no sistema, mesmo que eles possam aparecer em uma via real. Além disso, você omitiria tipos específicos de carros. Você simplifica o mundo real e inclui apenas as partes que realmente afetam a simulação. Um carro é muito importante para a simulação, mas o fato de ser um Cadillac ou fazer com que o carro controle seu nível de combustível é supérfluo para a simulação de tráfego.

A abstração tem duas vantagens. Primeiro, ela permite que você resolva um problema facilmente. Mais importante, a abstração o ajuda a obter reutilização. Muitas vezes, os componentes de

software são demasiadamente especializados. Essa especialização, combinada com uma interdependência desnecessária entre os componentes, torna difícil reutilizar um código existente em outra parte. Quando possível, você deve se esforçar por criar objetos que possam resolver um domínio inteiro de problemas. A abstração permite que você resolva um problema uma vez e depois use essa solução por todo o domínio desse problema.



**NOTA**  
Embora seja desejável escrever código abstrato e evitar uma especialização demasiada, é duro escrever código abstrato, especialmente quando você começa a praticar a POO.

Existe uma linha tênue entre muita e pouca especialização. A linha pode ser discernida apenas com a experiência. Entretanto, você precisa saber desse poderoso conceito.

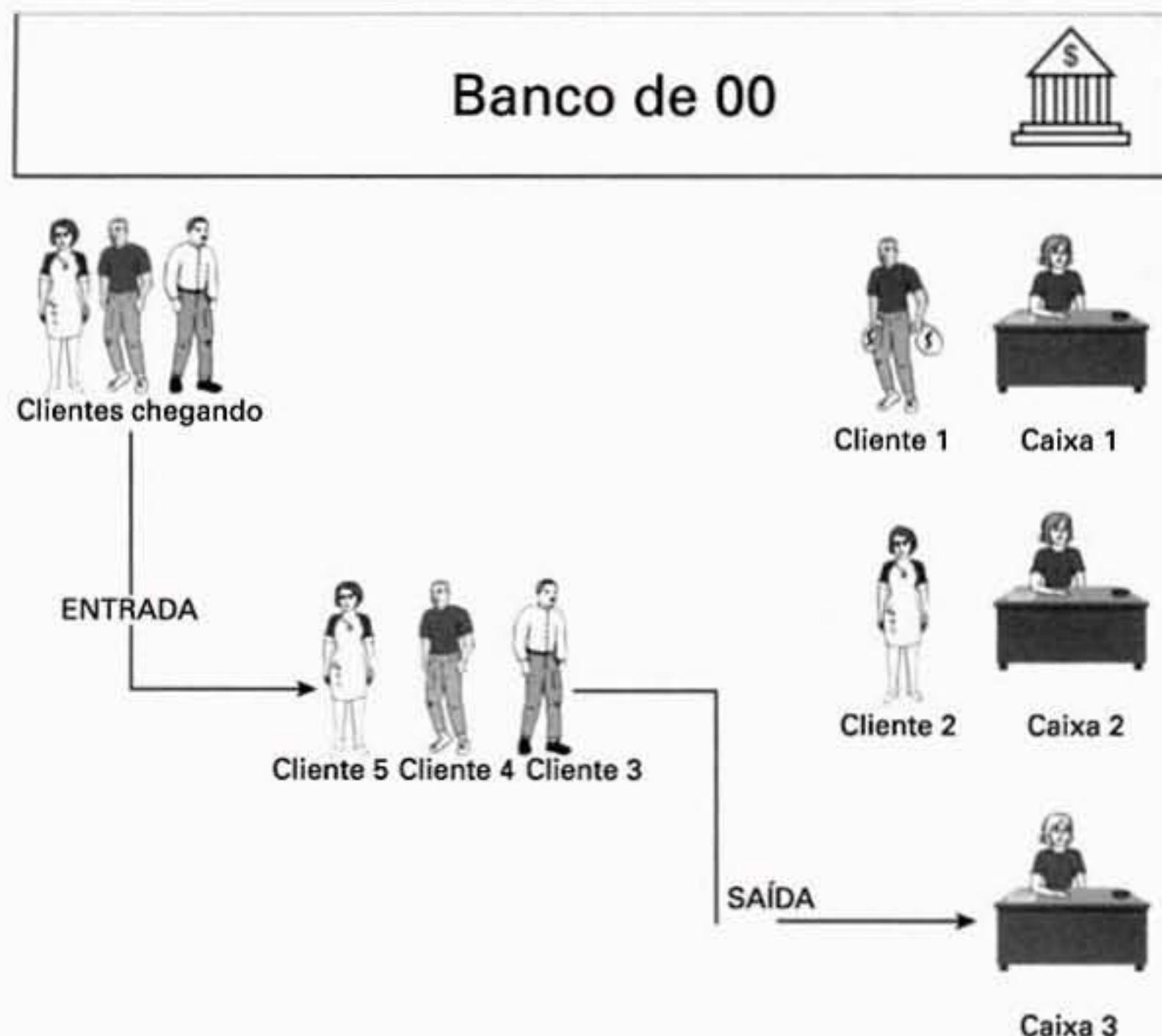
2

## Dois exemplos de abstração

Considere dois exemplos.

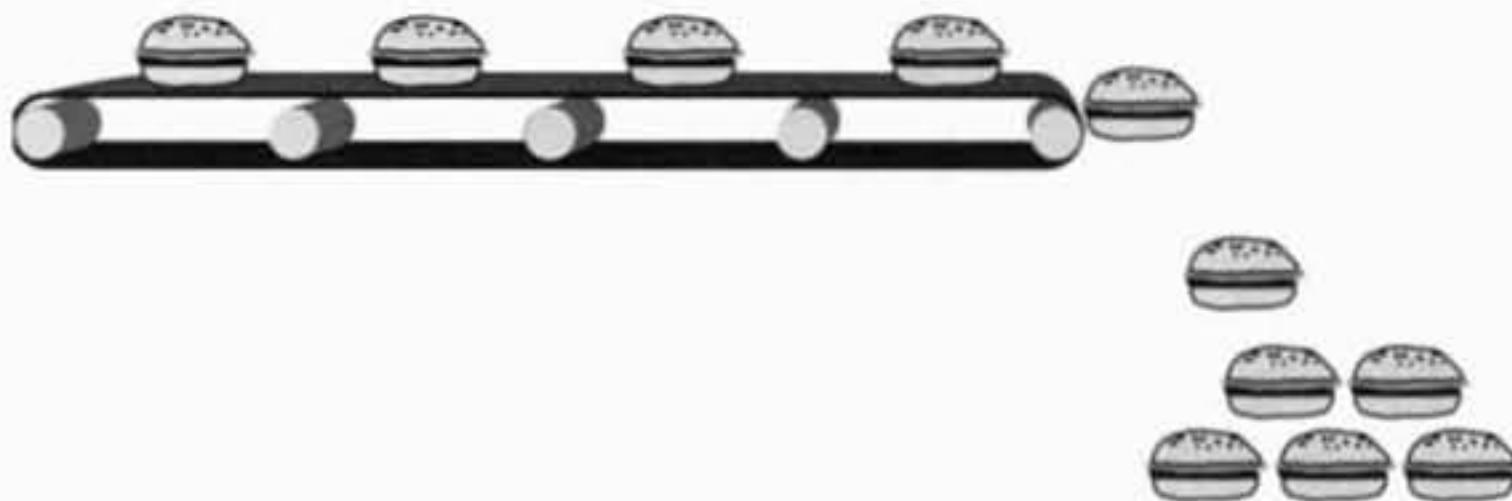
Primeiro, imagine pessoas em fila em um banco, esperando por um caixa. Assim que um caixa se torna disponível, a primeira pessoa da fila avança para a janela aberta. As pessoas sempre deixam a fila na ordem de que o primeiro a entrar é o primeiro a sair (FIFO): essa ordem é sempre mantida.

**FIGURA 2.2**  
*Uma fila em um banco.*



Segundo, considere um estabelecimento de sanduíches do tipo *fast food*. Quando um novo sanduíche fica pronto, ele é colocado atrás do último sanduíche que está no escaninho; veja a Figura 2.3. Desse modo, o primeiro sanduíche retirado também é o mais antigo. FIFO é o esquema do restaurante.

**FIGURA 2.3**  
*Sanduíches ficando prontos.*



Embora cada um desses exemplos seja específico, você pode encontrar uma descrição genérica que funcione em cada situação. Em outras palavras, você pode chegar a uma abstração.

Cada domínio é um exemplo de fila do tipo primeiro a entrar, primeiro a sair. Não importa quais tipos de elementos apareçam na fila. O que importa é que os elementos entram no final da fila e saem dela a partir da frente, conforme ilustrado na Figura 2.4.

**FIGURA 2.4**  
*Uma abstração de ambos os domínios.*



Abstraindo os domínios, você pode criar uma fila uma vez e reutilizá-la em qualquer problema que modele um domínio onde exista uma ordenação FIFO de elementos.

## Abstração eficaz

Neste ponto, você pode formular algumas regras para a abstração eficaz:

- Trate do caso geral e não do caso específico.
- Ao confrontar vários problemas diferentes, procure o que for comum a todos. Tente ver um conceito e não um caso específico.
- Não se esqueça de que você tem um problema a resolver. A abstração é valiosa, mas não descuide do problema na esperança de escrever código abstrato.
- A abstração pode não estar prontamente aparente. A abstração pode não saltar à sua frente na primeira, segunda ou terceira vez que você resolver um problema que está sujeito a ser abstraído.

- Prepare-se para a falha. É quase impossível escrever uma abstração que funcione em todas as situações. Você verá por que, posteriormente ainda hoje.

**ALERTA**

Não caia na paralisia da abstração. Resolva os problemas que você encontrar primeiro. Veja a abstração como um bônus e não como o objetivo final. Caso contrário, você vai se deparar com a possibilidade de prazos finais perdidos e abstração incorreta. Existem ocasiões para abstrair e ocasiões em que a abstração não é apropriada.

Uma boa regra geral é abstrair algo que você tiver implementado três vezes de maneira análoga. À medida que você ganhar experiência, aprenderá a escorrer a abstração mais rapidamente.

**NOTA****2**

Nem sempre você pode reconhecer oportunidades para uma abstração. Talvez você tenha de resolver um problema várias vezes, antes que uma abstração se torne aparente. Às vezes, diferentes situações ajudam a encontrar uma abstração eficaz e, mesmo então, a abstração pode precisar de alguma conversão. A abstração pode demorar a amadurecer.

A abstração pode tornar um componente encapsulado mais reutilizável, pois ele está personalizado para um domínio de problemas e não para um uso específico. Entretanto, há mais coisas importantes quanto ao encapsulamento do que a simples reutilização de componentes. O encapsulamento também é importante por ocultar os detalhes internos. O tipo abstrato de dados é um bom lugar para ver em seguida, na busca do encapsulamento eficaz.

## Guardando seus segredos através da ocultação da implementação

A abstração é apenas uma característica do encapsulamento eficaz. Você pode escrever código abstrato que não é encapsulado. Em vez disso, você também precisa ocultar as implementações internas de seus objetos.

A ocultação da implementação tem duas vantagens:

- Ela protege seu objeto de seus usuários.
- Ela protege os usuários de seu objeto do próprio objeto.

Vamos explorar a primeira vantagem — proteção do objeto.

## Protegendo seu objeto através do TAD (Abstract Data Type – Tipo Abstrato de Dados)

O Tipo Abstrato de Dados (TAD) não é um conceito novo. Os TADs, junto com a própria OO, cresceu a partir da linguagem de programação Simula, introduzida em 1966. Na verdade, os TADs são decididamente não OO; em vez disso, eles são um subconjunto da OO. Entretanto, os TADs apresentam duas características interessantes: *abstração* e *tipo*. É essa idéia de tipo que é importante, pois sem ela, você não pode ter um verdadeiro encapsulamento.



**NOTA** O verdadeiro encapsulamento é imposto em nível de linguagem, através de construções internas da linguagem. Qualquer outra forma de encapsulamento é simplesmente um acordo de cavalheiros, que é facilmente malogrado. Os programadores o contornarão porque podem fazer isso!

### Novo Termo

Um *TAD* é um conjunto de dados e um conjunto de operações sobre esses dados. Os TADs permitem que você defina novos tipos na linguagem, ocultando dados internos e o estado, atrás de uma interface bem definida. Essa interface apresenta o TAD como uma única unidade atômica.

Os TADs são uma maneira excelente de introduzir encapsulamento, pois eles o liberam de considerar o encapsulamento sem a bagagem extra da herança e do polimorfismo: você pode se concentrar no encapsulamento. Os TADs também permitem que você explore a noção de tipo. Uma vez que o tipo seja entendido, é fácil ver que a OO oferece uma maneira natural de estender uma linguagem, definindo tipos personalizados do usuário.

## O que é um tipo?

Quando programar, você criará diversas variáveis e atribuirá valores para elas. Os tipos definem as diferentes espécies de valores que estão disponíveis para seus programas. Você usa tipos para construir seu programa. Exemplos de alguns tipos comuns são integers (inteiros), longs (inteiros longos) e floats (reais). Essas definições de tipo informam exatamente quais espécies de tipos estão disponíveis, o que os tipos fazem e o que você pode fazer com eles.

Usaremos a seguinte definição de tipo:

### Novo Termo

Os *tipos* definem as diferentes espécies de valores que você pode usar em seus programas. Um tipo define o domínio a partir do qual seus valores válidos podem ser extraídos. Para inteiros positivos, são os números sem partes fracionárias e que são maiores ou iguais a 0. Para tipos estruturados, a definição é mais complexa. Além do domínio, a definição de tipo inclui quais operações são válidas no tipo e quais são seus resultados.



O tratamento formal de *tipo* está fora dos objetivos de um livro sobre POO para iniciantes.

Os tipos são unidades atômicas da computação. Isso significa que um tipo é uma unidade independente. Pegue o inteiro, por exemplo. Quando soma dois inteiros, você não pensa sobre a adição de bits individuais; você pensa apenas a respeito da adição de dois números. Mesmo que os bits representem o inteiro, a linguagem de programação apresenta o inteiro apenas como um número para o programador.

Pegue o exemplo Item do Dia 1. A criação da classe Item adiciona um novo tipo em seu vocabulário de programação. Em vez de pensar a respeito de uma id, uma descrição e um preço de produto como entidades separadas, provavelmente regiões desconectadas da memória ou variáveis, você pensa simplesmente em termos de Item. Assim, os tipos permitem representar estruturas complexas em um nível mais simples e mais conceitual. Eles o protegem dos detalhes desnecessários. Isso o libera para trabalhar no nível do problema, em vez de trabalhar no nível da implementação.

2

Embora seja verdade que um tipo protege o programador dos detalhes subjacentes, os tipos oferecem uma vantagem ainda mais importante. A definição de um tipo protege o tipo do programador. Uma definição de tipo garante que qualquer objeto que interaja com o tipo, o faça de maneira correta, consistente e segura. As restrições impostas por um tipo impedem os objetos de terem uma interação inconsistente, possivelmente destrutiva. A declaração de um tipo impede que o tipo seja usado de maneira não projetada e arbitrária. Uma declaração de tipo garante o uso correto.

Sem uma definição clara das operações permitidas, um tipo poderia interagir com outro tipo da maneira que quisesse. Freqüentemente, tal interação indefinida pode ser destrutiva.

Pense novamente no Item do Dia 1. Imagine que tivéssemos alterado um pouco a definição de Item:

```
public class UnencapsulatedItem {  
    //...  
  
    public double unit_price;  
    public double discount;    // uma porcentagem de desconto a ser aplicada no preço  
    public int quantity;  
    public String description;  
    public String id;  
}
```

Você notará que todas as variáveis internas agora estão publicamente disponíveis. E se alguém escrevesse o programa a seguir, usando o novo UnencapsulatedItem:

```
public static void main( String [] args ) {  
    UnencapsulatedItem monitor =  
        new UnencapsulatedItem( "electronics-012",
```

```
        "17\" SVGA Monitor",
        1,
        299.00 );

monitor.discount = 1.25; // inválido, o desconto deve ser menor que 100!

double price = monitor.getAdjustedTotal();

System.out.println( "Incorrect Total: $" + price );

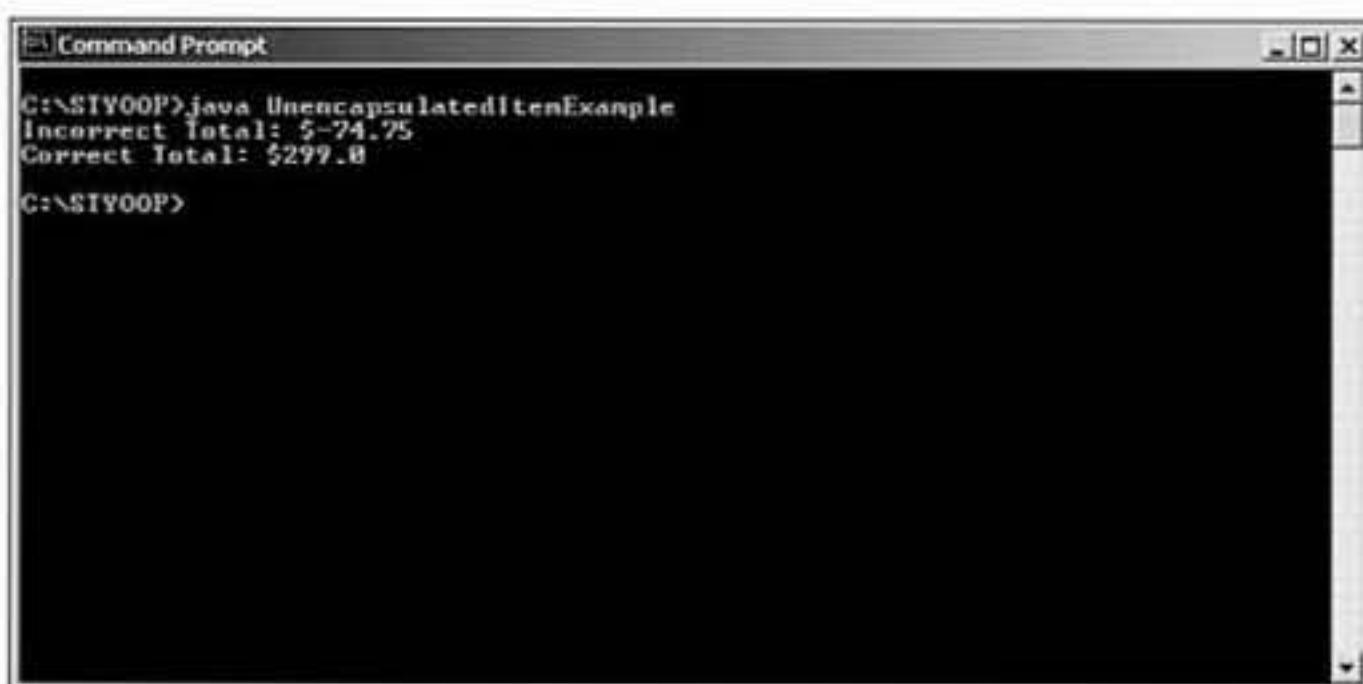
monitor.setDiscount( 1.25 ); // inválido
                           // entretanto, o configurador capturará o erro

price = monitor.getAdjustedTotal();

System.out.println( "Correct Total:$" + price );
}
```

A Figura 2.5 mostra o que acontece quando você executa o método main().

**FIGURA 2.5**  
*Um total inválido.*



Abrindo o tipo `UnencapsulatedItem` para acesso liberado, outros podem chegar e deixar uma instância de `UnencapsulatedItem` em um estado inválido. Nesse caso, `main()` cria um `UnencapsulatedItem` e, em seguida, aplica diretamente um desconto inválido. O resultado é um preço negativo ajustado!

Os TADs são ferramentas de encapsulamento valiosas, pois eles permitem que você defina novos tipos da linguagem que são seguros de usar. Assim como novas palavras são acrescentadas no idioma inglês a cada ano, um TAD permite que você crie novas palavras de programação, onde você precisa expressar uma nova idéia.

Uma vez definido, você pode usar um novo tipo como qualquer outro. Assim como você pode passar um inteiro para um método, também pode passar um TAD para um método. Isso é conhecido como sendo um objeto de primeira classe. Você pode passar objetos de primeira classe como parâmetros.

**Novo Termo**

Um *objeto de primeira classe* é aquele que pode ser usado exatamente da mesma maneira que um tipo interno.

**Novo Termo**

Um *objeto de segunda classe* é um tipo de objeto que você pode definir, mas não necessariamente usar, como faria com um tipo interno.

## Um exemplo de TAD

Vamos considerar o exemplo da fila abstrata apresentado anteriormente. Ao implementar uma fila, você dispõe de várias escolhas de implementação. Você pode implementar a fila como uma lista encadeada, uma lista duplamente encadeada, um vetor ou uma matriz. Entretanto, a implementação subjacente não muda o comportamento definido da fila. Independentemente da implementação, os itens ainda entram e saem de acordo com a ordem FIFO.

A fila é uma forte candidata para um TAD. Você já viu que não precisa conhecer a implementação subjacente para usar a fila. Na verdade, você não quer ter de se preocupar com a implementação. Se você não transformar a fila em um TAD, cada objeto que precisa de uma fila precisará reimplementar a estrutura de dados. Cada objeto que quiser manipular os dados na fila precisará entender a implementação e entender como fazer para interagir com ela corretamente. Você já viu os perigos do uso não projetado!

Em vez disso, você deve construir a fila como um TAD. Um TAD de fila bem encapsulado garante acesso consistente e seguro aos dados.

Ao sentar-se para projetar um TAD, você precisa perguntar-se o que o TAD faz. Neste caso, o que você pode fazer com uma fila? Você pode:

- Colocar elementos na fila: enqueue
- Remover elementos da fila: dequeue
- Consultar o estado da fila
- Ver o elemento da frente sem removê-lo: peek

Cada um dos métodos se transformará em uma entrada na interface pública de Queue.

Você também precisa nomear o TAD. Neste caso, o nome do TAD é Queue. O TAD é definido como segue:

```
public interface Queue {  
    public void enqueue( Object obj );  
    public Object dequeue();  
    public boolean isEmpty();  
    public Object peek();  
}
```

Note que a interface da fila não diz nada a respeito de como a fila contém seus dados internos. Note também que a interface não fornece acesso liberado a qualquer dos dados internos. Todos os detalhes ficam ocultos.

Em vez disso, agora você tem um novo tipo, uma fila. Agora, você pode usar esse tipo em qualquer um de seus programas.

Como se trata de um objeto de primeira classe, você pode usar a fila como parâmetro. Você pode tratar a abstração como uma unidade, pois todas as partes são independentes. Isso é poderoso; pois permite ao programador ser mais expressivo. Em vez de pensar em termos de ponteiros e listas, o programador pode pensar em um nível muito mais alto: em termos do problema a ser resolvido. Quando o programador diz fila, a palavra inclui todos os detalhes de uma lista e de um ponteiro, mas também permite que ele ignore esses detalhes e pense em uma estrutura de dados FIFO de alto nível.



**NOTA**  
Conforme você verá, à medida que continuar, um tipo poderia ser composto de outros tipos, através de restrições. Embora isso oculte os detalhes, também aumenta sua capacidade de se expressar. Os tipos que contêm outros tipos podem abranger muitos conceitos.

Por exemplo, quando você programa e diz int, o significado é muito simples; você simplesmente declarou um único inteiro. Entretanto, quando você diz Queue, sua declaração é muito mais expressiva. Há muito mais ocorrendo dentro de Queue do que dentro de int.

Vamos considerar a interface mais um pouco. Note que essa interface é muito genérica. Em vez de dizer que essa é uma fila de inteiros ou hamburgers, a interface simplesmente coloca e retira objetos da fila. Em Java, você pode tratar todos os objetos como Object. Entretanto, cada linguagem fornece seu próprio mecanismo semelhante. Declarando os parâmetros desse modo, você pode colocar qualquer objeto que queira na fila. Assim, essa definição torna o tipo Queue útil em muitas situações diferentes.



**ALERTA**  
As interfaces genéricas têm seus próprios perigos. Uma Queue de inteiros é muito exata. Você sabe que cada elemento em Queue é um inteiro. Entretanto, uma Queue de objetos é fracamente tipada. Quando você extrai um elemento, pode não saber qual é seu tipo.

Para um encapsulamento realmente eficaz, existem mais algumas características que você precisará tratar. Tocamos no aspecto da ocultação da implementação. Mas, e quanto ao outro lado da moeda — proteger os usuários de seus objetos?

## Protegendo outros de seus segredos através da ocultação da implementação

Até aqui, você viu que uma interface pode ocultar a implementação subjacente de um objeto. Quando oculta a implementação atrás de uma interface, você protege seu objeto de uso não pro-

jetado ou destrutivo. Proteger seu objeto de uso não projetado é uma vantagem da ocultação da implementação. Entretanto, existe outro lado na história: os usuários de seus objetos.

A ocultação da implementação leva a um projeto mais flexível, pois ela impede que os usuários de seus objetos se tornem fortemente acoplados à implementação subjacente dos objetos. Então, não apenas a ocultação da implementação protege seus objetos, como também protege aqueles que usam seus objetos, estimulando um código fracamente acoplado.

**Novo Termo**

O *código fracamente acoplado* é independente da implementação de outros componentes.

**Novo Termo**

O *código fortemente acoplado* é fortemente vinculado à implementação de outros componentes.

2

Você poderia estar se perguntando, “para que serve código fracamente acoplado?”

Quando um recurso aparece na interface pública de um objeto, todo mundo que usa o recurso se torna dependente do fato de ele existir. Se o recurso desaparecer repentinamente, você precisará alterar o código que tiver se desenvolvido de forma dependente a esse comportamento ou atributo.

**Novo Termo**

*Código dependente* é dependente da existência de determinado tipo. O código dependente é inevitável. Entretanto, existem graus para a dependência aceitável e para a superdependência.

Existem graus para a dependência. Você não pode eliminar a dependência totalmente. Entretanto, você deve se esforçar para minimizar a dependência entre objetos. Normalmente, você limita tal dependência programando uma interface bem definida. Os usuários só podem se tornar dependentes quanto ao que você decide colocar na interface. Entretanto, se alguma implementação do objeto se tornar parte da interface, os usuários do objeto poderão se tornar dependentes dessa implementação. Tal código fortemente acoplado elimina sua liberdade de alterar a implementação de seu objeto. Uma pequena alteração na implementação de seu objeto poderia necessitar de uma cascata de alterações por todos os usuários do objeto.



O encapsulamento e a ocultação da implementação não são mágica. Se você precisar alterar uma interface, precisará atualizar o código que é dependente da interface antiga. Ocultando os detalhes e escrevendo software para uma interface, você cria software que é fracamente acoplado.

O código fortemente acoplado anula o objetivo do encapsulamento: criar objetos independentes e reutilizáveis.

## Um exemplo real de ocultação da implementação

Um exemplo concreto de ocultação da implementação esclarecerá esta lição. Considere a seguinte definição de classe:

```
public class Customer {  
    // ...vários métodos de cliente ...  
    public Item [] items;    //este array contém todos os itens selecionados  
}
```

Um Customer contém itens selecionados. Aqui, Customer torna o array Item parte de sua interface externa:

```
public static void main( String [] args ) {  
    Customer customer = new Customer();  
  
    // ... seleciona alguns itens ...  
  
    // preço dos itens  
    double total = 0.0;  
    for( int i = 0; i < customer.items.length; i++ ) {  
        Item item = customer.items[i];  
        total = total + item.getAdjustedTotal();  
    }  
}
```

Esse método `main()` pega um cliente, adiciona alguns itens e totaliza o pedido. Tudo funciona, mas o que acontece se você quiser mudar a maneira como um Customer contém itens? Suponha que você quisesse introduzir uma classe Basket. Se você mudar a implementação, precisará atualizar todo o código que acessa diretamente o array Item.

Sem ocultação da implementação, você perde sua liberdade de melhorar seus objetos. No exemplo Customer, você deve tornar o array Item privado. Forneça acesso aos itens através de acessores.



### NOTA

A ocultação da implementação tem seus inconvenientes. Existem ocasiões em que você poderia precisar saber um pouco mais do que a interface pode informar.

No mundo da programação, você desejará uma caixa preta que funcione dentro de determinada tolerância ou que use a quantidade certa de precisão. Você poderia saber que precisa de inteiros de 64 bits, pois está tratando com números muito grandes. Ao definir sua interface, é importante não apenas fornecer uma interface, mas também documentar esses tipos de detalhes específicos sobre a implementação. Entretanto, assim como em qualquer outra parte da interface pública, uma vez que você declara um comportamento, não pode alterá-lo.

A ocultação da implementação permite que você escreva código que é independente e fracamente acoplado com outros componentes. O código fracamente acoplado é menos frágil e mais flexível para alterar. Um código flexível facilita a reutilização e o aprimoramento, pois as alterações em uma parte do sistema não afetarão outras partes não relacionadas.

**DICA**

Como você obtém ocultação da implementação eficaz e código fracamente acoplado?

Aqui estão algumas dicas:

- Só permita acesso ao seu TAD através de uma interface baseada em método. Tal interface garante que você não exponha informações sobre a implementação.
- Não forneça acesso involuntário a estruturas de dados internas, retornando ponteiros ou referências acidentalmente. Após alguém obter uma referência, a pessoa pode fazer tudo com ela.
- Nunca faça suposições sobre os outros tipos que você usa. A não ser que um comportamento apareça na interface ou na documentação, não conte com ele.
- Cuidado enquanto escrever dois tipos intimamente relacionados. Não programe accidentalmente em suposições e dependências.

2

## Divisão da responsabilidade: preocupando-se com seu próprio negócio

A ocultação da implementação evolui naturalmente para uma discussão sobre a divisão da responsabilidade. Na seção anterior, você viu como poderia desacoplar código ocultando detalhes da implementação. A ocultação da implementação é apenas um passo na direção da escrita de código fracamente acoplado.

Para ter realmente código fracamente acoplado, você também deve ter uma divisão da responsabilidade correta. Divisão da responsabilidade correta significa que cada objeto deve executar uma função — sua responsabilidade — e executá-la bem. A divisão da responsabilidade correta também significa que o objeto é coesivo. Em outras palavras, não faz sentido encapsular muitas funções aleatórias e variáveis. Elas precisam ter um forte vínculo conceitual entre si. Todas as funções devem trabalhar no sentido de uma responsabilidade comum.

**NOTA**

A ocultação da implementação e a responsabilidade andam lado a lado. Sem ocultação da implementação, a responsabilidade pode faltar em um objeto. É de responsabilidade do objeto saber como fazer seu trabalho. Se você deixar a implementação aberta para o mundo exterior, um usuário poderá começar a atuar diretamente na implementação — duplicando assim a responsabilidade.

Assim que dois objetos começam a fazer a mesma tarefa, você sabe que não tem uma divisão da responsabilidade correta. Quando você observar a existência de lógica redundante, precisará refazer seu código. Mas não se sinta mal; refazer o trabalho é uma parte esperada do ciclo de desenvolvimento OO. À medida que seus projetos amadurecerem, você encontrará muitas oportunidades para melhorá-los.

Vamos considerar um exemplo real de divisão da responsabilidades: o relacionamento entre gerente e programador.

Imagine que seu gerente venha até você, forneça as especificações de sua parte em um projeto e, em seguida, o deixe trabalhar. Ele sabe que você tem um trabalho a fazer e que sabe como fazer o melhor trabalho possível.

Agora, imagine que seu chefe não é tão esperto. Ele explica o projeto e pelo que você será responsável. Ele lhe garante que está lá para facilitar seu trabalho. Mas, quando você começa, ele puxa uma cadeira! Pelo resto do dia, seu chefe fica em cima de você e fornece instruções passo a passo, enquanto você codifica.

Embora o exemplo seja um tanto extremo, os programadores programam dessa maneira em seu código, o tempo todo. O encapsulamento é como o gerente eficiente. Como no mundo real, conhecimento e responsabilidade precisam ser delegados para aqueles que sabem como fazer o trabalho da melhor forma possível. Muitos programadores estruturaram seu código como um chefe autoritário trata seus funcionários. Esse exemplo é facilmente transportado para os termos da programação. Vamos considerar um exemplo assim:

```
public class BadItem {  
  
    private double unit_price;  
    private double adjusted_price;  
    private double discount;      // uma porcentagem de desconto para aplicar no  
                                // preço  
    private int    quantity;  
    private String description;  
    private String id;  
  
    public BadItem( String id, String description,  
                   int quantity, double price ) {  
        this.id = id;  
        this.description = description;  
  
        if( quantity >= 0 ) {
```

```
        this.quantity = quantity;
    }
else {
    this.quantity = 0;
}
    this.unit_price = price;
}

public double getUnitPrice() {
    return unit_price;
}

// aplica uma porcentagem de desconto no preço
public void setDiscount( double discount ) {
    if( discount <= 1.00 ) {
        this.discount = discount;
    }
}

public double getDiscount() {
    return discount;
}

public int getQuantity() {
    return quantity;
}

public void setQuantity( int quantity ) {
    this.quantity = quantity;
}

public String getProductID() {
    return id;
}

public String getDescription() {
    return description;
}

public double getAdjustedPrice() {
    return adjusted_price;
}

public void setAdjustedPrice( double price) {
    adjusted_price = price;
}
```

2

BadItem não contém mais a responsabilidade de calcular o preço ajustado. Então, como você gera um preço ajustado? Considere o método `main()` a seguir:

```
public static void main( String [] args ) {
    // cria os itens
    BadItem milk = new BadItem( "dairy-011", "1 Gallon Milk", 2, 2.50 );

    // aplica cupons
    milk.setDiscount( 0.15 );

    // obtém os preços ajustados
    double milk_price    = milk.getQuantity() * milk.getUnitPrice();
    double milk_discount = milk.getDiscount() * milk_price;
    milk.setAdjustedPrice( milk_price - milk_discount );

    System.out.println( "Your milk costs:\t $" + milk.getAdjustedPrice() );
}
```

Agora, em vez de simplesmente solicitar a Item seu preço ajustado, você precisa se comportar como o gerente ineficiente. Você precisa dizer ao objeto item o que fazer, passo a passo.

Ter de chamar várias funções para calcular o total ajustado, retira a responsabilidade do item e a coloca nas mãos do usuário. Retirar a responsabilidade dessa maneira é tão ruim quanto expor implementações internas. Você acaba com responsabilidade duplicada por todo o seu código. Cada objeto que desejar calcular o total ajustado precisará repetir a lógica encontrada no método `main()`.

Ao escrever suas interfaces, você precisa certificar-se de que não esteja simplesmente apresentando a implementação através de um conjunto de nomes diferente. Lembre da fila — você não quer métodos chamados `addObjectToList()`, `updateEndListPointer()` etc. Esses tipos de comportamentos são específicos da implementação. Em vez disso, você oculta a implementação, através dos comportamentos `enqueue()` e `dequeue()`, de nível mais alto (mesmo que, internamente, você possa atualizar ponteiros e adicionar o objeto a uma lista). Em termos de BadItem, você não desejará ter de chamar um método `calculateAdjustedPrice()`, antes de poder recuperar o preço ajustado através do método `getAdjustedPrice()`. Em vez disso, `getAdjustedPrice()` deve saber efetuar o cálculo.

Quando você tem objetos que não dividem corretamente a responsabilidade, acaba com código procedural, centrado nos dados. O método `main` para calcular o preço ajustado é muito procedural. Um método `main` que instruísse um objeto Queue em cada passo de seu processo `enqueue()` seria procedural. Se você simplesmente enviar uma mensagem para um objeto e confiar que ele faça seu trabalho, esse é o verdadeiro desenvolvimento orientado a objetos.

O encapsulamento está completamente ligado à ocultação de detalhes. A responsabilidade coloca o conhecimento de certos detalhes onde eles pertencem. É importante que os objetos tenham apenas uma ou um pequeno número de responsabilidades. Se um objeto tiver responsabilidades

demais, sua implementação se tornará muito confusa e difícil de manter e estender. Para alterar uma responsabilidade, você correrá o risco de alterar outro comportamento inadvertidamente, se um objeto contiver muitos comportamentos. Ele também centralizará muito conhecimento, que estaria melhor espalhado. Quando um objeto fica grande demais, ele quase se torna um programa completo e cai nas armadilhas procedurais. Como resultado, você se depara com todos os problemas que encontraria em um programa que não usasse nenhum encapsulamento.

Quando você verificar que um objeto executa mais de uma responsabilidade, precisará mover essa responsabilidade para seu próprio objeto.



**A ALERTA** A ocultação da implementação é apenas um passo para o encapsulamento eficiente. Sem divisões de responsabilidade corretas, você simplesmente acaba com uma lista de procedimentos.

2

Neste ponto, você pode melhorar a definição de encapsulamento.

**Novo Termo**

*Encapsulamento efetivo* é abstração mais ocultação da implementação mais responsabilidade.

Retire a abstração e você terá um código que não é reutilizável. Retire a ocultação da implementação e você ficará com um código fortemente acoplado e frágil. Retire a responsabilidade e você ficará com um código centrado nos dados, procedural, fortemente acoplado e descentralizado.

Sem todas as três partes, você não pode ter um encapsulamento efetivo, mas a falta de responsabilidade o deixa com a pior situação de todas: programação procedural em um ambiente orientado a objetos.

## Dicas e armadilhas do encapsulamento

Ao se aplicar encapsulamento, existem várias dicas a seguir e armadilhas a evitar.

### Dicas e armadilhas da abstração

Ao escrever uma classe, você pode ter problemas, se tentar trabalhar de forma abstrata demais. É impossível escrever uma classe que satisfaça todos os usuários e cada situação. Imagine que você tivesse de escrever um objeto pessoa para um sistema de folha de pagamento de uma empresa. Esse objeto pessoa vai ser muito diferente de um objeto pessoa no simulador de fluxo de tráfego que discutimos anteriormente.



**ALERTA**  
A abstração pode ser perigosa. Mesmo que você tenha abstraído algum elemento, ele poderá não funcionar em todos os casos. É muito difícil escrever uma classe que satisfaça as necessidades de todos os usuários. Não caia na fixação da abstração — resolva seus problemas primeiro!

Tudo se resume a fazer o suficiente para resolver o problema imediato. Incluir todos os detalhes necessários para o objeto pessoa funcionar nos dois contextos seria muito dispendioso. Isso pode provocar todos os problemas que você viu hoje, devido à responsabilidade embaralhada. Embora você possa ligar seu objeto pessoa às duas situações, ele não será mais um objeto pessoa abstrato. Você perde toda a simplificação que a abstração oferece.



**ALERTA**  
Não coloque em uma classe mais do que o necessário para resolver o problema. Não tente resolver todos os problemas; resolva o problema imediato. Sómente então você deverá procurar meios de abstrair o que fez.

É claro que existem ocasiões onde um problema é complexo, como um cálculo difícil ou uma simulação complicada. Estamos falando de complexidade do ponto de vista da responsabilidade. Quanto mais responsabilidades um objeto assume, mais complexo ele será e mais difícil será mantê-lo.



**DICA**  
Lembre-se de que adicionar uma nova classe em seu sistema é o mesmo que criar um novo tipo. Ter essa noção em mente ajuda a focalizar o que você está realmente fazendo. Ao falar sobre seu problema, você verá que estará falando em termos dos objetos e das interações e não de dados e métodos.

Finalmente, a verdadeira abstração só pode vir com o tempo.

A verdadeira abstração normalmente nasce a partir de usos reais e não do fato de um programador se sentar e decidir criar um objeto reutilizável. Como diz o ditado, a necessidade é a mãe da invenção. Os objetos funcionam da mesma maneira. Normalmente, você não pode sentar-se e escrever um objeto abstrato realmente reutilizável, logo na primeira vez. Em vez disso, os objetos reutilizáveis normalmente são derivados de um código amadurecido, que foi posto à prova e que enfrentou muitas alterações.

A verdadeira abstração também vem com a experiência. É um objetivo a ser buscado no domínio da POO.

## Dicas e armadilhas do TAD

A transformação de um TAD em uma classe é específica da linguagem. Entretanto, existem algumas considerações independentes da linguagem que você pode fazer a respeito das classes.

A maioria das linguagens OO fornece palavras-chave que o ajudam a definir classes encapsuladas. Primeiro, existe a própria definição de classe. A classe é como o TAD, mas com alguns recursos importantes, que você verá nos próximos dias.

Dentro de uma classe, normalmente você tem métodos e variáveis internas — os dados. O acesso a essas variáveis e métodos é fornecido por funções de acesso. Tudo na interface do TAD deve parecer fazer parte da interface pública do objeto.

2



ALERTA

Os TADs não são diretamente análogos à classe da OO. Eles não têm herança e recursos de polimorfismo. A importância desses recursos se tornará evidente quando você estudar o Dia 4 e o Dia 6, “Polimorfismo: aprendendo a prever o futuro”.

## Dicas da ocultação da implementação

O que expor e o que ocultar em sua interface nem sempre é fácil de decidir. Entretanto, podemos ter algumas considerações independentes da linguagem sobre o acesso. Apenas os métodos que você pretende que outros usem devem estar na interface pública. Os métodos que apenas o tipo usará devem estar ocultos. No exemplo da fila, `dequeue()` e `enqueue()` devem estar na interface pública. Entretanto, você deve ocultar métodos auxiliares, como `updateFrontPointer()` e `addToList()`.

Você sempre deve ocultar as variáveis internas, a não ser que elas sejam constantes. Achamos que elas não devem estar apenas ocultas, mas também acessíveis apenas para a própria classe. Você vai explorar esse conceito mais detidamente, no Dia 4. Abrir variáveis internas para acesso externo expõe sua implementação.



NOTA

Você pode abrir variáveis internas para uso externo apenas se sua linguagem tratar desses valores como trata métodos. O Delphi e o Borland C++ tratam de variáveis internas dessa maneira.

Se usuários externos puderem acessar métodos e valores sem saber que estão mexendo em um valor, então estará correto abri-los. Em tal linguagem, uma variável interna exposta seria igual a um método que não recebe parâmetros. Infelizmente, não são muitas linguagens OO que tratam valores e métodos da mesma maneira.

Finalmente, não crie interfaces que apresentam apenas a representação interna com um nome diferente. A interface deve apresentar comportamentos de alto nível.

## Como o encapsulamento atende os objetivos da programação orientada a objetos

O Dia 1 mostrou que o objetivo da programação orientada a objetos é produzir software:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível
6. Oportuno

O encapsulamento atende cada um desses objetivos:

- Natural: o encapsulamento permite que você divida a responsabilidade da maneira como as pessoas pensam naturalmente. Através da abstração, você fica livre para modelar o problema em termos do próprio problema e não em termos de alguma implementação específica. A abstração permite que você pense e programe no geral.
- Confiável: isolando a responsabilidade e ocultando a implementação, você pode validar cada componente individual. Quando um componente for validado, você poderá usá-lo com confiança. Isso possibilita testes de unidade completos. Você ainda precisa realizar testes de integração, para certificar-se de que o software construído funciona corretamente.
- Reutilizável: a abstração fornece código flexível e utilizável em mais de uma situação.
- Manutenível: o código encapsulado é mais fácil de manter. Você pode fazer qualquer alteração que queira na implementação de uma classe, sem danificar código dependente. Essas alterações podem incluir mudanças na implementação, assim como a adição de novos métodos na interface. Apenas as alterações que violam a semântica da interface exigirão mudanças no código dependente.
- Extensível: você pode mudar implementações sem danificar código. Como resultado, você pode fazer melhorias de desempenho e mudar funcionalidade sem danificar o código existente. Além disso, como a implementação fica oculta, o código que usar o componente será atualizado automaticamente, para tirar proveito de todos os novos recursos que você introduzir. Se você fizer tais alterações, certifique-se de fazer os testes de unidade novamente! Danificar um objeto pode ter um efeito de dominó por todo o código que use o objeto.
- Oportuno: dividindo seu software em partes independentes, você pode dividir a tarefa de criar as partes entre vários desenvolvedores, acelerando assim o desenvolvimento.

Uma vez que esses componentes estejam construídos e validados, eles não precisarão ser reconstruídos. Assim, o programador fica livre para reutilizar funcionalidade, sem ter de recriá-la.

## Advertências

Você pode estar pensando, “mas eu não preciso de OO para abstrair e encapsular meu código”. Quer saber? Você está certo — você não precisa de OO para ter código encapsulado. Os próprios TADs não são OO. É bastante possível ter encapsulamento em praticamente qualquer linguagem.

Entretanto, há um problema. Em outros tipos de linguagens, você freqüentemente precisa criar seus próprios mecanismos de encapsulamento. Como não existe nada na linguagem que o obrigue a respeitar seus padrões, você precisa estar atento. Você precisa se obrigar a seguir suas diretrizes. Você também terá de recriar suas diretrizes e seu mecanismo para cada programa que escrever.

Isso está bem para um desenvolvedor. Mas e para dois desenvolvedores? Dez? Uma empresa inteira? Quanto mais desenvolvedores são adicionados, mais difícil é ter todos na mesma página.

Uma verdadeira linguagem OO fornece um mecanismo para encapsulamento. Ela impõe o mecanismo de um modo que você não conseguiria sozinho. A linguagem encapsula os detalhes do mecanismo de encapsulamento do usuário. Uma linguagem OO fornece algumas palavras-chave. O programador simplesmente usa as palavras-chave e a linguagem cuida de todos os detalhes.

Ao trabalhar com os recursos fornecidos pela linguagem, a linguagem apresenta a todos os programadores o mesmo mecanismo consistente.

2

## Resumo

Agora que você entende de encapsulamento, pode começar a programar com objetos. Usando encapsulamento, você pode tirar proveito das vantagens da abstração, da ocultação da implementação e da responsabilidade em seu código diário.

Com a abstração, você pode escrever objetos que são úteis em várias situações. Se você ocultar corretamente a implementação de seu objeto, estará livre para fazer quaisquer melhorias que queira em seu código — a qualquer momento. Finalmente, se você dividir corretamente a responsabilidade entre seus objetos, evitará lógica duplicada e código procedural.

Se você fechar este livro agora e nunca mais consultá-lo, terá aprendido novas habilidades de OO suficientes para escrever componentes independentes. Entretanto, a história da OO não termina com o encapsulamento. Continue a ler e você aprenderá como tirar proveito de todos os recursos oferecidos pela POO.

## Perguntas e respostas

**P** **Como você sabe quais métodos deve incluir em uma interface?**

**R** É simples saber quais métodos deve incluir. Você precisa incluir apenas os métodos que tornam o objeto útil; os métodos de que você precisa para que outro objeto faça seu trabalho.

Quando você começar a escrever uma interface, desejará produzir a menor interface que ainda satisfaça suas necessidades. Torne sua interface o mais simples possível. Não inclua métodos que você ‘poderia’ precisar. Você pode adicioná-los quando realmente precisar deles.

Conheça certos tipos de métodos de conveniência. Se você fizer um objeto conter outros objetos, normalmente desejará evitar a criação de métodos que simplesmente encaminham uma chamada de método para um dos objetos contidos.

Por exemplo, digamos que você tenha um objeto carrinho de compras que contenha itens. Você não deve adicionar um método de conveniência no carrinho, que consultará um item para saber seu preço e retorná-lo. Em vez disso, você deve ter um método que permita obter o item. Quando você tiver o item, poderá solicitar o preço em si.

**P** **Você mencionou as palavras-chave `public`, `protected` e `private`. Existem outros modificadores de acesso?**

**R** Cada linguagem define seus modificadores de acesso de sua própria maneira. Entretanto, a maioria das linguagens OO define esses três níveis. A linguagem Java também tem um modificador de acesso do pacote padrão. Você especifica esse nível omitindo um modificador. Esse nível restringe o acesso a apenas as classes do mesmo pacote. Para mais informações sobre pacotes, consulte o Apêndice B, “Resumo do Java”.

**P** **Os modificadores de acesso também têm o papel de mecanismo de segurança?**

**R** Não. Os modificadores de acesso só restringem o modo como outros objetos podem interagir com determinado objeto. Os modificadores não têm nada a ver com a segurança do computador.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Como o encapsulamento atinge os objetivos da programação orientada a objetos?
2. Defina abstração e dê um exemplo demonstrando abstração.
3. Defina implementação.
4. Defina interface.
5. Descreva a diferença entre interface e implementação.
6. Por que a divisão clara da responsabilidade é importante para o encapsulamento eficaz?

7. Defina tipo.
8. Defina TAD (Tipo Abstrato de Dados).
9. Como você obtém ocultação da implementação eficaz e código fracamente acoplado?
10. Quais são alguns dos perigos inerentes à abstração?

## Exercícios

2

1. Considere a estrutura de dados de pilha clássica. Uma pilha é uma estrutura “Último a entrar, primeiro a sair” (LIFO). Ao contrário de uma fila FIFO, você só pode adicionar e remover elementos a partir da mesma extremidade de uma pilha. Assim como em uma fila, uma pilha permite que você verifique se ela está vazia e escolha o primeiro elemento que pode remover.

Defina um TAD para a classe pilha.

2. Pegue o TAD de pilha do Exercício 1 e esboce uma implementação. Quando terminar, defina uma segunda implementação.
3. Examine os exercícios 1 e 2. A interface que você projetou no Exercício 1 era adequada para as duas implementações formuladas no Exercício 2? Se assim foi, quais vantagens a interface forneceu? Caso contrário, o que faltou na interface original?

PÁGINA EM BRANCO

# SEMANA 1

## DIA 3

### Encapsulamento: hora de escrever algum código

Ontem, você aprendeu tudo sobre encapsulamento. Ao iniciar as lições de hoje, você deverá ter uma boa idéia do que é encapsulamento, como aplicá-lo eficazmente e em quais erros comuns deve prestar atenção. Neste ponto, o que você não tem é experiência prática com a técnica. É claro que você conhece a teoria, mas nada melhor do que se aprofundar no código. Para o restante do dia, você completará vários laboratórios que deverão cimentar as lições do Dia 2.

Hoje você aprenderá:

- Como configurar o ambiente Java
- Sobre as classes básicas
- Como implementar o encapsulamento
- A respeito do pacote de primitivas Java

### Laboratório 1: configurando o ambiente Java

Você usará a linguagem de programação Java para completar todos os laboratórios desta semana, assim como o projeto final. Para programar em Java, você precisa obter uma versão do Java 2. Para completar estes laboratórios, você precisa ter pelo menos a versão 1.2 do SDK.

Se você ainda não tem, terá de fazer download e instalar um kit de desenvolvimento agora. Existem muitos kits de desenvolvimento diferentes disponíveis. Entretanto, você pode obter facilmente a versão mais recente, a partir do endereço <http://www.javasoft.com/j2se/>.

A Sun suporta três plataformas principais: Solaris, Linux e Windows. Quando este livro estava sendo produzido, a versão mais recente da Sun era Java 2 SDK, Standard Edition, v 1.3.

A IBM também oferece vários kits de desenvolvimento, no endereço <http://www.ibm.com/java/jdk/index.html>.

Além das plataformas suportadas pela Sun, a IBM também fornece suporte a várias plataformas, como OS/2, AS/400 e AIX.

Cada kit de desenvolvimento vem com instruções de instalação adequadas. Siga essas instruções para instalar o kit em sua máquina de desenvolvimento. Você também pode consultar o Apêndice B, “Resumo do Java”, para obter mais ajuda.

Você também pode optar por usar um popular IDE Java, como Forte, JBuilder ou Visual Age for Java. Estes exemplos e laboratórios também funcionarão nesses ambientes.

Este livro pressupõe uma familiaridade básica com programação, mas você não precisa de um conhecimento profundo de Java para completar estes laboratórios. Se você precisar de alguma ajuda para conhecer os fundamentos da linguagem Java, consulte o Apêndice B.

## Exposição do problema

Na verdade, você precisa do boi antes do carro. Antes de poder programar, você precisa obter e configurar um ambiente de desenvolvimento Java. Se você ainda não fez isso, obtenha um kit de desenvolvimento Java e siga os passos delineados no Apêndice B para instalá-lo. Uma vez instalado, o apêndice o conduzirá na configuração do caminho das classes, assim como na compilação e execução de seu primeiro programa Java. Quando você concluir este laboratório, saberá se sua instalação de Java funciona. Você também saberá tudo que precisa para compilar e executar seus programas Java.

## Laboratório 2: classes básicas

É muito importante que você se recorde das lições dos dias 1 e 2, enquanto escreve suas primeiras classes.

No Dia 1, você aprendeu alguns fundamentos sobre classes e objetos. O Dia 2 mostrou como você pode se beneficiar do encapsulamento para produzir objetos bem definidos.

A biblioteca de classes Java contém um rico conjunto de estruturas de dados clássicas, como listas e tabelas hashing. Considere a classe DoubleKey da Listagem 3.1.

**LISTAGEM 3.1** DoubleKey.java

```
public class DoubleKey {  
  
    private String key1, key2;  
  
    // um construtor sem argumentos  
    public DoubleKey() {  
        key1 = "key1";  
        key2 = "key2";  
    }  
  
    // um construtor com argumentos  
    public DoubleKey( String key1, String key2 ) {  
        this.key1 = key1;  
        this.key2 = key2;  
    }  
  
    // acessor  
    public String getKey1() {  
        return key1;  
    }  
  
    // mutante  
    public void setKey1( String key1 ) {  
        this.key1 = key1;  
    }  
  
    // acessor  
    public String getKey2() {  
        return key2;  
    }  
  
    // mutante  
    public void setKey2( String key2 ) {  
        this.key2 = key2;  
    }  
  
    // igual e código hashing omitidos por brevidade  
}
```

3

Quando coloca um objeto em qualquer implementação de `java.util.Map`, você pode especificar qualquer objeto como chave para esse objeto. Quando precisa recuperar um objeto, você simplesmente usa a chave para recuperar o valor. `DoubleKey` permite que você use hashing em duas chaves `String`, em vez de em uma.

Você notará que DoubleKey tem dois construtores:

```
public DoubleKey() {  
    key1 = "key1";  
    key2 = "key2";  
}  
  
public DoubleKey( String key1, String key2 ) {  
    this.key1 = key1;  
    this.key2 = key2;  
}
```

Os construtores aparecem em duas formas: aqueles sem argumentos (construtores *noargs*) e aqueles com argumentos.

**Novo Termo** *Construtores noargs* são construtores que não recebem nenhum argumento.



*Construtor noarg* é um termo Java. O equivalente em C++ é *construtor padrão*.

Os construtores sem argumento instanciam um objeto com valor padrão, enquanto aqueles que aceitam argumentos usam os argumentos para inicializar o estado interno dos objetos.

`public DoubleKey()` é um exemplo de construtor noarg, enquanto `public DoubleKey( String key1, String key2 )` aceita argumentos.

Conforme o Dia 1 ensinou, métodos como `public String getKey1()` e `public String getKey2()` são conhecidos como acessores, pois eles permitem que você acesse os valores internos do objeto.



O mundo Java reconhece dois tipos de acessores: de configuração e de obtenção. Os de *configuração* permitem que você configure uma variável de instância, enquanto os de *obtenção* permitem que você leia uma variável de instância.

A Sun Microsystems desenvolveu uma convenção de atribuição de nomes em torno de acessores de configuração e de obtenção, conhecida como JavaBean Design Patterns. JavaBeans é uma maneira padronizada de escrever seus componentes. Se seus componentes obedecem a esse padrão, você pode conectá-los em qualquer IDE compatível com JavaBean. Tal IDE poderia permitir a construção visual de seus programas, usando os beans.

As convenções de atribuição Java são simples. A convenção JavaBean para atribuição de nomes para acessores de obtenção e de configuração é:

```
public void set<VariableName>( <type> value )  
public <type> get<VariableName>()
```

onde `<type>` é o tipo da variável de instância e `<VariableName>` é o nome da variável de instância.

Tome como exemplo um objeto Pessoa. Um objeto Pessoa tem um nome. Os acessores de obtenção e de configuração do nome poderiam ter o seguinte formato:

```
public void setName( String name )
public String getName()
```

Finalmente, você chama métodos como `public void setKey1( String key1 )` e `public void setKey2( String key2 )` de mutantes, pois eles permitem alterar o estado interno do objeto.

DoubleKey demonstra o uso correto do encapsulamento. Empregando uma interface bem definida, DoubleKey oculta sua implementação do mundo exterior. DoubleKey também é bastante abstrata. Você pode reutilizar DoubleKey onde precisar usar hashing com duas chaves String. Finalmente, DoubleKey divide corretamente a responsabilidade, fornecendo apenas os métodos necessários para atuar como uma chave.

## Exposição do problema

No Dia 2, você viu um Banco OO. No Banco OO, os clientes entram em uma fila, enquanto esperam por um caixa. Mas não se preocupe, você não vai escrever uma classe Queue. A linguagem Java tem bastante suporte interno para estruturas de dados clássicas. Em vez disso, você vai programar uma classe de conta — a linguagem Java ainda deixa o programador com alguns trabalhos a fazer.

Seja essa uma conta corrente, uma conta poupança ou uma conta de mercado financeiro, todas elas têm algumas características compartilhadas. Todas as contas têm um saldo. Uma conta também permitirá que você deposite valores, saque valores e consulte o saldo.

Hoje, você vai escrever uma classe de conta. O Laboratório 2 vem completo, com uma classe Teller. A classe Teller tem um método `main()` que você usará para testar a implementação de sua conta.

A classe Teller espera uma interface pública específica. Aqui estão as regras:

- Você deve chamar a classe conta de Account.
- A classe deve ter os dois construtores a seguir:

```
public Account()
public Account( double initial_deposit )
```

O construtor noargs configurará o saldo inicial como 0.00. O segundo construtor configurará o saldo inicial como `initial_deposit`.

- A classe deve ter os três métodos a seguir. O primeiro método credita na conta o valor de funds:

```
public void depositFunds( double funds )
```

O método seguinte debita na conta o valor de funds:

```
public double withdrawFunds( double funds )
```

Entretanto, `withdrawFunds()` não deve permitir um saque a descoberto. Em vez disso, se `funds` for maior que o saldo, apenas debita o resto do saldo. `withdrawFunds()` deve retornar a real quantidade sacada da conta.

O terceiro método recupera o saldo corrente da conta:

```
public double getBalance()
```

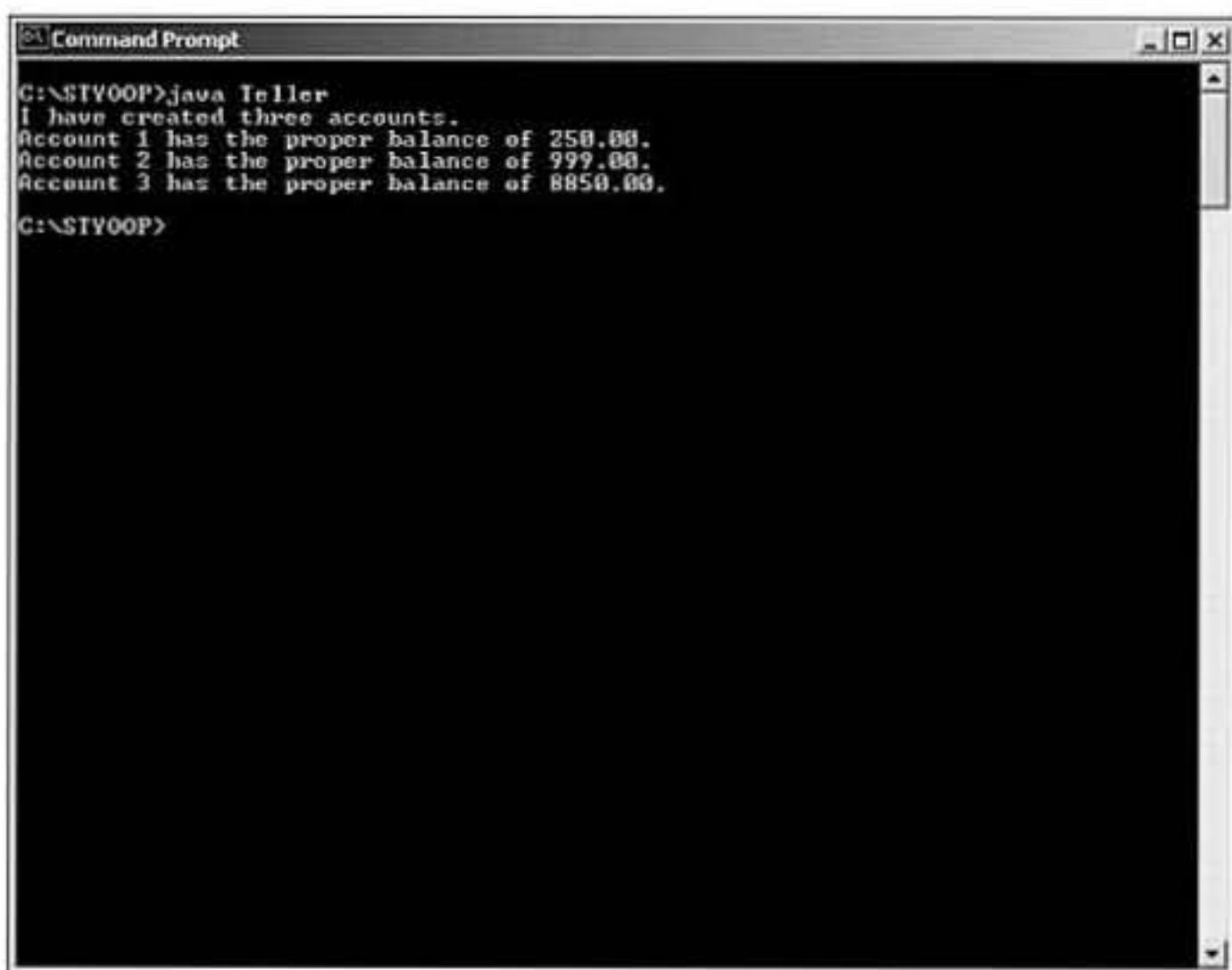
Além dessas regras, você pode adicionar quaisquer outros métodos que possa considerar úteis. Entretanto, certifique-se de implementar cada um dos métodos exatamente como listado anteriormente. Caso contrário, o caixa não poderá fazer seu trabalho!

Uma vez que você tenha terminado de escrever a classe `Account`, certifique-se de compilar as classes `Account` e `Teller`. Uma vez feito isso, execute o `main` de `Teller`, digitando `java Teller`.

Se você fez seu trabalho corretamente, deverá ver a saída ilustrada na Figura 3.1.

**FIGURA 3.1**

*A saída correta  
de Teller.*



```
C:\STVOOP>java Teller
I have created three accounts.
Account 1 has the proper balance of 250.00.
Account 2 has the proper balance of 999.00.
Account 3 has the proper balance of 8850.00.
C:\STVOOP>
```

**ALERTA**

A próxima seção discutirá as soluções do Laboratório 2. Não prossiga até concluir o Laboratório 2!

## Soluções e discussão

A Listagem 3.2 apresenta uma possível implementação de `Account`.

**LISTAGEM 3.2** Account.java

```
public class Account {  
  
    // dados privados  
    private double balance;  
  
    // construtor  
    public Account( double init_deposit ) {  
        balance = init_deposit;  
    }  
  
    public Account() {  
        // não precisa fazer nada, _balance terá 0 como padrão  
    }  
  
    // deposita dinheiro na conta  
    public void depositFunds( double amount ) {  
        balance = balance + amount;  
    }  
  
    // consulta o saldo  
    public double getBalance() {  
        return balance;  
    }  
  
    // saca fundos da conta  
    public double withdrawFunds( double amount ) {  
  
        if(amount > balance){ // ajusta o valor  
            amount = balance;  
        }  
  
        balance = balance - amount;  
        return amount;  
    }  
}
```

3

A classe Account ilustra os conceitos importantes existentes por trás do encapsulamento. Account é bastante abstrata. Ela funcionará como a base para muitos tipos diferentes de contas. A classe Account oculta sua implementação atrás de uma interface bem definida. Finalmente, a classe Account mostra uma divisão correta de responsabilidades, pois contém todo o conhecimento de como debitar e creditar no saldo da conta. O conhecimento de como executar essas tarefas não ‘vaza’ para fora do objeto.

Entretanto, Account não é perfeita; ainda há espaço para melhoria. Por questão de brevidade, essa solução de classe Account pula a validação do argumento, além da verificação de saque a descoberto simples. Para usar no mundo real, você precisaria incluir código para validar todos os parâmetros dos métodos.

## Laboratório 3: o encapsulamento

O Dia 2 abordou três características do encapsulamento eficaz:

- Abstração
- Ocultação da implementação
- Divisão da responsabilidade

Cada característica é uma habilidade importante a ser dominada, enquanto você projeta e escreve suas classes. Você precisa aplicar todas as três características para ter objetos bem encapsulados.

Vamos aplicar essas três características em um jogo de cartas.

Primeiro, vamos aplicar a abstração. Lembre-se de não abusar da abstração. Você ainda tem um problema para resolver e não pode resolver todos os problemas. Assim, você deve primeiro tentar resolver os problemas que conhece!

O quê você pode dizer genericamente a respeito de jogos de carta jogados com um baralho de pôquer padrão?

Um bom lugar para começar é no próprio maço de cartas. Um baralho padrão contém 52 cartas. Você pode embaralhar um maço, assim como escolher uma carta do baralho, em qualquer posição. Do mesmo modo, você pode retornar uma carta para qualquer posição no maço. Qualquer outra extração é apenas uma especialização da escolha de uma carta a partir de qualquer parte do maço.

O que você pode dizer a respeito das cartas em si?

Todas as cartas compartilham uma estrutura comum. Cada carta tem um naipe: ouros, copas, espadas ou paus. Cada carta também tem um valor: 2 a 10, valete, dama, rei ou ás. A única diferença de uma carta para outra é o valor desses dois atributos.

Levado a um extremo, você poderia tentar descrever cada tipo de maço de cartas, sejam elas cartas de beisebol ou de tarô. Novamente, quando você começa a abstrair, precisa certificar-se de não abstrair em demasia.

E quanto a ocultação da implementação?

A não ser que você roube enquanto joga baralho, nunca verá o que está no maço, até receber uma carta. Você também não insere cartas que não fazem parte do baralho.

Finalmente, e quanto à responsabilidade?

No mundo real, as cartas em si não fazem muito. Uma carta simplesmente exibe seu naipe e seu valor. Uma carta tem um estado: face para cima ou face para baixo. Do mesmo modo, um baralho não faz muito no mundo real. Em vez disso, o carteador é aquele que embaralha e distribui as cartas. O baralho simplesmente contém as cartas de jogo.

No mundo dos computadores, uma carta conterá seu naipe, valor e estado. Em um programa simples, uma carta também saberá como ser apresentada. Um baralho criará e conterá as cartas. Finalmente, o carteador saberá embaralhar as cartas e distribuir uma carta.

**NOTA**

Posteriormente, você aprenderá a importância de separar a exibição de seu modelo/dados. Entretanto, para nossos objetivos aqui, você pode misturar os dois.

3

## Exposição do problema

Use as classes de projeto de descrição de carta de pôquer para representar as cartas, o maço de cartas e o carteador. Então, você deve escrever um pequeno método `main()`, que instancie o carteador e seu maço de cartas, embaralhe as cartas e, em seguida, imprima o baralho.

Este laboratório deixa a você muita margem de movimento, enquanto projeta suas cartas, o baralho e o carteador. Ao pensar nas classes que você criará, certifique-se de considerar a ocultação da implementação e a divisão da responsabilidade. Coloque a responsabilidade apenas onde ela pertence e, quando você a colocar, certifique-se de que ela não ‘vaze’.

**NOTA**

Veja `java.lang.Math.random()` para a geração de números aleatórios. A função `random()` será útil para embaralhar o maço. Você pode obter a documentação completa das APIs Java no endereço <http://www.javasoft.com/>.

Por exemplo, `(int) (Math.random() * 52)` fornecerá um número entre 0 e 51.

**ALERTA**

A próxima seção discutirá as soluções do Laboratório 3. Não prossiga até completar o Laboratório 3!

## Soluções e discussão

A Listagem 3.3 apresenta uma possível classe `Card`.

**LISTAGEM 3.3** Card.java

```
public class Card {  
  
    private int rank;
```

**LISTAGEM 3.3 Card.java (continuação)**

```
private int suit;
private boolean face_up;

// constantes usadas para instanciar
// naipes
public static final int DIAMONDS = 4;
public static final int HEARTS   = 3;
public static final int SPADES   = 6;
public static final int CLUBS    = 5;
// valores
public static final int TWO     = 2;
public static final int THREE    = 3;
public static final int FOUR    = 4;
public static final int FIVE    = 5;
public static final int SIX     = 6;
public static final int SEVEN   = 7;
public static final int EIGHT   = 8;
public static final int NINE    = 9;
public static final int TEN     = 10;
public static final int JACK    = 74;
public static final int QUEEN   = 81;
public static final int KING    = 75;
public static final int ACE     = 65;

// cria uma nova carta - usa apenas as constantes para inicializar
public Card( int suit, int rank ) {
    // Em um programa real, você precisaria fazer a validação dos argumentos.

    this.suit = suit;
    this.rank  = rank;
}

public int getSuit() {
    return suit;
}

public int getRank() {
    return rank;
}

public void faceUp() {
    face_up = true;
}

public void faceDown() {
```

**LISTAGEM 3.3** Card.java (*continuação*)

```
    face_up = false;
}

public boolean isFaceUp() {
    return face_up;
}

public String display() {
    String display;

    if( rank > 10 ) {
        display = String.valueOf( (char) rank );
    } else {
        display = String.valueOf( rank );
    }

    switch ( suit ) {
        case DIAMONDS:
            return display + String.valueOf( (char) DIAMONDS );
        case HEARTS:
            return display + String.valueOf( (char) HEARTS );
        case SPADES:
            return display + String.valueOf( (char) SPADES );
        default:
            return display + String.valueOf( (char) CLUBS );
    }
}
}
```

3

A definição da classe Card começa definindo várias constantes. Essas constantes enumeram os valores e naipes de carta válidos.

Você notará que, uma vez instanciado, não é possível mudar o valor da carta. As instâncias de Card são imutáveis. Tornando a carta imutável, ninguém poderá alterar erroneamente o valor de uma carta.

**Novo Termo** Um objeto *imutável* é aquele cujo estado não muda, uma vez construído.

A classe Card é responsável por conter seu naipe, assim como o valor. A carta também sabe como retornar uma representação de String de si mesma.

A Listagem 3.4 apresenta uma possível implementação de Deck.

**LISTAGEM 3.4** Deck.java

```
public class Deck {  
  
    private java.util.LinkedList deck;  
  
    public Deck() {  
        buildCards();  
    }  
  
    public Card get( int index ) {  
        if( index < deck.size() ) {  
            return (Card) deck.get( index );  
        }  
        return null;  
    }  
  
    public void replace( int index, Card card ) {  
        deck.set( index, card );  
    }  
  
    public int size() {  
        return deck.size();  
    }  
  
    public Card removeFromFront() {  
        if( deck.size() > 0 ){  
            Card card = (Card) deck.removeFirst();  
            return card;  
        }  
        return null;  
    }  
  
    public void returnToBack( Card card ) {  
        deck.add( card );  
    }  
  
    private void buildCards() {  
  
        deck = new java.util.LinkedList();  
  
        deck.add( new Card( Card.CLUBS, Card.TWO ) );  
        deck.add( new Card( Card.CLUBS, Card.THREE ) );  
        deck.add( new Card( Card.CLUBS, Card.FOUR ) );  
        deck.add( new Card( Card.CLUBS, Card.FIVE ) );  
        // a definição completa foi cortada por brevidade  
        // veja a listagem completa no código-fonte  
    }  
}
```

---

A classe Deck é responsável por instanciar as cartas e, em seguida, fornecer acesso a elas. A classe Deck fornece métodos para recuperar e retornar as cartas.

A Listagem 3.5 apresenta a implementação de Dealer.

**LISTAGEM 3.5 Dealer.java**

```
public class Dealer {  
  
    private Deck deck;  
  
    public Dealer( Deck d ){  
        deck =d;  
    }  
  
    public void shuffle() {  
        //torna o array de cartas aleatório  
        int num_cards = deck.size();  
        for( int i = 0; i < num_cards; i ++ ) {  
            int index = (int)( Math.random() * num_cards );  
            Card card_i = ( Card ) deck.get( i );  
            Card card_index = ( Card ) deck.get( index );  
            deck.replace( I, card_index );  
            deck.replace( index, card_i );  
        }  
    }  
  
    public Card dealCard() {  
        if( deck.size() > 0 ) {  
            return deck.removeFromFront();  
        }  
        return null;  
    }  
}
```

3

A classe Dealer é responsável por embaralhar o maço e distribuir as cartas. Essa implementação de Dealer é honesta. Outra implementação de Dealer poderia dar as cartas a partir do final do baralho!

Todas as três classes têm uma divisão da responsabilidade clara. A classe Card representa cartas de pôquer, a classe Deck contém as cartas e a classe Dealer distribui as cartas. Todas as três classes também ocultam sua implementação. Nada sugere que a classe Deck tenha realmente uma `LinkedList` de cartas.

Embora Card possa definir várias constantes, isso não compromete a integridade de sua implementação, pois a classe Card está livre para usar as constantes como quiser. Ela também está livre para mudar os valores das constantes a qualquer momento.

O método `buildCards()` de Deck destaca uma deficiência da ocultação da implementação. Você poderia instanciar cartas com números de 2 a 10 em um laço `for`. Se você examinar as constantes, verá que TWO a TEN contam de 2 a 10, seqüencialmente. Tal laço é muito mais simples do que instanciar cada carta individualmente.

Entretanto, tal suposição o vincula aos valores correntes das constantes. Você não deve permitir que seu programa se torne dependente de determinado valor, escondido na constante. Em vez disso, você deve usar a constante cegamente, chamando `Card.TWO`, `Card.THREE`, etc. Você não deve fazer quaisquer tipos de suposições sobre o valor. Card poderia redefinir os valores das constantes a qualquer momento. No caso de `buildCards()`, é fácil cair na tentação de usar os valores das constantes diretamente.

Aqui, o contrato entre Card e o usuário das constantes de Card são os nomes das constantes e não seu valor subjacente. O Dia 12, “Padrões avançados de projeto”, apresentará uma solução um pouco mais elegante do que esse uso de constantes.

## Laboratório 4: estudo de caso — os pacotes de primitivas Java (opcional)



O Laboratório 4 é um laboratório opcional. Embora a conclusão do laboratório dê a você mais idéias sobre a programação orientada a objetos, sua conclusão não é necessária para ter êxito nos próximos dias.

Cada linguagem orientada a objetos tem suas próprias regras para determinar o que é e o que não é um objeto. Algumas linguagens orientadas a objetos são mais ‘puras’ que outras. Uma linguagem puramente orientada a objetos, como a Smalltalk, considera tudo um objeto, até mesmo operadores e primitivas.

**Novo Termo** Uma linguagem orientada a objetos *pura* suporta a noção de que tudo é um objeto.

Em uma linguagem puramente orientada a objetos, tudo — classes, primitivas, operadores e até blocos de código — é considerado um objeto.

A linguagem Java tem suas próprias regras para determinar o que é e o que não é um objeto. Na linguagem Java nem tudo é um objeto. Por exemplo, a linguagem Java declara diversos valores de primitivas. As primitivas não são consideradas objetos na linguagem Java. Essas primitivas compreendem `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`.

**NOVO TERMO** Uma linguagem *orientada a objetos* não considera tudo um objeto.

As primitivas oferecem algumas vantagens em relação os objetos. Para usar uma primitiva, você não precisa instanciar uma nova instância usando new. Como resultado, usar uma primitiva é muito mais eficiente do que usar um objeto, pois ela não sofre da sobrecarga associada aos objetos.

Por outro lado, às vezes você achará o uso de primitivas limitante. Você não pode tratar primitivas como objetos. Isso significa que você não pode usá-las em lugares que exigem um objeto. Considere `java.util.Vector`, da coleção de classes genéricas. Para colocar um valor no vetor, você precisa chamar o método `add()` do vetor:

```
public boolean add( Object o );
```

Para armazenar um valor no vetor, o valor deve ser um objeto. Colocado de maneira simples, se você quiser colocar uma primitiva em um vetor, estará sem sorte.

Para contornar essas falhas, a linguagem Java tem vários pacotes de primitivas, incluindo Boolean, Character, Byte, Double, Float, Integer, Long e Short. Essas classes são chamadas de pacotes, pois elas contêm, ou possuem, um valor de primitiva.

**NOVO TERMO** Um *pacote* é um objeto cujo único propósito é conter outro objeto ou primitiva. Um pacote fornecerá qualquer número de métodos para obter e manipular o valor possuído.

Vamos considerar a interface pública de Boolean, que está delineada na Listagem 3.6.

#### **LISTAGEM 3.6** `java.lang.Boolean`

```
public final class Boolean implements Serializable {
    public Boolean( boolean value );
    public Boolean( String s );

    public static final Boolean FALSE;
    public static final Boolean TRUE;
    public static final CLASS TYPE;

    public static boolean getBoolean( String name );
    public static Boolean valueOf( String s );

    public boolean booleanValue();
    public boolean equals( Object obj );
    public int hashCode();
    public String toString();
}
```

**NOTA**

*Final* se refere ao conceito de impedir que classes descendentes alterem esse elemento, quando herdadas. A herança será discutida no Dia 4, "Herança: obtendo algo para nada".

*Implements* está relacionado à construção de 'interface' especial da linguagem Java, discutida no Apêndice B, 'Resumo do Java'.

Internamente, o pacote Boolean conterá uma primitiva boolean. Assim, para passar um valor boolean para um vetor, você precisaria primeiro instanciar um pacote Boolean, possuir a primitiva boolean e passar esse pacote para o vetor.

A interface Boolean introduz outros recursos das linguagens orientadas a objetos: métodos de classe e variáveis de classe.

Até agora, todos os métodos e variáveis que você viu eram métodos de instância e variáveis de instância. Isto é, cada variável e cada método está ligado a alguma instância de objeto. Para chamar o método ou acessar a variável, você deve ter uma instância do objeto.

O fato de que você precisa de uma instância é freqüentemente lógico. Considere o método `booleanValue()` de Boolean. O método `booleanValue()` é um método de instância. O valor que o método retorna dependerá do estado interno das instâncias de Boolean individuais. Uma instância pode possuir o valor true, outra pode possuir o valor false. O valor retornado dependerá do valor que a instância contém internamente.

Agora, considere o método `getBoolean()` de Boolean. Esse é um método de classe. Se você estudar a definição de `getBoolean()`, notará a palavra-chave static. Na linguagem Java, a palavra-chave static declara que o método ou variável é um método ou variável de classe.

Ao contrário das variáveis e métodos de instância, os métodos e variáveis de classe não estão vinculados a nenhuma instância. Em vez disso, você acessa métodos de classe através da própria classe. Assim, para chamar `getBoolean()`, você não precisa de uma instância (contudo, você ainda poderia chamar o método como se ele fosse um método de instância). Em vez disso, você pode simplesmente chamar `Boolean.getBoolean()`. A resposta de `getBoolean()` não depende do estado de nenhuma instância. Por isso, ele pode escapar impunemente, sendo declarado como um método de classe.

**Novo TERMO** *Variáveis de classe* são variáveis que pertencem à classe e não a uma instância específica. As variáveis de classe são compartilhadas entre todas as instâncias da classe.

**Novo TERMO** *Métodos de classe* são métodos que pertencem à classe e não a uma instância específica. A operação executada pelo método não é dependente do estado de qualquer instância.

As variáveis de classe funcionam da mesma maneira. Você não precisa de uma instância para acessá-las. Entretanto, elas também têm outro uso. Como a variável é mantida no nível da classe, todas as instâncias compartilham a mesma variável (e, se for pública, todos os objetos poderão compartilhá-la). As variáveis de classe diminuem os requisitos de memória. Considere `public static final Boolean FALSE`. Essa é uma constante que possui o valor false. Como ela é estática, todas as instâncias compartilham essa mesma constante. Cada instância não precisa de sua própria cópia. Considere a classe, `CountedObject`, a seguir:

```
public class CountedObject {  
  
    private static int instances;  
  
    /** Cria novo CountedObject */  
    public CountedObject() {  
        instances++;  
    }  
  
    public static int getNumberInstances() {  
        return instances;  
    }  
  
    public static void main( String[] args ) {  
        CountedObject obj = null;  
        for( int i = 0; i < 10; i++ ) {  
            obj = new CountedObject();  
        }  
        System.out.println( "Instances created: " +  
                           obj.getNumberInstances() );  
        // note que isso também funcionará  
        System.out.println( "Instances created: " +  
                           CountedObject.getNumberInstances() );  
    }  
}
```

3

CountedObject declara uma variável de classe chamada instances. Ela também declara um método de classe para recuperar o valor, getNumberInstances(). Dentro do construtor, o valor é incrementado sempre que uma instância é criada. Como todas as instâncias compartilham a variável, a variável instances atua como um contador. À medida que cada objeto é criado, ela incrementa o contador.

O método main() cria 10 instâncias. Você notará que pode usar uma instância para fazer a chamada de getNumberInstances() ou a própria classe.

Se você vai ou não declarar um método ou variável estática é uma decisão de projeto. Se o método ou variável for independente do estado de qualquer instância, provavelmente é uma boa idéia torná-lo um método ou variável de classe. Entretanto, você não pode declarar variáveis e métodos que são dependentes da instância, como estáticos, como o método booleanValue() de Boolean.

Você pode ter feito algumas observações a respeito de Boolean. Se você estudar a interface, notará que não há meios de mudar o valor boolean possuído, uma vez que tenha instanciado a instância de Boolean! Como você não pode mudar seu valor, diz-se que as instâncias de Boolean são imutáveis.

Existem ocasiões em que usar um objeto imutável é fundamental. Se você estiver familiarizado com linhas de execução, um objeto imutável é inherentemente seguro quanto a linha de execução, pois seu estado nunca pode mudar.

Entretanto, existem ocasiões em que os objetos imutáveis causam mais danos do que trazem vantagens. No caso dos pacotes de primitivas, a sobrecarga da instanciação de um pacote para cada primitiva pode se tornar dispendiosa.

## Exposição do problema

Para o Laboratório 4, você precisa criar um pacote de primitiva Boolean mutável. No mínimo, esse pacote deve permitir que você obtenha e configure o valor possuído. O pacote também deve fornecer dois construtores: um construtor noargs e um construtor que recebe o valor inicial do pacote.

Sinta-se livre para adicionar quaisquer outros métodos que considere conveniente. Entretanto, não se esqueça de seguir as regras do encapsulamento eficaz.

Talvez você ache interessante examinar a discussão do Apêndice B sobre a palavra-chave `static`, se decidir fornecer todos os métodos oferecidos pelo pacote de primitivas Boolean.



A próxima seção discutirá as soluções do Laboratório 4. Não prossiga até completar o Laboratório 4!

## Soluções e discussão

A Listagem 3.7 apresenta uma possível solução para o Laboratório 4.

---

### LISTAGEM 3.7 MyBoolean.java

---

```
public class MyBoolean {  
  
    // algumas constantes, por conveniência  
    public static final Class TYPE = Boolean.TYPE;  
  
    private boolean value;  
  
    // construtor sem argumento - tem false como padrão  
    public MyBoolean() {  
        value = false;  
    }  
  
    // configura o valor inicial como value  
    public MyBoolean( boolean value ) {  
        this.value = value;  
    }  
  
    public boolean booleanValue() {
```

**LISTAGEM 3.7 MyBoolean.java (continuação)**

```
    return value;
}

public void setBooleanValue( boolean value ) {
    this.value = value;
}

// para getBoolean e valueOf, podemos simplesmente delegar para Boolean
// você vai aprender mais sobre delegação no Capítulo 4
public static boolean getBoolean( String name ) {
    return Boolean.getBoolean( name );
}

public static MyBoolean valueOf( String s ) {
    return new MyBoolean( Boolean.getBoolean( s ) );
}

// definições de hashCode, equals e toString omitidas por brevidade
}
```

3

MyBoolean mantém a interface pública encontrada em Boolean, com três exceções:

- MyBoolean adiciona um mutante: `public void setBooleanValue( boolean value )`. Esse mutante permite que você mude o valor dos pacotes.
- MyBoolean redefine `valueOf()` de modo que retorne uma instância de MyBoolean, em vez de Boolean.
- MyBoolean remove as constantes TRUE e FALSE. Agora que MyBoolean é mutante, esses valores não se tornam constantes adequadas, pois seus valores podem ser alterados por qualquer um, a qualquer momento.

A solução do Laboratório 4 também fornece um vislumbre do Dia 4, “Herança: obtendo algo para nada”. Métodos como `valueOf()` demonstram a delegação. A solução de código-fonte completa do Laboratório 4 também proporciona uma visão da herança e da sobrecarga, através dos métodos `toString()`, `hashCode()` e `equals()`.

## Perguntas e respostas

**P** No Laboratório 3, você escreveu, “Essa implementação de Dealer é honesta. Outra implementação de Dealer poderia distribuir as cartas a partir do final do baralho!” O que você quer dizer com outra implementação?

**R** Você pode dizer que os métodos `shuffle()` e `dealCard()` constituem a interface pública de Dealer. A classe Dealer apresentada é honesta. Ela distribui as cartas a partir do início

do baralho. Você poderia escrever outro carteador, chamado `DishonestDealer`, que tivesse a mesma interface pública. Entretanto, esse carteador poderia distribuir as cartas a partir do final do baralho.

Você chama esse carteador de outra implementação, porque ele reimplementa uma interface igual àquela encontrada em `Dealer`. Entretanto, essa classe implementa a funcionalidade oculta no método de forma ligeiramente diferente.

### P O encapsulamento pode ser prejudicial?

R Na verdade, o encapsulamento pode ser prejudicial. Imagine que você tenha um componente que efetue cálculos matemáticos. Suponha que você precise manter determinada precisão, quando concluir seu cálculo. Infelizmente, o componente pode encapsular completamente a quantidade de precisão que mantém. Você poderia acabar com um valor incorreto, se a implementação usasse uma precisão diferente daquela que precisa. Você pode acabar com erros estranhos, se alguém alterar o componente.

Assim, o encapsulamento pode ser prejudicial, se você precisar de um controle preciso sobre as maneiras pelas quais um objeto manipula seus pedidos.

A única defesa é a boa documentação. Você deve documentar todos os detalhes e suposições importantes da implementação. Uma vez documentados, você não poderá fazer alterações facilmente em quaisquer detalhes ou suposições documentados. Assim como no componente matemático, se você fizer uma alteração, correrá o risco de prejudicar todos os usuários desse objeto.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Examine a classe `Account` do Laboratório 2. Qual(is) método(s) é(são) mutante(s)? Qual(is) método(s) é(são) acessor(es)?
2. Quais são os dois tipos de construtores? A partir das soluções de laboratório, encontre um exemplo de cada tipo de construtor.
3. (Opcional) Boolean, conforme discutido no Laboratório 4, declara três variáveis públicas. Neste caso, o uso de variáveis públicas é aceitável. Você pode explicar por que está certo usar variáveis públicas nesse caso?
4. (Opcional) Como você pode tornar a solução do Laboratório 3 mais eficiente?

5. Por que você acha que a solução do Laboratório 3 não criou uma classe Card separada para cada naipe?
6. No Laboratório 3, você explorou a divisão da responsabilidade. Quais vantagens a divisão da responsabilidade proporciona às classes Card, Deck e Dealer?

## Exercícios

1. (Opcional) Pegue o Laboratório 2 e abstraia DoubleKey ainda mais. Refaça o projeto de DoubleKey de modo que ela possa aceitar qualquer tipo de objeto como chave — não apenas uma String.

Para que sua nova classe DoubleKey funcione, você precisará alterar a definição dos métodos `equals()` e `hashCode()`. Esses métodos foram omitidos por brevidade nas soluções impressas. Entretanto, os métodos estão disponíveis no código-fonte completo das soluções.

2. (Opcional) No Laboratório 3, as instâncias de Card sabem como se apresentar. Entretanto, a classe Deck não sabe como se apresentar sozinha. Refaça o projeto de Deck de modo que as instâncias de Deck saibam como se apresentar.

PÁGINA EM BRANCO

# SEMANA 1

## DIA 4

### Herança: obtendo algo para nada

Nos três últimos dias, você se concentrou em aprender sobre o primeiro pilar da programação orientada a objetos: encapsulamento. Embora o encapsulamento seja um conceito fundamental na POO, há mais na história do que apenas suportar TADs e módulos simples. Na verdade, a POO ofereceria muito pouco em relação ao estilo de programação antigo, se tudo que ela fizesse fosse oferecer encapsulamento simples. É claro que a POO oferece muito mais.

A POO vai além, adicionando dois outros recursos: *herança* e *polimorfismo*. Você vai passar os próximos dois dias considerando a herança, o segundo pilar da POO.

Hoje você vai aprender:

- O que é herança
- Os diferentes tipos de herança
- Algumas das armadilhas da herança
- Dicas para a herança eficaz
- Como a herança atende aos objetivos da OO

### O que é herança?

Ontem, você viu como o encapsulamento permite escrever objetos bem definidos e independentes. O encapsulamento permite que um objeto *use* outro objeto, através de mensagens. O *uso* é apenas uma das maneiras pelas quais os objetos podem se relacionar na POO. A POO também fornece uma segunda maneira de relacionamento entre os objetos: herança.

A herança permite que você baseie a definição de uma nova classe em uma classe previamente existente. Quando você baseia uma classe em outra, a definição da nova classe herda automaticamente todos os atributos, comportamentos e implementações presentes na classe previamente existente.

**Novo Termo** *Herança* é um mecanismo que permite a você basear uma nova classe na definição de uma classe previamente existente. Usando herança, sua nova classe herda todos os atributos e comportamentos presentes na classe previamente existente. Quando uma classe herda de outra, todos os métodos e atributos que aparecem na interface da classe previamente existente aparecerão automaticamente na interface da nova classe.

Considere a classe a seguir:

```
public class Employee {  
  
    private String first_name;  
    private String last_name;  
    private double wage;  
  
    public Employee( String first_name, String last_name, double wage ) {  
        this.first_name = first_name;  
        this.last_name = last_name;  
        this.wage = wage;  
    }  
  
    public double getWage() {  
        return wage;  
    }  
  
    public String getFirstName() {  
        return first_name;  
    }  
  
    public String getLastname() {  
        return last_name;  
    }  
}
```

Instâncias de uma classe como `Employee` podem aparecer em um aplicativo de banco de dados de folha de pagamento. Agora, suponha que você precisasse modelar um funcionário comissionado. Um funcionário comissionado tem um salário-base, mais uma pequena comissão por venda. Além desse requisito simples, a classe `CommissionedEmployee` é exatamente igual à classe `Employee`. Afinal, um objeto `CommissionedEmployee` é um objeto `Employee`.

Usando-se o encapsulamento direto, existem duas maneiras de escrever a nova classe `CommissionedEmployee`. Você poderia simplesmente repetir o código encontrado em `Employee` e adicionar

o código necessário para controlar comissões e calcular o pagamento. Entretanto, se você fizer isso, terá de manter duas bases de código separadas, mas semelhantes. Se você precisar corrigir um erro, terá de fazê-lo em cada lugar.

Assim, simplesmente copiar e colar o código não é boa uma opção. Você precisará tentar outra coisa. Você poderia ter uma variável `employee` dentro da classe `CommissionedEmployee` e delegar todas as mensagens, como `getWage()` e `getFirstName()`, à instância de `Employee`.

**Novo Termo** *Delegação* é o processo de um objeto passar uma mensagem para outro objeto, para atender algum pedido.

Entretanto, a delegação ainda o obriga a redefinir todos os métodos encontrados na interface de `Employee` para passar todas as mensagens. Assim, nenhuma dessas duas opções parece satisfatória.

Vamos ver como a herança pode corrigir esse problema:

```
public class CommissionedEmployee extends Employee {  
  
    private double commission; // o custo por unidade  
    private int units; // controle do número de unidades vendidas  
  
    public CommissionedEmployee( String first_name, String last_name,  
                                double wage, double commission ) {  
        super( first_name, last_name, wage ); // chama o construtor original  
                                            // para inicializar corretamente  
                                            // o valor da comissão  
        this.commission = commission;  
    }  
  
    public double calculatePay() {  
        return getWage() + ( commission * units );  
    }  
  
    public void addSales( int units ) {  
        this.units = this.units + units;  
    }  
  
    public void resetSales() {  
        units = 0;  
    }  
}
```

4

Aqui, `CommissionedEmployee` baseia sua definição na classe `Employee` já existente. Como `CommissionedEmployee` herda de `Employee`, `getFirstName()`, `getLastName()`, `getWage()`, `first_name`, `last_name` e `wage` se tornarão todos parte de sua definição.

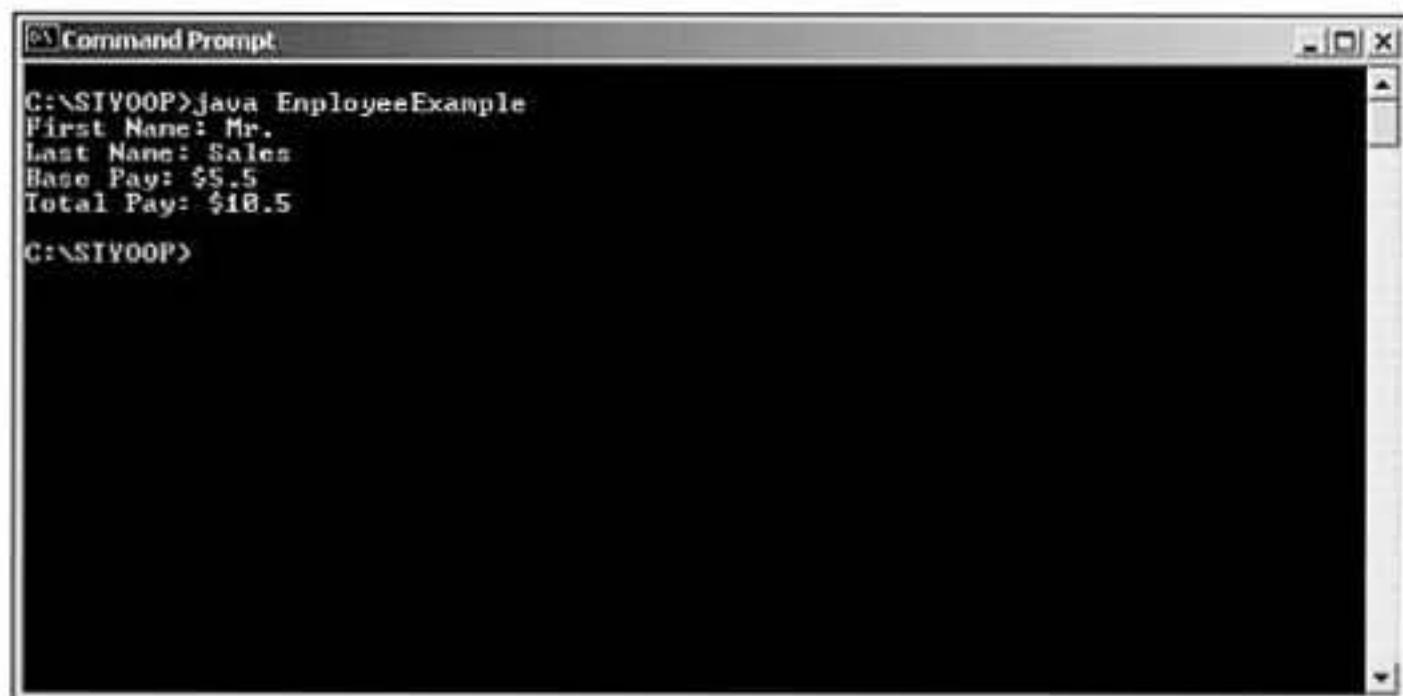
Como a interface pública de Employee se torna parte da interface de CommissionedEmployee, você pode enviar para CommissionedEmployee qualquer mensagem que poderia enviar para Employee. Considere o método main() a seguir, que faz exatamente isso:

```
public static void main(String [] args ) {  
    CommissionedEmployee c =  
        new CommissionedEmployee("Mr.", "Sales", 5.50, 1.00);  
    c.addSales(5);  
    System.out.println( "First Name: " + c.getFirstName() );  
    System.out.println( "Last Name: " + c.getLastName() );  
    System.out.println( "Base Pay:$ " + c.getWage() );  
    System.out.println( "Total Pay:$ " + c.calculatePay() );  
}
```

A Figura 4.1 ilustra o que você verá após a execução desse código.

**FIGURA 4.1**

Saída gerada a partir de  
*CommissionedEmployee*.



## Por que herança?

Conforme você viu no último exemplo, às vezes o relacionamento de *uso* do encapsulamento simples não é suficiente. Entretanto, há mais na herança do que simplesmente herdar uma interface pública e implementação.

Conforme você verá posteriormente no dia de hoje, a herança permite à classe que está herdando redefinir qualquer comportamento de que não goste. Tal recurso permite que você adapte seu software, quando seus requisitos mudarem. Se você precisar fazer uma alteração, bastará escrever uma nova classe, que herde a antiga funcionalidade. Então, sobreponha a funcionalidade que precisa mudar ou adicione a funcionalidade que está faltando e pronto. A sobreposição é interessante, pois permite mudar a maneira como um objeto funciona sem tocar na definição original da classe! Você pode deixar seu código bem testado e validado intacto. A sobreposição funciona mesmo que você não tenha o código-fonte original de uma classe.

A herança tem outro uso muito importante. No Dia 1, “Introdução à programação orientada a objetos”, você viu como uma classe agrupa objetos relacionados. A herança permite que você agrupe classes relacionadas. A POO sempre se esforça por produzir software natural. Assim como no mundo real, a POO permite que você agrupe e classifique suas classes.

## “É um” versus “tem um”: aprendendo quando usar herança

Para apresentar os mecanismos de herança, a primeira seção abordou o que é conhecido como *herança de implementação*. Conforme você viu, a herança de implementação permite que suas classes herdem a implementação de outras classes. Entretanto, somente porque uma classe pode herdar de outra não significa que isso deve ser feito!

Então, como você sabe quando deve usar herança? Felizmente, existe uma regra geral a ser seguida, para evitar uma herança incorreta.

Quando você está considerando a herança para reutilização ou por qualquer outro motivo, precisa primeiro perguntar-se se a classe que está herdando é do mesmo tipo que a classe que está sendo herdada. O fato de pensar em termos de tipo enquanto se herda é freqüentemente referido como teste ‘é um’.

**Novo Termo** *É um* descreve o relacionamento em que uma classe é considerada do mesmo tipo de outra.

Para usar ‘é um’, você diz a si mesmo, “um objeto CommissionedEmployee ‘é um’ Employee”. Essa declaração é verdadeira e você saberia imediatamente que a herança é válida nessa situação. Agora, pare e considere a interface Iterator Java:

```
public interface Iterator {  
    public boolean hasNext();  
    public Object next();  
    public void remove();  
}
```

4

Digamos que você quisesse escrever uma classe que implementasse essa interface. Se você lembrar do Dia 2, poderá perceber que uma implementação de Queue poderia ser útil na construção de sua interface Iterator. Você poderia usar toda a implementação de Queue previamente existente, para conter os elementos da interface Iterator. Quando você precisa verificar `hasNext()` ou `remove()`, pode simplesmente chamar o método Queue correto e retornar o resultado.

Nesse caso, a herança fornecerá um modo rápido de implementar uma interface Iterator. Entretanto, antes de começar a codificar, não se esqueça do teste ‘é um’.

“Uma interface Iterator ‘é uma’ Queue”. Claramente essa declaração é falsa. Esqueça-se de herdar de Queue!



Uma Queue pode ‘ter uma’ interface Iterator que saiba como percorrer os elementos.

Existirão muitas situações onde o teste ‘é um’ falhará, quando você quiser reutilizar alguma implementação. Felizmente, existem outras maneiras de reutilizar implementação. Você sempre pode usar composição e delegação (veja o quadro a seguir). O teste ‘tem um’ salva o dia.

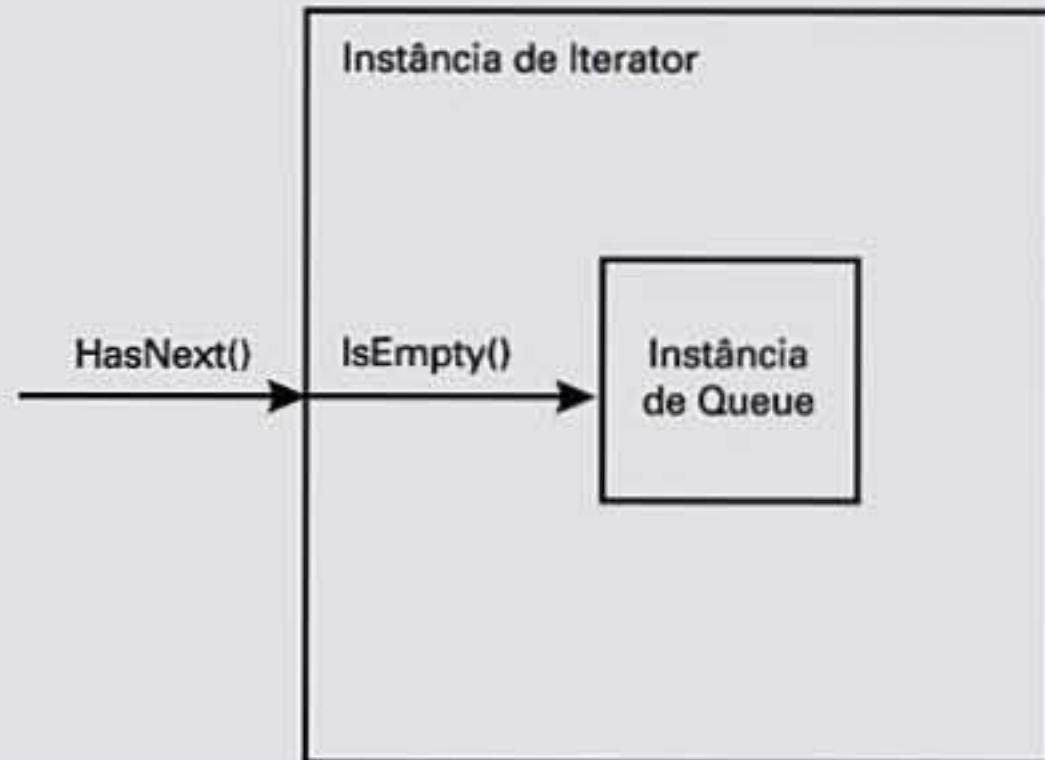
**Novo TERMO** *Tem um* descreve o relacionamento em que uma classe contém uma instância de outra classe.

**Novo TERMO** *Composição* significa que uma classe é implementada usando-se variáveis internas (chamadas de variáveis membro), que contêm instâncias de outras classes.

Composição é uma forma de reutilização que você já viu. Se você não puder herdar, nada o impede de usar instâncias da outra classe dentro da nova classe. Quando você quiser usar os recursos de outra classe, use simplesmente uma instância dessa classe como uma de suas partes constituintes. É claro que você sofre as limitações apresentadas anteriormente.

Considere novamente o exemplo Queue/Iterator. Em vez de herdar de Queue, a interface Iterator pode simplesmente criar uma instância de Queue e armazená-la em uma variável de instância. Quando a interface Iterator precisa recuperar um elemento ou verificar se está vazia, ela pode simplesmente delegar o trabalho para a instância de Queue, como demonstrado na Figura 4.2.

**FIGURA 4.2**  
*Uma interface Iterator delegando chamadas de método para Queue.*



Quando usa composição, você escolhe cuidadosamente o que vai usar. Através da delegação, você pode expor alguns ou todos os recursos de seus objetos constituintes. A Figura 4.2 ilustra como a interface Iterator direciona o método `hasNext()` para o método `isEmpty()` de Queue.

É importante indicar que a delegação difere da herança de duas maneiras importantes:

1. Com a herança, você tem apenas uma instância do objeto. Existe apenas um objeto indivisível, pois o que é herdado se torna uma parte intrínseca da nova classe.
2. A delegação geralmente fornece ao usuário apenas o que está na interface pública. A herança normal dá mais acesso aos detalhes internos da classe herdada. Vamos falar a respeito de tal acesso em detalhes, no final da lição de hoje.

## Aprendendo a navegar na teia emaranhada da herança

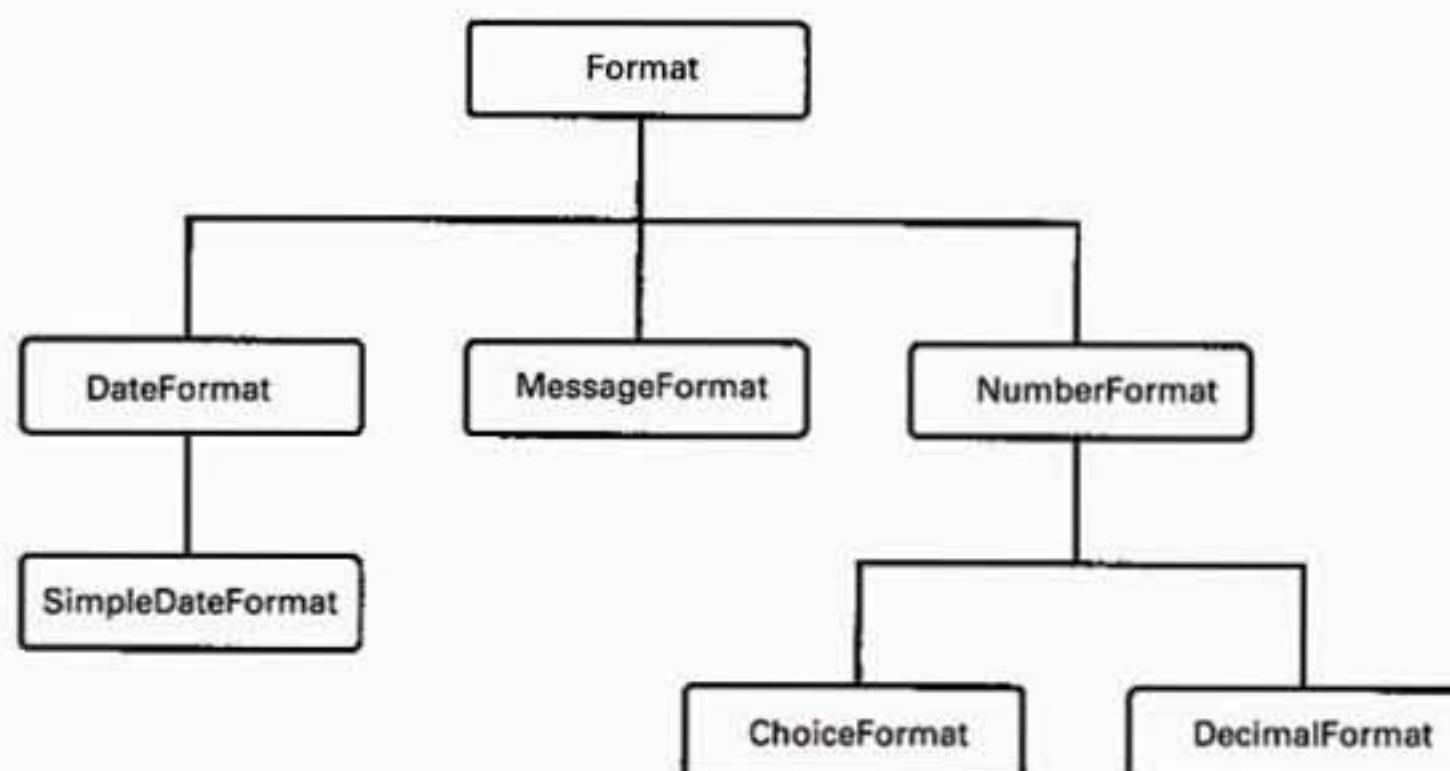
Os conceitos de ‘é um’ e composição mudam a natureza da discussão sobre herança da ambiciosa reutilização da implementação para inter-relacionamentos de classe. Uma classe que herda de outra deve se relacionar com essa classe de alguma maneira, para que os relacionamentos ou hierarquias de herança resultantes façam sentido.

**Novo TERMO**

Uma *hierarquia de herança* é um mapeamento do tipo árvore de relacionamentos que se formam entre classes como resultado da herança. A Figura 4.3 ilustra uma hierarquia real extraída da linguagem Java.

**FIGURA 4.3**

Um exemplo de hierarquia de `java.text`.



4

A herança define a nova classe, a *filha*, em termos de uma classe antiga, a *progenitora ou mãe*. Esse relacionamento filha-progenitora ou filha-mãe é o relacionamento de herança mais simples. Na verdade, todas as hierarquias de herança começam com uma progenitora e uma filha.

**Novo TERMO**

A *classe filha* é a classe que está herdando; também conhecida como subclasse.

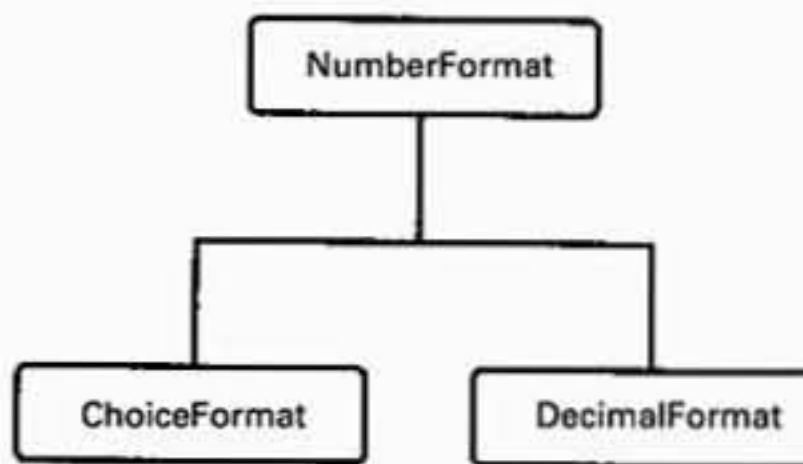
**Novo TERMO**

A *classe progenitora ou mãe* é a classe da qual a filha herda diretamente; ela também é conhecida como superclasse.

A Figura 4.4 ilustra um relacionamento progenitora/filha. `NumberFormat` é a progenitora das duas filhas `ChoiceFormat` e `DecimalFormat`.

**FIGURA 4.4**

*Uma progenitora com várias filhas.*



Agora que você já viu mais algumas definições, pode refinar a definição de herança.

**Novo TERMO**

*Herança* é um mecanismo que permite estabelecer relacionamentos ‘é um’ entre classes. Esse relacionamento também permite que uma subclasse herde os atributos e comportamentos de sua superclasse.

**NOTA**

Quando uma filha herdar de uma progenitora, a filha obterá todos os atributos e comportamentos que a progenitora possa ter herdado de outra classe.

Conforme você viu, para que a hierarquia de herança faça sentido, deve ser possível fazer na filha tudo que é possível fazer em sua progenitora. É isso que o teste ‘é um’ realmente testa. A uma filha só é permitido aumentar a funcionalidade e adicionar funcionalidades. Uma filha nunca pode remover funcionalidade.

**ALERTA**

Se você verificar que uma filha precisa remover funcionalidade, isso será uma indicação de que ela deve aparecer antes da progenitora na hierarquia de herança!

Assim como pais e filhos da vida real, as filhas e progenitoras da classe serão semelhantes entre si. Em vez de compartilhar genes, as classes compartilham informações de tipo.

**NOTA**

Como na vida real dos filhos, uma classe pode ter apenas uma progenitora física. Tudo depende de como a linguagem implementa herança.

Algumas linguagens permitem que uma classe tenha mais de uma progenitora. Isso é conhecido como *herança múltipla*.

Algumas linguagens restringem a filha a uma progenitora.

Outras linguagens, como Java, permitem apenas uma progenitora por implementação, mas fornecem um mecanismo para herdar múltiplas interfaces (mas não a implementação, apenas as assinaturas de método).

Assim como as filhas reais, as classes filhas podem adicionar novos comportamentos e atributos a si mesmas. Por exemplo, uma filha real pode aprender a tocar piano, mesmo que a mãe nunca tenha aprendido. Do mesmo modo, uma filha pode redefinir um comportamento herdado. Por exemplo, a mãe pode ter sido má aluna de matemática. A filha pode estudar mais e se tornar uma boa aluna de matemática. Quando você quer adicionar novo comportamento em uma classe, pode fazer isso adicionando um novo método na classe ou redefinindo um comportamento antigo.

## Mecânica da herança

Quando uma classe herda de outra, ela herda implementação, comportamentos e atributos. Isso significa que todos os métodos e atributos disponíveis na interface da progenitora aparecerão na interface da filha. Uma classe construída através de herança pode ter três tipos importantes de métodos e atributos:

- Sobreposto: a nova classe herda o método ou atributo da progenitora, mas fornece uma nova definição.
- Novo: a nova classe adiciona um método ou atributo completamente novo.
- Recursivo: a nova classe simplesmente herda um método ou atributo da progenitora.



A maioria das linguagens OO não permite que você sobreponha um atributo. Entretanto, o atributo sobreposto foi incluído aqui para sermos completos.

4

Primeiro, vamos considerar um exemplo. Em seguida, exploraremos cada tipo de método e atributo.

```
public class TwoDimensionalPoint {  
  
    private double x_coord;  
    private double y_coord;  
  
    public TwoDimensionalPoint( double x,double y ) {  
        setXCoordinate( x );  
        setYCoordinate( y );  
    }  
  
    public double getXCoordinate() {  
        return x_coord;  
    }  
  
    public void setXCoordinate( double x ) {  
        x_coord = x;  
    }  
}
```

```
public double getYCoordinate() {
    return y_coord;
}

public void setYCoordinate( double y ){
    y_coord = y;
}

public String toString() {
    return "I am a 2 dimensional point.\n" +
        "My x coordinate is: " + getXCoordinate() + "\n" +
        "My y coordinate is: " + getYCoordinate();
}

}

public class ThreeDimensionalPoint extends TwoDimensionalPoint {

    private double z_coord;

    public ThreeDimensionalPoint( double x, double y, double z ) {
        super( x,y ); // inicializa os atributos herdados
                      // chamando o construtor progenitor
        setZCoordinate( z );
    }

    public double getZCoordinate() {
        return z_coord;
    }

    public void setZCoordinate( double z ) {
        z_coord = z;
    }

    public String toString() {
        return "I am a 3 dimensional point.\n" +
            "My x coordinate is: " + getXCoordinate() + "\n" +
            "My y coordinate is: " + getYCoordinate() + "\n" +
            "My z coordinate is: " + getZCoordinate();
    }
}
```

Aqui, você tem duas classes ponto que representam pontos geométricos. Você poderia usar pontos em uma ferramenta de traçado de gráficos, em um modelador visual ou em um planejador de vôo. Os pontos têm muitos usos práticos.

Aqui, `TwoDimensionalPoint` contém coordenadas x e y. A classe define métodos para obter e configurar os pontos, assim como para criar uma representação de String da instância do ponto.

ThreeDimensionalPoint herda de TwoDimensionalPoint. ThreeDimensionalPoint acrescenta a coordenada z, assim como um método para recuperar o valor e para configurar o valor. A classe também fornece um método para obter uma representação de String da instância. Como ThreeDimensionalPoint herda de TwoDimensionalPoint, ela também tem os métodos contidos dentro de TwoDimensionalPoint.

Esse exemplo demonstra cada tipo de método.

## Métodos e atributos sobrepostos

A herança permite que você pegue um método ou atributo previamente existente e o redefina. A redefinição de um método permite que você mude o comportamento do objeto para esse método.

Um método ou atributo sobreposto aparecerá na progenitora e na filha. Por exemplo, ThreeDimensionalPoint redefine o método `toString()` que aparece em TwoDimensionalPoint:

```
// de TwoDimensionalPoint
public String toString() {
    return "I am a 2 dimensional point.\n" +
        "My x coordinate is: " + getXCoordinate() + "\n" +
        "My y coordinate is: " + getYCoordinate();
}
```

TwoDimensionalPoint define um método `toString()` que identifica a instância como um ponto bidimensional e imprime suas duas coordenadas .

ThreeDimensionalPoint redefine o método `toString()` para identificar a instância como um ponto tridimensional e imprime suas três coordenadas :

```
// de ThreeDimensionalPoint
public String toString() {
    return "I am a 3 dimensional point.\n" +
        "My x coordinate is: " + getXCoordinate() + "\n" +
        "My y coordinate is: " + getYCoordinate() + "\n" +
        "My z coordinate is: " + getZCoordinate();
}
```

Considere o método `main()` a seguir:

```
public static void main( String [] args ) {
    TwoDimensionalPoint two = new TwoDimensionalPoint(1,2);
    ThreeDimensionalPoint three = new ThreeDimensionalPoint(1,2,3);

    System.out.println(two.toString());
    System.out.println(three.toString());
}
```

A Figura 4.5 ilustra o que você verá após executar o método `main()`.

**FIGURA 4.5**

*Testando o método  
toString() sobreposto.*

```
C:\STYOOOP>java PointExample
I am a 2 dimensional point.
My x coordinate is: 1.0
My y coordinate is: 2.0
I am a 3 dimensional point.
My x coordinate is: 1.0
My y coordinate is: 2.0
My z coordinate is: 3.0
C:\STYOOOP>
```

Conforme você pode ver na Figura 4.5, ThreeDimensionalPoint retorna sua representação de String sobreposta.

Sobrepor um método também é conhecido como *redefinir* um método. Redefinindo um método, a filha fornece sua própria implementação personalizada do método. Essa nova implementação fornecerá um comportamento novo para o método. Aqui, ThreeDimensionalPoint redefine o comportamento do método `toString()`, para que ele seja corretamente transformado em um objeto `String`.

**Novo Termo**

*Sobrepor* é o processo de uma filha pegar um método que aparece na progenitora e reescrevê-lo para mudar o comportamento do método. A sobreposição de um método também é conhecida como *redefinição* de um método.

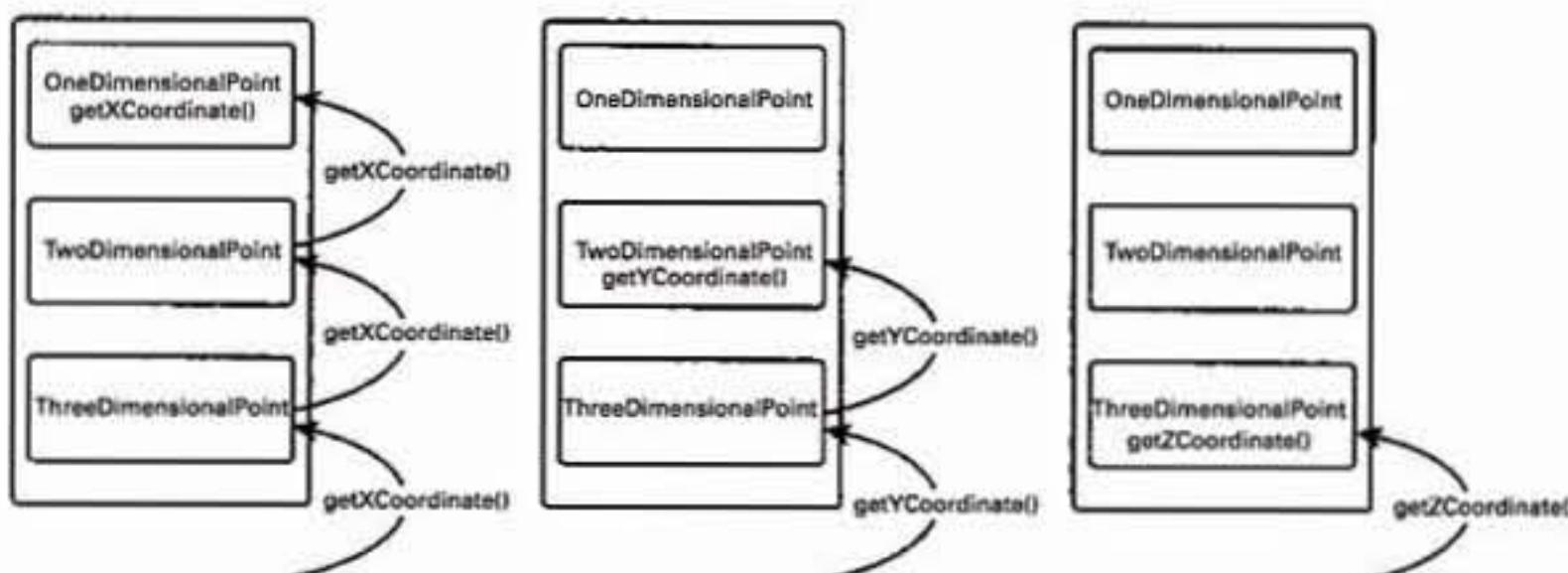
Então, como o objeto sabe qual definição deve usar?

A resposta depende do sistema OO subjacente. A maioria dos sistemas OO procurará primeiro a definição no objeto para o qual é passada a mensagem. Se uma definição não for encontrada lá, o ambiente em tempo de execução percorrerá a hierarquia, até que uma definição seja encontrada. É importante perceber que é assim que uma mensagem é manipulada e que é por isso que a sobreposição funciona. A definição da filha será a primeira a ser chamada, pois é a primeira encontrada. O mecanismo é igual para métodos e atributos recursivos, que veremos posteriormente.

A Figura 4.6 ilustra a propagação de método entre os objetos ponto para uma chamada de `getXCoordinate()`. Uma chamada de método para `getXCoordinate()` percorrerá a hierarquia até encontrar uma definição para o método.

**FIGURA 4.6**

*Propagação de  
mensagem entre  
os objetos ponto.*



Ao considerarmos a sobreposição de um método ou atributo, é importante perceber que nem todos os métodos e atributos estão disponíveis para sua filha sobrepor. A maioria das linguagens orientadas a objetos tem alguma noção de controle de acesso. As palavras-chave controle de acesso definem exatamente quem pode ver e acessar métodos e atributos. Genericamente, esses níveis de acesso caem em três categorias, conforme discutido brevemente no Dia 2:

- Privado: um nível de acesso que restringe o acesso apenas à classe.
- Protegido: um nível de acesso que restringe o acesso à classe e às filhas.
- Público: um nível de acesso que permite o acesso a todos e a qualquer um.

Os métodos e atributos protegidos são aqueles aos quais você deseja que apenas as subclasses tenham acesso. Não deixe tais métodos públicos. Apenas aqueles com amplo conhecimento da classe devem usar métodos e atributos protegidos.

Você deve tornar privados todos os atributos não-constantes e qualquer método destinado unicamente à própria classe. O nível privado impede que qualquer outro objeto chame o método, exceto quanto ao próprio objeto. Não torne protegidos os métodos privados, apenas para o caso de alguma subclasse querer acessá-los algum dia. Use o nível protegido apenas para os métodos que você *sabe* que uma subclasse deseja usar. Caso contrário, use o nível privado ou público. Tal prática rígida significará que talvez você tenha de voltar ao seu código posteriormente e mudar o nível de acesso de um método. Entretanto, isso leva a um projeto mais conciso do que um que abra tudo para uma subclasse.

4

**NOTA**

Voltar e mudar níveis de acesso pode parecer uma prática ruim. Entretanto, as hierarquias de herança nunca devem acontecer por acidente. Em vez disso, as hierarquias devem ser desenvolvidas naturalmente, enquanto você programa. Não há vergonha em refazer suas hierarquias com o passar do tempo. A POO real é um processo iterativo.

Contudo, lembre-se de que tornar tudo privado é uma regra geral. Existem casos em que esse conselho não funcionará a seu favor. Tudo depende do que você estiver programando. Por exemplo, se você vender bibliotecas de classe genéricas sem fornecer o código-fonte, provavelmente deverá ter o nível protegido como padrão, para que seus clientes possam usar herança para estender suas classes.

Na verdade, existem ocasiões em que você desejará projetar uma subclasse com a herança em mente. Em tal caso, faz sentido estabelecer um protocolo de herança. Um protocolo de herança é uma estrutura abstrata, visível apenas através dos elementos protegidos da classe. A classe progenitora chamará esses métodos e a classe filha poderá sobrepor esses métodos para aumentar o comportamento. Você vai ver um exemplo assim, amanhã.

Usando essas definições e regras, é fácil ver que os métodos e atributos protegidos e públicos são os mais importantes para a herança.

## Novos métodos e atributos

Um novo método ou atributo é um método ou atributo que aparece na filha, mas não aparece na progenitora. A filha acrescenta o novo método ou atributo em sua interface. Você viu novos métodos no exemplo `ThreeDimensionalPoint`. `ThreeDimensionalPoint` acrescenta os novos métodos `getZCoordinate()` e `setZCoordinate()`. Você pode adicionar nova funcionalidade na interface de sua filha, adicionando novos métodos e atributos.

## Métodos e atributos recursivos

Um método ou atributo recursivo é definido na progenitora ou em alguma outra ancestral, mas não na filha. Quando você acessa o método ou atributo, a mensagem é enviada para cima na hierarquia, até que uma definição do método seja encontrada. O mecanismo é igual àquele apresentado na seção sobre métodos e atributos sobrepostos.

Você viu métodos recursivos no código-fonte de `TwoDimensionalPoint` e `ThreeDimensionalPoint`. `getXCoordinate()` é um exemplo de método recursivo, pois ele é definido por `TwoDimensionalPoint` e não por `ThreeDimensionalPoint`.

Os métodos sobrepostos também podem se comportar de forma recursiva. Embora um método sobreposto apareça na filha, a maioria das linguagens orientadas a objetos fornece um mecanismo que permite a um método sobreposto chamar a versão da progenitora (ou de algum outro ancestral) do método. Essa capacidade permite que você enfatize a versão da superclasse, enquanto define novo comportamento na subclasse. Na linguagem Java, a palavra-chave `super` dá acesso à implementação de uma progenitora. Você terá a chance de usar `super` nos laboratórios do Dia 5, “Herança: hora de escrever algum código”.



### NOTA

Nem todas as linguagens fornecem a palavra-chave `super`. Para essas linguagens, você precisará tomar o cuidado de inicializar corretamente qualquer código herdado.

Não referenciar corretamente as classes herdadas pode ser uma fonte sutil de erros.

## Tipos de herança

Ao todo, existem três maneiras principais de usar herança:

1. Para reutilização de implementação
2. Para diferença
3. Para substituição de tipo

Esteja avisado de que alguns tipos de reutilização são mais desejáveis que outros! Vamos explorar cada uso em detalhes.

## Herança para implementação

Você já viu que a herança possibilita que uma nova classe reutilize implementação de outra classe. Em vez de recortar e colar código ou instanciar e usar um componente através de composição, a herança torna o código automaticamente disponível, como parte da nova classe. Como mágica, sua nova classe nasce com funcionalidade.

A hierarquia Employee e a mal guiada Queue/Iterator demonstram a reutilização de implementação. Nos dois casos, a filha reutilizou vários comportamentos encontrados na progenitora.



### DICA

Lembre-se de que, quando programa com herança de implementação, você está preso à implementação que herda. Escolha as classes que herdará com cuidado. Você precisará ponderar as vantagens da reutilização em relação a todos os fatos negativos de reutilizar algumas implementações.

Entretanto, uma classe corretamente definida para herança fará bastante uso de métodos protegidos refinados. Uma classe que herda pode sobrepor esses métodos protegidos para alterar a implementação. A sobreposição pode diminuir o impacto da herança de uma implementação mal feita ou inadequada.

4

## Problemas da herança da implementação

Até aqui, a herança de implementação parece excelente. Cuidado, contudo — o que parece uma técnica útil na superfície se mostra uma prática perigosa no uso. Na verdade, a herança de implementação é a forma mais deficiente de herança e normalmente você deve evitá-la. A reutilização pode ser fácil, mas, conforme você verá, ela tem um alto preço.

Para entender as falhas, você precisa considerar os tipos. Quando uma classe herda de outra, ela assume automaticamente o tipo da classe herdada. A herança de tipo correto sempre deve ter precedência, ao se projetar hierarquias de classe. Você verá os motivos posteriormente, por enquanto, assuma isso como verdade.

Dê uma olhada no exemplo Queue/Iterator novamente. Quando Iterator herda de Queue, ela se torna uma Queue. Isso significa que você pode tratar Iterator como se fosse do tipo Queue. Como Iterator também é uma Queue, ela tem toda a funcionalidade que estava presente na Queue. Isso significa que os métodos como enqueue() e dequeue() também fazem parte da interface pública de Iterator.

Superficialmente, isso não parece ser um problema, mas dê uma olhada melhor na definição de Iterator. Uma interface Iterator simplesmente define dois métodos, um para recuperar um elemento e outro para testar se restam quaisquer elementos no Iterador. Por definição, você não pode adicionar itens em um Iterador; entretanto, Queue define o método enqueue() justamente

para um caso assim. Em vez disso, você só pode remover elementos. Você não pode pegar um elemento e, ao mesmo tempo, deixá-lo dentro da interface Iterator. Novamente, Queue define o método peek(), justamente para esse caso. É simples ver que usar Queue como uma base herdada para Iterator não é uma boa escolha; isso fornece comportamentos que simplesmente não pertencem a uma interface Iterator.

**NOTA**

Algumas linguagens permitem que uma classe simplesmente herde implementação, sem herdar as informações de tipo. Se sua linguagem permite tal herança, então o exemplo Queue/Iterator não é muito problemático. Entretanto, a maioria das linguagens não permite a separação de interface e implementação, durante a herança. Das linguagens que fazem a separação, algumas fazem isso automaticamente. Outras ainda, como C++, permitem a separação, mas exigem que o programador a solicite explicitamente. Tal linguagem exige que o programador projete e solicite a separação explicitamente, enquanto codifica a classe. Obviamente, pode ser muito fácil ignorar o fato de que você precisará separar a implementação e o tipo, se não tomar cuidado.

**NOTA**

Este livro usa uma definição de herança simples. A discussão sobre herança pressupõe que ela inclui implementação e interface, quando uma classe herda de outra.

Uma herança pobre é o monstro de Frankenstein da programação. Quando você usa herança unicamente para reutilização de implementação, sem quaisquer outras considerações, freqüentemente pode acabar com um monstro construído a partir de partes que não se encaixam.

## Herança para diferença

Você viu a herança para diferença no exemplo de `TwoDimensionalPoint` e `ThreeDimensionalPoint`. Você também a viu no exemplo `Employee`.

A programação pela diferença permite que você programe especificando apenas como uma classe filha difere de sua classe progenitora.

**Novo Termo**

*Programação por diferença* significa herdar uma classe e adicionar apenas o código que torne a nova classe diferente da classe herdada.

No caso de `ThreeDimensionalPoint`, você vê que ela difere de sua classe progenitora pelo acréscimo de uma coordenada Z. Para suportar a coordenada Z, `ThreeDimensionalPoint` adiciona dois novos métodos para configurar e recuperar o atributo. Você também vê que `ThreeDimensionalPoint` redefine o método `toString()`.

A programação por diferença é um conceito poderoso. Ela permite que você adicione apenas o código necessário o suficiente para descrever a diferença entre a classe progenitora e a classe filha. Isso permite que você programe através de incrementos.

Um código menor e mais fácil de gerenciar torna seus projetos mais simples. E como você programa menos linhas de código, teoricamente deve introduzir menos erros. Assim, quando programa pela diferença, você escreve código mais correto em um espaço de tempo mais curto. Assim como a herança de implementação, você pode fazer essas alterações incrementais sem alterar o código existente.

Através da herança, existem duas maneiras de programar pela diferença: adicionando novos comportamentos e atributos, e redefinindo comportamentos e atributos antigos. Cada caso é conhecido como especialização. Vamos ver detalhadamente a especialização.

## Especialização

### Novo Termo

*Especialização* é o processo de uma classe filha ser projetada em termos de como ela é diferente de sua progenitora. Quando tudo estiver dito e feito, a definição de classe da filha incluirá apenas os elementos que a tornam diferente de sua progenitora.

Uma classe filha se especializa em relação à sua progenitora, adicionando novos atributos e métodos em sua interface, assim como redefinindo atributos e métodos previamente existentes. A adição de novos métodos ou a redefinição de métodos já existentes permite que a filha expresse comportamentos que são diferentes de sua progenitora.

Não se confunda com o termo especialização. A especialização permite apenas que você adicione ou redefina os comportamentos e atributos que a filha herda de sua progenitora. A especialização, ao contrário do que o nome possa sugerir, não permite que você remova da filha comportamentos e atributos herdados. Uma classe não obtém herança seletiva.

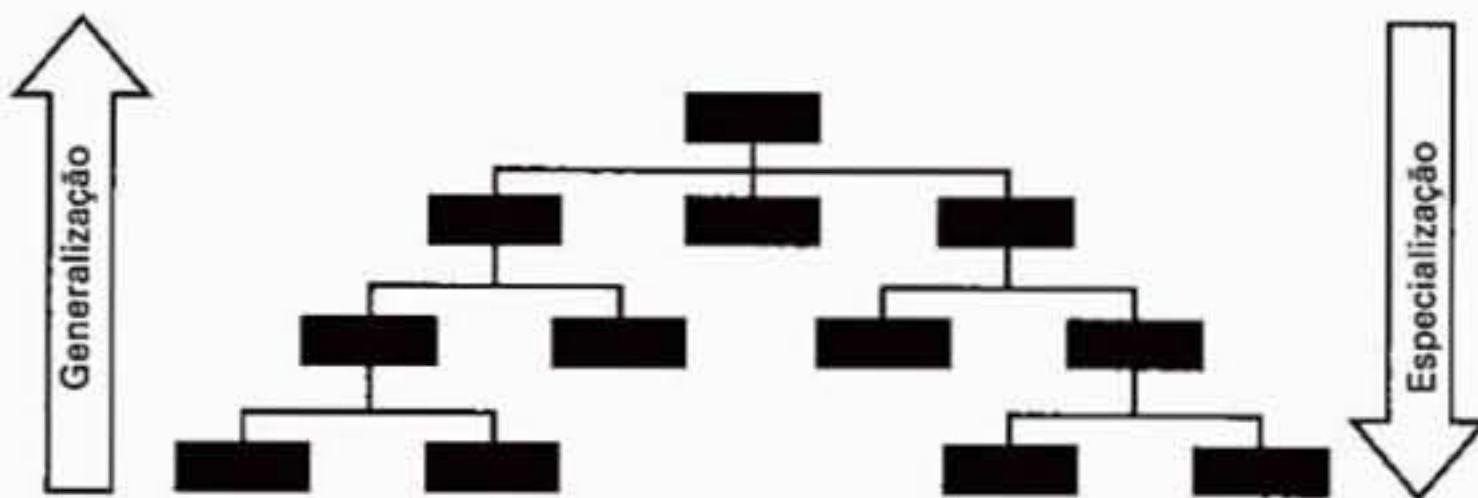
O que a especialização faz é restringir o que pode e o que não pode ser um ponto tridimensional. Um *ThreeDimensionalPoint* *sempre* pode ser um *TwoDimensionalPoint*. Entretanto, é incorreto dizer que um *TwoDimensionalPoint* *sempre* pode ser um *ThreeDimensionalPoint*.

Em vez disso, um *ThreeDimensionalPoint* é uma especialização de um *TwoDimensionalPoint* e um *TwoDimensionalPoint* é uma generalização de um *ThreeDimensionalPoint*.

A Figura 4.7 ilustra a diferença entre generalização e especialização. Quando você percorre uma hierarquia para baixo, você se especializa. Quando você percorre uma hierarquia para cima, você generaliza. À medida que você generaliza, mais classes podem cair sob esse agrupamento. À medida que você especializa, menos classes podem satisfazer todos os critérios para serem classificadas nesse nível.

**FIGURA 4.7**

Quando percorre uma hierarquia para cima, você generaliza. Quando percorre uma hierarquia para baixo, você especializa.



Como você vê, especialização não significa uma restrição de funcionalidade, ela significa restrição da categorização de tipo. A especialização não precisa acabar com `ThreeDimensionalPoint`. Na verdade, ela não precisa necessariamente nem mesmo começar com `TwoDimensionalPoint`. A herança vai ter a profundidade que você quiser. Você pode usar herança para formar estruturas de hierarquia de classe complexas. A noção de hierarquia introduzida anteriormente leva a mais dois termos novos: *ancestral* e *descendente*.



Apenas porque você pode ter hierarquias complicadas não significa que deve tê-las. Você deve se esforçar por ter hierarquias pouco profundas e não hierarquias demasiadamente profundas. À medida que uma hierarquia se aprofunda, ela se torna mais difícil de manter.

**Novo TERMO**

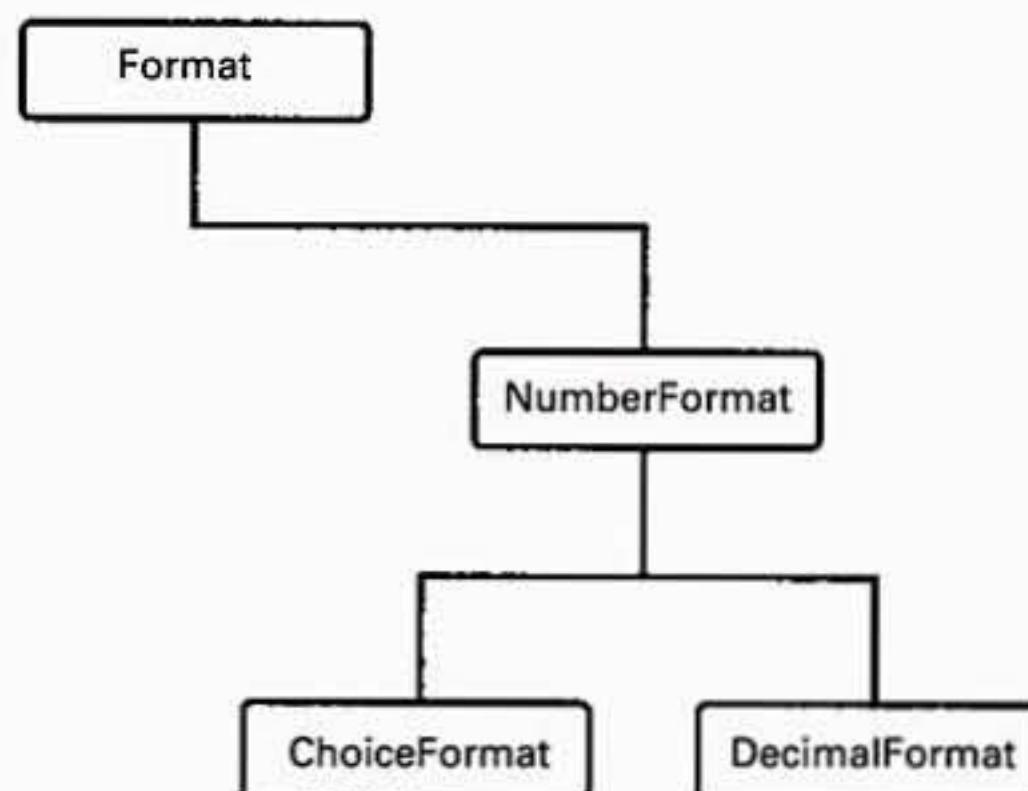
Dada alguma filha, uma *ancestral* é uma classe que aparece na hierarquia de classes antes da progenitora. Conforme a Figura 4.8 ilustra, `Format` é uma ancestral de `DecimalFormat`.

**Novo TERMO**

Dada uma classe, toda classe que aparece depois dela na hierarquia de classes é uma *descendente* da classe dada. Conforme a Figura 4.8 ilustra, `DecimalFormat` é uma descendente de `Format`.

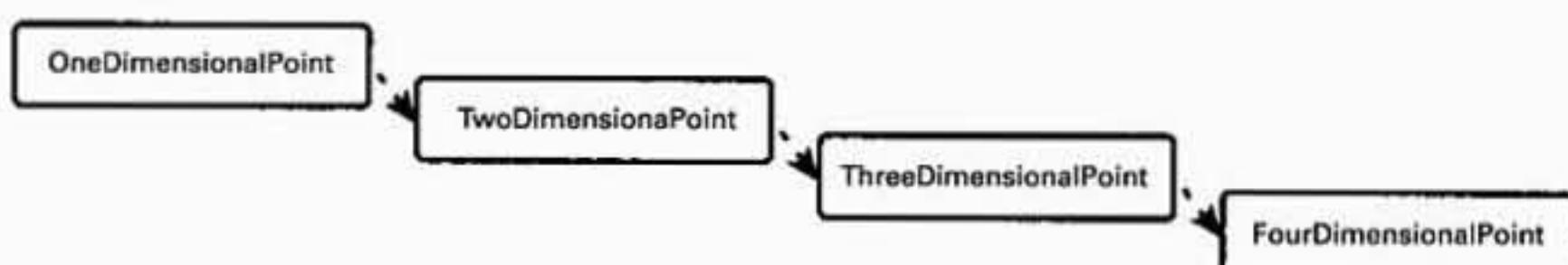
**FIGURA 4.8**

`DecimalFormat` é descendente de `Format`.



Digamos que tivéssemos a hierarquia de herança de classes mostrada na Figura 4.9. Dizemos que `OneDimensionalPoint` é a progenitora de `TwoDimensionalPoint` e ancestral de `ThreeDimensionalPoint` e de `FourDimensionalPoint`. Também podemos dizer que `TwoDimensionalPoint`, `ThreeDimensionalPoint` e `FourDimensionalPoint` são todas descendentes de `OneDimensionalPoint`. Todos os descendentes compartilham os métodos e atributos de seus ancestrais.

**FIGURA 4.9**  
A hierarquia  
de ponto.



Podemos fazer mais algumas declarações interessantes sobre a hierarquia de classes. `OneDimensionalPoint` é a raiz e `FourDimensionalPoint` é uma folha.

**Novo TERMO**

A *classe raiz* (também referida comumente como *classe base*) é a classe superior da hierarquia de herança. A Figura 4.9 mostra que `OneDimensionalPoint` é uma classe raiz.

**Novo TERMO**

Uma *classe folha* é uma classe sem filhas. Na Figura 4.8, `DecimalFormat` é uma classe folha.

É importante notar que as descendentes refletirão as alterações feitas nas ancestrais. Digamos que você encontre um erro em `TwoDimensionalPoint`. Se você corrigir `TwoDimensionalPoint`, todas as classes de `ThreeDimensionalPoint` até `FourDimensionalPoint` tirarão proveito da alteração. Assim, se você corrigir um erro ou tornar uma implementação mais eficiente, todas as classes descendentes da hierarquia tirarão proveito disso.

4

**Herança múltipla**

Em todos os exemplos você viu herança simples. Algumas implementações de herança permitem que um objeto herde diretamente de mais de uma classe. Tal implementação de herança é conhecida como *herança múltipla*. A herança múltipla é um aspecto controverso da POO. Alguns dizem que ela só torna o software mais difícil de entender, projetar e manter. Outros têm grande confiança nela e dizem que uma linguagem não está completa sem ela.

De qualquer modo, a herança múltipla pode ser valiosa, se usada cuidadosa e corretamente. Existem vários problemas introduzidos pela herança múltipla. Entretanto, uma discussão completa do que a herança múltipla pode e não pode fazer está fora dos objetivos deste dia.

## Herança para substituição de tipo

O tipo final de herança é a herança para substituição de tipo. A substituição de tipo permite que você descreva relacionamentos com capacidade de substituição. O que é um relacionamento com capacidade de substituição?

Considere a classe Line:

```
public class Line {  
  
    private TwoDimensionalPoint p1;  
    private TwoDimensionalPoint p2;  
  
    public Line( TwoDimensionalPoint p1, TwoDimensionalPoint p2 ) {  
        this.p1 = p1;  
        this.p2 = p2;  
    }  
  
    public TwoDimensionalPoint getEndpoint1() {  
        return p1;  
    }  
  
    public TwoDimensionalPoint getEndpoint2() {  
        return p2;  
    }  
  
    public double getDistance() {  
        double x =  
            Math.pow( (p2.getXCoordinate() - p1.getXCoordinate()), 2 );  
        double y =  
            Math.pow( (p2.getYCoordinate() - p1.getYCoordinate()), 2 );  
        double distance = Math.sqrt( x + y );  
        return distance;  
    }  
  
    public TwoDimensionalPoint getMidpoint() {  
        double new_x = (p1.getXCoordinate() + p2.getXCoordinate()) / 2;  
        double new_y = (p1.getYCoordinate() + p2.getYCoordinate()) / 2;  
        return new TwoDimensionalPoint( new_x, new_y );  
    }  
}
```

Line recebe dois objetos TwoDimensionalPoint como argumentos e fornece alguns métodos para recuperar os valores, um método para calcular a distância entre os pontos e um método para calcular o ponto médio.

Um relacionamento com capacidade de substituição significa que você pode passar para o construtor de Line *qualquer* objeto que herde de TwoDimensionalPoint.

Lembre-se de que, quando uma filha herda de sua progenitora, você diz que a filha ‘é uma’ progenitora. Assim, como um objeto ThreeDimensionalPoint ‘é um’ objeto TwoDimensionalPoint, você pode passar um objeto ThreeDimensionalPoint para o construtor.

Considere o método main() a seguir:

```
public static void main( String [] args ) {
    ThreeDimensionalPoint p1 = new ThreeDimensionalPoint( 12, 12, 2 );
    TwoDimensionalPoint p2 = new TwoDimensionalPoint( 16, 16 );

    Line l = new Line( p1, p2 );

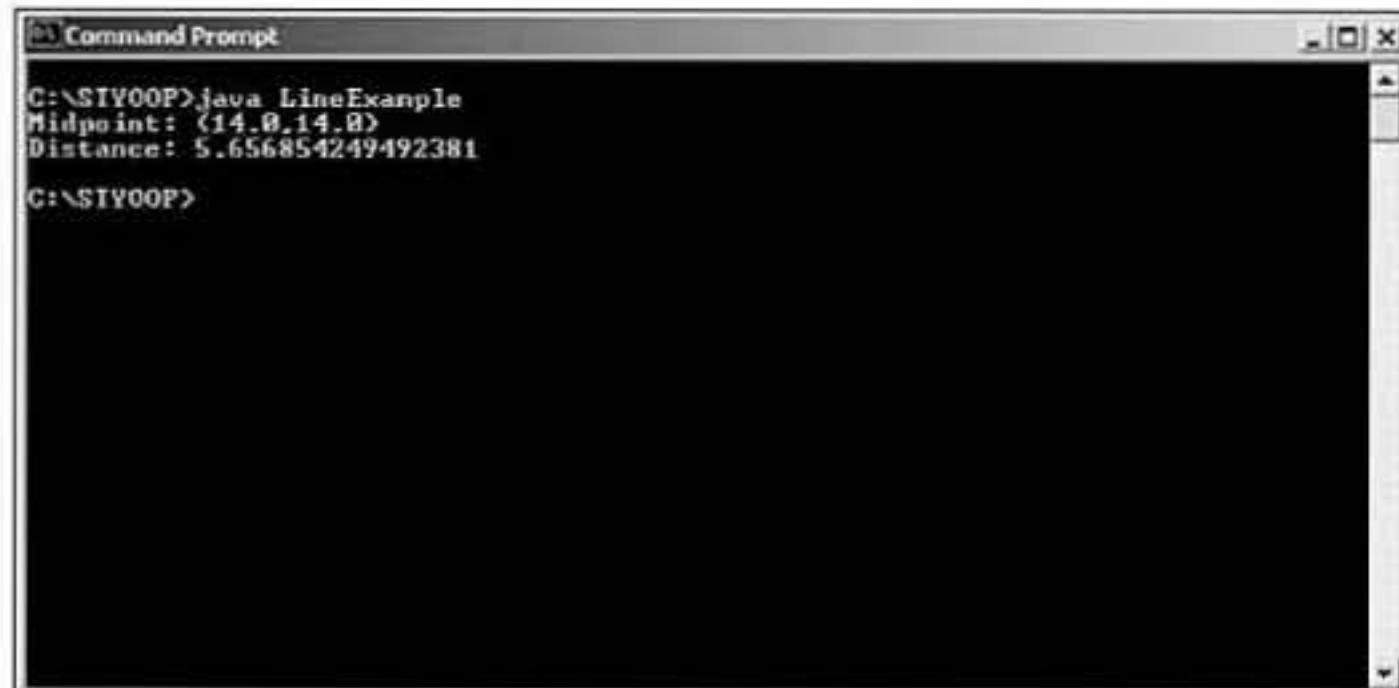
    TwoDimensionalPoint mid = l.getMidpoint();
    System.out.println( "Midpoint: (" +
                        mid.getXCoordinate() +
                        ", " +
                        mid.getYCoordinate() +
                        ")" );
    System.out.println( "Distance: " + l.getDistance() );
}
```

Você notará que o método principal passa um objeto TwoDimensionalPoint e um objeto ThreeDimensionalPoint para o construtor de Line. A Figura 4.10 ilustra o que você verá, se executar o método main().

4

**FIGURA 4.10**

*Testando relacionamentos com capacidade de substituição.*



**NOTA**

Tente imaginar as possibilidades que os relacionamentos com capacidade de substituição oferecem a você. No exemplo da linha, eles poderiam possibilitar uma maneira rápida de trocar de um modo de visualização em 3D para um modo de visualização em 2D em uma GUI.

Capacidade de conexão é um conceito poderoso. Como você pode enviar a uma filha qualquer mensagem que pode ser enviada para sua progenitora, é possível tratá-la como se ela pudesse ser

substituída pela progenitora. Esse é o motivo pelo qual você não deve *remover* comportamentos ao criar uma filha. Se você fizer isso, a capacidade de conexão será invalidada.

Usando a capacidade de conexão, você pode adicionar novos subtipos em seu programa, a qualquer momento. Se seu programa for feito para usar uma ancestral, ele saberá como usar os novos objetos. O programa não precisará se preocupar com o tipo exato do objeto. Desde que tenha um relacionamento com capacidade de substituição com o tipo que espera que possa usar.



Saiba que os relacionamentos com capacidade de substituição só podem ir até um nível acima na hierarquia de herança. Se você programar seu objeto para aceitar determinado tipo de objeto, não poderá passar para ele a progenitora do objeto esperado. Entretanto, você pode passar para ele qualquer descendente.

Pegue como exemplo o construtor de Line:

```
public Line( TwoDimensionalPoint p1,TwoDimensionalPoint p2 )
```

Você pode passar para o construtor um objeto `TwoDemensionalPoint` ou qualquer descendente de `TwoDimensionalPoint`. Entretanto, você não pode passar para o construtor um objeto `OneDimensionalPoint`, pois essa classe aparece antes de `TwoDimensionalPoint` na hierarquia.

**Novo TERMO** Um *subtipo* é um tipo que estende outro tipo através de herança.

A capacidade de substituição aumenta sua oportunidade de reutilização. Digamos que você tenha escrito um pacote para conter objetos `TwoDimensionalPoint`. Devido a capacidade de conexão, você também pode usar o pacote para qualquer descendente de `TwoDimensionalPoint`.

A capacidade de substituição é importante, pois ela permite que você escreva código genérico. Em vez de ter várias instruções case ou testes if/else para ver que tipo de ponto o programa estava usando, você simplesmente programa seus objetos para tratar com objetos do tipo `TwoDimensionalPoint`.

## Dicas para a herança eficaz

A herança vem com seu próprio conjunto de problemas de projeto. Embora seja poderosa, na verdade a herança fornece a corda para você se enforcar, quando usada incorretamente. As dicas a seguir o ajudarão a usar a herança eficazmente:

- Em geral, use herança para reutilização de interface e para definir relacionamentos de substituição. Você também pode usar herança para estender uma implementação, mas somente se a classe resultante passar no teste ‘é um’.
- Em geral, prefira a composição em vez da herança para reutilização de implementação simples. Use herança apenas se você puder aplicar o teste ‘é um’ na hierarquia resultante. Não use herança para reutilização de implementação ambiciosa.
- Sempre use a regra ‘é um’.

As hierarquias de herança corretas não acontecem sozinhas. Freqüentemente, você descobrirá hierarquias à medida que prosseguir. Quando isso acontecer, refaça seu código. Em outras ocasiões, você precisará projetar deliberadamente suas hierarquias. De qualquer modo, existem alguns princípios de projeto a seguir:

- Como regra geral, mantenha suas hierarquias de classe relativamente rasas.
- Projete cuidadosamente a hierarquia de heranças e remova as características comuns das classes base abstratas. As classes base abstratas permitem que você defina um método sem fornecer uma implementação. Como a classe base não especifica uma implementação, você não pode instanciá-la. Entretanto, o mecanismo abstrato obriga uma classe que esteja herdando a fornecer uma implementação. As classes abstratas são valiosas para herança planejada. Elas ajudam o desenvolvedor a ver o que elas precisam implementar.

**NOTA**

Se sua linguagem não fornece um mecanismo abstrato, crie métodos vazios e documente o fato de que subclasses devem implementar completamente esses métodos.

4

- As classes freqüentemente compartilham código comum. Não há sentido em ter várias cópias de código. Você deve remover o código comum e isolá-lo em uma única classe progenitora. Entretanto, não o coloque muito acima. Coloque-o apenas no primeiro nível acima de onde ele é necessário.
- Simplesmente não é possível planejar sempre suas hierarquias completamente. As características comuns não aparecerão até que você escreva o mesmo código algumas vezes. Quando você ver características comuns, não tenha medo de refazer suas classes. Esse trabalho é freqüentemente referido como *refazer*.

O encapsulamento é tão importante entre progenitora e filha quanto entre classes não relacionadas. Não relaxe quanto ao encapsulamento, quando você estiver herdando. A prática de usar uma interface bem definida é tão válida entre progenitora e filha quanto entre classes completamente não relacionadas. Aqui estão algumas dicas que o ajudarão a evitar a quebra do encapsulamento, quando você herdar:

- Use interfaces bem definidas entre a progenitora e a filha, exatamente como as usaria entre classes.
- Se você adicionar métodos especificamente para uso por subclasses, certifique-se de torná-los protegidos, para que apenas a subclasse possa vê-los. Os métodos protegidos permitem que você ofereça às suas subclasses um pouco mais de controle, sem abrir esse controle para toda classe.
- Em geral, evite abrir a implementação interna de seu objeto para subclasses. Uma subclasse pode se tornar dependente da implementação, se você fizer isso. Tal acoplamento tem todos os problemas delineados no Dia 2.

Aqui estão alguns segredos finais para a herança eficaz:

- Nunca se esqueça de que a substituição é o objetivo número um. Mesmo que um objeto deva ‘intuitivamente’ aparecer em uma hierarquia, isso não quer dizer que ele deve aparecer. Apenas porque é possível ou porque sua intuição clama, não significa que você deve fazer isso.
- Programe pela diferença para manter o código fácil de gerenciar.
- Sempre prefira a composição à herança para reutilização de implementação. Geralmente é mais fácil alterar as classes envolvidas na composição.

## Resumo

Existem dois tipos de relacionamentos fornecidos pela POO: um relacionamento de *uso* entre objetos e um relacionamento de herança entre classes. Cada relacionamento fornece uma forma de reutilização. Entretanto, cada um vem com suas próprias vantagens e problemas.

A simples instanciação e uso freqüentemente limitam a flexibilidade de uma classe. Através da reutilização simples, não há meios de reutilizar ou estender uma classe. Em vez disso, você fica com uma instanciação simples ou recorte e colagem. A herança supera essas deficiências, fornecendo um mecanismo interno para a reutilização segura e eficiente do código.

A reutilização de implementação proporciona a você um modo rápido e grosseiro de usar código previamente existente em suas novas classes. Ao contrário da operação de recorte e colagem simples, existe apenas uma cópia do código para manter. Entretanto, simplesmente herdar para reutilizar é imprevidente e limita seus projetos.

A implementação para diferença permite que você programe suas novas classes em termos de como elas diferem da classe original. Você só programa os atributos que diferenciam a filha da progenitora.

Finalmente, a herança para substituição permite que você programe genericamente. Com a substituição, você pode trocar de subclasses para a progenitora a qualquer momento, sem danificar seu código. Isso permite que seu programa seja flexível para futuros requisitos.

## Como a herança atende aos objetivos da OO

A herança preenche cada um dos objetivos da POO. Ela ajuda a produzir software que é:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível
6. Oportuno

Ela atinge esses objetivos, como segue:

- Natural: a herança permite que você modele o mundo mais naturalmente. Através da herança, você pode formar hierarquias de relacionamento complexas entre suas classes. Como seres humanos, nossa tendência natural é querer categorizar e agrupar os objetos que estão em torno de nós. A herança permite que você traga essas tendências para a programação.

A herança também abrange o desejo do programador de evitar trabalho repetitivo. Não faz sentido fazer trabalho redundante.

- Confiável: a herança resulta em código confiável.

A herança simplifica seu código. Quando programa pela diferença, você adiciona apenas o código que descreve a diferença entre a progenitora e a filha. Como resultado, cada classe pode ter código menor. Cada classe pode ser altamente especializada no que faz. Menos código significa menos erros.

A herança permite que você reutilize código bem testado e comprovado, como a base de suas novas classes. A reutilização de código comprovado é sempre mais desejável do que escrever novo código.

Finalmente, o mecanismo de herança em si é confiável. O mecanismo é incorporado à linguagem, de modo que você não precisa construir seu próprio mecanismo de herança e certificar-se de que todo mundo segue suas regras.

Entretanto, a herança não é perfeita. Ao usar subclasses, você deve estar vigilante com relação à introdução de erros sutis destruindo inadvertidamente dependências não expostas. Prossiga com cuidado, quando herdar.

- Reutilizável: a herança auxilia a reutilização. A própria natureza da herança permite que você use classes antigas na construção de novas classes.

A herança também permite que você reutilize classes de maneiras nunca imaginadas pela pessoa que escreveu a classe. Sobrepondo e programando pela diferença, você pode alterar o comportamento de classes existentes e usá-las de novas maneiras.

- Manutenível: a herança auxilia a manutibilidade. A reutilização de código testado significa que você terá menos erros em seu novo código. E quando você encontrar um erro em uma classe, todas as subclasses tirarão proveito da correção.

Em vez de se aprofundar no código e adicionar recursos diretamente, a herança permite que você pegue código previamente existente e o trate como a base da construção de uma nova classe. Todos os métodos, atributos e informações de tipo se tornam parte de sua nova classe. Ao contrário do recorte e colagem, existe apenas uma cópia do código original para manter. Isso ajuda na manutenção, diminuindo a quantidade de código que você precisa manter.

Se você fosse fazer alterações diretamente no código existente, poderia danificar a classe base e afetar partes do sistema que usem essa classe.

- Extensível: a herança torna a extensão ou especialização de classe possível. Você pode pegar uma classe antiga e adicionar nova funcionalidade a qualquer momento. A programação pela diferença e a herança para capacidade de conexão estimulam a extensão de classes.
- Oportuno: a herança o ajuda a escrever código oportuno. Você já viu como a reutilização simples pode diminuir o tempo de desenvolvimento. Programar pela diferença significa que existe menos código para escrever; portanto, você deve terminar mais rapidamente. Capacidade de substituição significa que você pode adicionar novos recursos, sem ter de alterar muito o código já existente.

A herança também pode tornar os testes mais fáceis, pois você só precisa testar a nova funcionalidade e qualquer interação com a funcionalidade antiga.

## Perguntas e respostas

**P** Hoje, foram listados três motivos separados para usar herança. Esses motivos precisam ser mutuamente exclusivos ou posso combiná-los? Por exemplo, quando eu herdo pela diferença, parece que também poderia herdar para implementação.

**R** Não, os motivos por trás da herança não precisam ser mutuamente exclusivos. Você poderia usar herança e acabar satisfazendo cada um dos motivos.

**P** Herdar para reutilização de implementação parece ter uma conotação negativa. A reutilização não é um dos principais motivos para se usar programação orientada a objetos?

**R** A reutilização é apenas um dos objetivos da POO. A POO é uma estratégia de programação que permite modelar as soluções para seus problemas de uma maneira mais natural: através de objetos. Embora a reutilização seja importante, você não deve simplesmente buscá-la, ignorando os outros objetivos da OO. Lembre do exemplo Iterator/Queue. Aquele era um modelo natural de uma interface Iterator? É claro que não!

Além disso, a herança para reutilização de implementação é apenas uma maneira de obter a reutilização. Freqüentemente, a delegação é a melhor maneira de obter reutilização de implementação simples. A herança simplesmente não é a ferramenta correta, se seu objetivo é apenas reutilizar uma implementação. A herança é a ferramenta correta quando você quer programar pela diferença ou estabelecer capacidade de substituição de tipo.

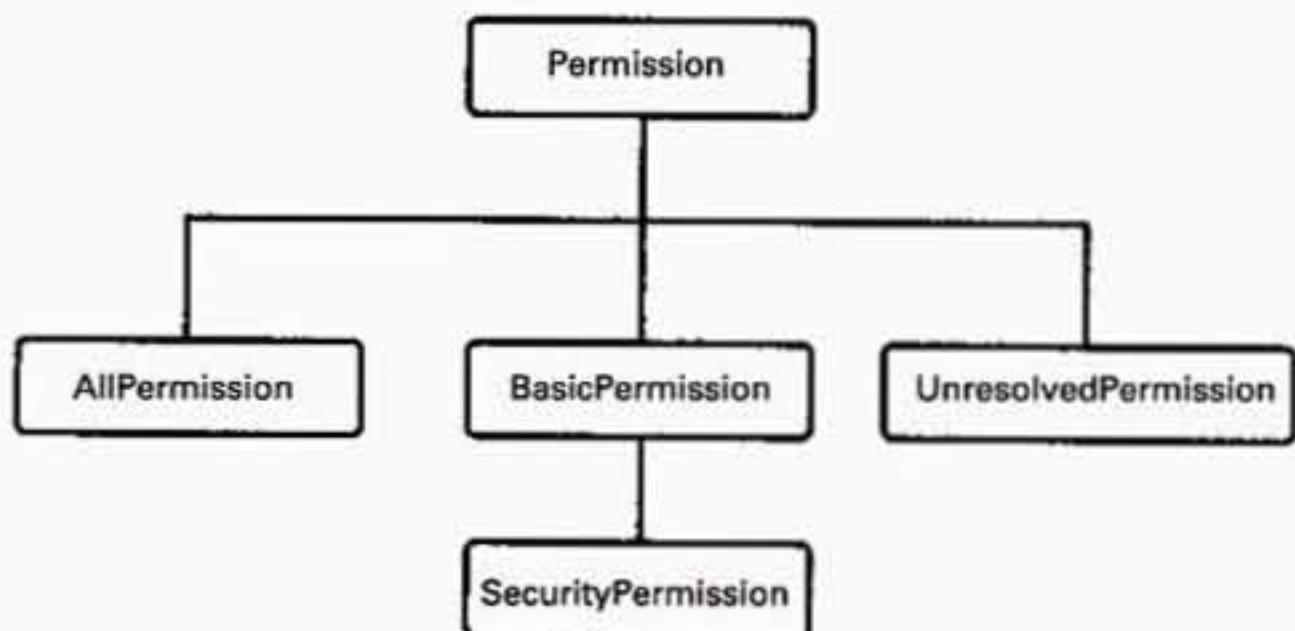
## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Quais são algumas das limitações da reutilização simples?
2. O que é herança?
3. Quais são as três formas de herança?
4. Por que a herança de implementação é perigosa?
5. O que é programação pela diferença?
6. Ao herdar uma classe, pode-se ter três tipos de métodos e atributos. Quais são esses três tipos de atributos e métodos?
7. Quais vantagens a programação pela diferença oferece?
8. Considere a hierarquia da Figura 4.11, extraída da segurança do Java.

**FIGURA 4.11**  
A hierarquia Permission.



4

Se você voltar sua atenção para a classe `Permission`, quais classes são suas filhas? Quais são descendentes?

Considerando a hierarquia inteira, qual classe é a classe raiz? Quais classes são classes folhas?

Finalmente, `Permission` é uma ancestral de `SecurityPermission`?

9. O que é herança para substituição de tipo?
10. Como a herança pode destruir o encapsulamento? Como você pode impor o encapsulamento ao usar herança?

## Exercícios

1. Dada a definição da classe a seguir, quais problemas poderiam ocorrer, se ela fosse herdada?

```
public class Point {
    public Point( int x, int y ) {
```

```
    this.x = x;
    this.y = y;
}
public Point getLocation() {
    return new Point( x, y );
}
public void move( int x, int y ) {
    this.x = x;
    this.y = y;
}
public void setLocation( int x, int y ) {
    this.x = x;
    this.y = y;
}
public void setLocation( Point p ) {
    this.x = p.x;
    this.y = p.y;
}
public int x;
public int y;
}
```

2. Como você evitaria esses problemas?

# SEMANA 1

## DIA 5

### Herança: hora de escrever algum código

A herança é uma ferramenta poderosa. Hoje, você vai explorar o uso dessa nova ferramenta, através de vários exercícios práticos de laboratório. No final da lição de hoje, você deverá se sentir um pouco mais à vontade com a teoria apresentada no Dia 4.

Hoje você aprenderá:

- Como usar herança enquanto programa
- Como as classes abstratas o ajudam a planejar a herança
- Sobre a importância do relacionamento ‘é um’ e ‘tem um’
- Como a linguagem Java pode ter violado os relacionamentos ‘é um’ e ‘tem um’

### Laboratório 1: herança simples

A Listagem 5.1 apresenta a classe base MoodyObject personificada.

---

#### **LISTAGEM 5.1** MoodyObject.java

```
public class MoodyObject {  
  
    // retorna o humor  
    protected String getMood() {
```

**LISTAGEM 5.1** MoodyObject.java (*continuação*)

```
        return "moody";
    }

    // pergunta ao objeto como ele se sente
    public void queryMood() {
        System.out.println("I feel " + getMood() + " today!");
    }
}
```

---

MoodyObject define um método público: queryMood(). queryMood() imprime o humor do objeto na linha de comando. MoodyObject também declara um método protegido, getMood(). queryMood() usa getMood() internamente para obter o humor que coloca em sua resposta. As subclasses podem simplesmente sobrepor getMood() para especializar seu humor.

Se uma subclasse quisesse mudar a mensagem escrita na linha de comando, ela precisaria sobrepor queryMood().

## Exposição do problema

Neste laboratório, você criará duas subclasses: SadObject e HappyObject. As duas subclasses devem sobrepor getMood() para fornecerem seu próprio humor especialmente personalizado.

SadObject e HappyObject também devem adicionar alguns métodos próprios. SadObject deve adicionar um método: public void cry(). Do mesmo modo, HappyObject deve adicionar um método: public void laugh(). laugh() deve escrever ‘hahaha’ na linha de comando. Do mesmo modo, cry() deve escrever ‘boo hoo’ na linha de comando.

A Listagem 5.2 configura um driver de teste que você deve compilar e executar quando tiver concluído a escrita de HappyObject e SadObject.

**LISTAGEM 5.2** MoodyDriver.java

```
public class MoodyDriver {
    public final static void main( String [] args ) {
        MoodyObject moodyObject = new MoodyObject();
        SadObject sadObject = new SadObject();
        HappyObject happyObject = new HappyObject();

        System.out.println( "How does the moody object feel today?" );
        moodyObject.queryMood();
        System.out.println( "" );
        System.out.println( "How does the sad object feel today?" );
        sadObject.queryMood(); //note que a sobreposição muda o humor
        SadObject.cry();
        System.out.println( "" );
```

**LISTAGEM 5.2** MoodyDriver.java (*continuação*)

```
System.out.println( "How does the happy object feel today?" );
happyObject.queryMood(); //note que a sobreposição muda o humor
happyObject.laugh();
System.out.println( "" );
}
```

{



A próxima seção discute as soluções do Laboratório 1. Não prossiga até concluir o Laboratório 1.

## Soluções e discussão

As listagens 5.3 e 5.4 apresentam uma solução para o laboratório.

**LISTAGEM 5.3** HappyObject.java

```
public class HappyObject extends MoodyObject {

    // redefine o humor da classe
    protected String getMood() {
        return "happy";
    }

    // especialização
    public void laugh() {
        System.out.println("hehehe...ahaha...HAHAHAHAHAHA!!!!!");
    }
}
```

5

**LISTAGEM 5.4** SadObject.java

```
public class SadObject extends MoodyObject {

    // redefine o humor da classe
    protected String getMood() {
        return "sad";
    }

    // especialização
    public void cry() {
```

**LISTAGEM 5.4** SadObject.java (*continuação*)

```
    System.out.println("'wah' 'boo hoo' 'weep' 'sob' 'weep');");
}
}
```

Quando você executar o driver de teste, deverá ver uma saída semelhante a da Figura 5.1.

**FIGURA 5.1**

A saída correta de MoodyDriver.



```
C:\STYOOOP>java MoodyDriver
How does the moody object feel today?
I feel moody today!
How does the sad object feel today?
I feel sad today!
'wah' 'boo hoo' 'weep' 'sob' 'weep'
How does the happy object feel today?
I feel happy today!
hehehe... hahaha... HAHAHAHAHAHAT!!!!
C:\STYOOOP>
```

De interesse é a chamada de `queryMood()`. Quando você chama `queryMood()` em `SadObject`, “I feel sad today!” é impresso na tela. Do mesmo modo, `HappyObject` imprime, “I feel happy today!” Tal comportamento pode parecer surpreendente, pois nenhuma classe redefine `queryMood()`.

Você precisa ver `queryMood()` detalhadamente. Internamente, `queryMood()` chama `getMood()` para obter o humor. Como as subclasses redefinem `getMood()`, `queryMood()` chamará a versão filha de `getMood()`. Esse comportamento é um exemplo do processo ilustrado na Figura 4.6 do Dia 4.

## Laboratório 2: usando classes abstratas para herança planejada

Existirão ocasiões em que você desejará desenvolver uma classe especificamente para que outros possam herdar dela. Quando você desenvolve algumas classes relacionadas, pode encontrar código que é comum a todas as suas classes. A boa prática diz que, ao ver código comum, você o coloca em uma classe base. Quando você escreve essa classe base, planeja para que outras classes herdem dela.

Entretanto, uma vez que você tiver terminado de mover o código, poderá notar que não faz sentido instanciar a classe base diretamente. Embora a classe base contenha código comum, que é

muito valioso para subclasses, ela pode não ter nenhum valor para instanciação e uso direto. Em vez disso, só faz sentido usar as subclasses. As subclasses se especializam em relação à classe base e fornecem o que está faltando.

Considere a classe Employee:

```
public class Employee {  
  
    private String first_name;  
    private String last_name;  
    private double wage;  
  
    public Employee( String first_name, String last_name, double wage ) {  
        this.first_name = first_name;  
        this.last_name = last_name;  
        this.wage = wage;  
    }  
  
    public double getWage() {  
        return wage;  
    }  
  
    public String getFirstName() {  
        return first_name;  
    }  
  
    public String getLastname() {  
        return last_name;  
    }  
  
    public double calculatePay() {  
        // Não sei como fazer isso!  
        return 0;  
    }  
  
    public String printPaycheck() {  
        String full_name = last_name + ", " + first_name;  
        return ( "Pay: " + full_name + " $" + calculatePay() );  
    }  
}
```

Você pode usar Employee como classe base de CommissionedEmployees, HourlyEmployees e SalariedEmployees. Cada subclass sabe como calcular seu pagamento. Entretanto, o algoritmo usado para calcular pagamento vai variar de acordo com o tipo de funcionário. Quando criamos essa hierarquia, imaginamos que cada subclass precisaria definir seu próprio método calculatePay().

Há um pequeno problema: `Employee` não tem nenhuma regra para calcular seu pagamento. Não faz sentido executar `calculatePay()` para um objeto `Employee`. Não existe nenhum algoritmo para calcular o pagamento de um funcionário genérico.

Uma solução é não definir `calculatePay()` na classe base. Entretanto, não definir o método na classe base seria uma decisão infeliz. Isso não modela um funcionário muito bem. Cada funcionário saberá calcular seu pagamento. A única diferença é a implementação real do método `calculatePay()`. Assim, na verdade, o método pertence à classe base.

Se você não definir `calculatePay()` na classe base, não poderá tratar os funcionários genericamente. Você perderá a capacidade de conexão de subtipo para o método `calculatePay()`. Outra solução é simplesmente codificar um retorno enlatado. O método poderia simplesmente retornar `wage`.

Um retorno codificado não é uma solução muito limpa. Não há garantias de que outro desenvolvedor vá se lembrar de sobrepor o método, ao desenvolver uma nova subclasse. Além disso, não faz nenhum sentido instanciar um objeto `Employee`. Felizmente, a POO oferece um tipo especial de classe, destinada especificamente à herança planejada: a classe abstrata.

Uma classe abstrata é muito parecida com qualquer outra definição de classe. A definição da classe pode definir comportamentos e atributos, exatamente como uma classe normal. Entretanto, você não pode instanciar diretamente uma classe abstrata, pois uma classe abstrata pode deixar alguns métodos indefinidos.

**Novo Termo**

Um método declarado, mas não implementado, é chamado de *método abstrato*. Somente classes abstratas podem ter métodos abstratos.

Em vez disso, você pode instanciar apenas as descendentes da classe abstrata que realmente implementam os métodos abstratos.

Vamos ver uma classe `Employee` abstrata:

```
public abstract class Employee {  
    ...  
    public abstract double calculatePay();  
    // o restante da definição permanece igual  
}
```

A classe abstrata `Employee` define um método `calculatePay()`; entretanto, ela o deixa indefinido. Agora, fica por conta de cada subclasse implementar realmente o método. `HourlyEmployee` é uma dessas subclasses:

```
public class HourlyEmployee extends Employee {  
  
    private int hours;    //controla o número de horas trabalhadas  
  
    public HourlyEmployee( String first_name, String last_name, double wage ) {  
        super( first_name, last_name, wage );  
    }  
}
```

```
}

public double calculatePay() {
    return getWage() * hours;
}

public void addHours( int hours ) {
    this.hours = this.hours + hours;
}

public void resetHours() {
    hours = 0;
}
}
```

Declarando métodos abstratos, você obriga suas subclasses a se especializarem em relação à classe base, fornecendo uma implementação para os métodos abstratos. Tornando uma classe base abstrata e criando métodos abstratos, você planeja com antecipação o que a subclasse deve redefinir.

## Exposição do problema

No Laboratório 1, você criou uma classe MoodyObject. Todas as subclasses redefinem getMood(). Para o Laboratório 2, altere um pouco essa hierarquia. Torne o método getMood() abstrato. Você também precisará atualizar a classe MoodyDriver para que ela não tente mais instanciar MoodyObject diretamente. Você não terá de fazer quaisquer alterações em SadObject ou HappyObject, pois elas já fornecem uma implementação de getMood().



A próxima seção discute as soluções do Laboratório 2. Não prossiga até concluir o Laboratório 2.

5

## Soluções e discussão

As listagens 5.5 e 5.6 apresentam as definições de MoodyObject e MoodyDriver reestruturadas.

### LISTAGEM 5.5 MoodyObject.java

```
public classe abstrata MoodyObject {

    // retorna o humor
    protected abstract String getMood();

    // pergunta ao objeto como ele se sente
    public void queryMood() {
```

**LISTAGEM 5.5** MoodyObject.java (*continuação*)

```
    System.out.println("I feel " + getMood() + " today!");  
}  
}
```

---

**LISTAGEM 5.6** MoodyDriver.java

```
public class MoodyDriver {  
    public final static void main( String [] args ) {  
        //MoodyObject mo = new MoodyObject(); //não pode instanciar MoodyObject  
        SadObject so = new SadObject();  
        HappyObject ho = new HappyObject();  
  
        //System.out.println( "How does the moody object feel today?" );  
        //mo.queryMood();  
        //System.out.println( "" );  
        System.out.println( "How does the sad object feel today?" );  
        so.queryMood(); // note que a sobreposição muda o humor  
        so.cry();  
        System.out.println( "" );  
        System.out.println( "How does the happy object feel today?" );  
        ho.queryMood(); // note que a sobreposição muda o humor  
        ho.laugh();  
        System.out.println( "" );  
    }  
}
```

---

As alterações são muito simples. MoodyObject define um método `getMood()` abstrato e deixa por conta de suas subclasses fornecer a implementação real. Quando o método `queryMood()` precisa recuperar o humor, ele simplesmente faz uma chamada ao método abstrato.

O uso de classes abstratas define o contrato que as subclasses devem atender para utilizar a classe base. Como desenvolvedor, quando você ver uma classe base abstrata, saberá exatamente o que precisa especializar ao herdar. Você pode especializar os métodos abstratos. Entretanto, você sabe que definindo os métodos abstratos, sua nova classe se encaixará na hierarquia corretamente.

Quando uma classe base tem muitos métodos, pode ser confuso descobrir quais deles devem ser sobrepostos. As classes abstratas fornecem uma dica.

## Laboratório 3: conta em banco — praticando a herança simples

Agora é hora de testar seus conhecimentos de herança. Vamos voltar ao Banco OO e ver o que a herança pode fazer para o sistema de contas do banco.

O Banco OO oferece aos seus clientes algumas escolhas de contas: uma conta poupança, uma conta com cheques, uma conta com vencimento programado e uma conta de cheque especial.

### Uma conta genérica

Cada tipo de conta permite ao usuário depositar e sacar fundos, assim como verificar o saldo corrente. A conta básica genérica não permite saque a descoberto.

### A conta poupança

A conta poupança especializa a conta genérica do banco, aplicando juros no saldo, quando instruída a fazer isso. Por exemplo, se um depositante tem um saldo de US\$1.000 e a taxa de juros é de 2%, após o pagamento dos juros, o saldo será de US\$1020:

```
balance = balance + (balance * interest_rate)
```

A conta poupança não permite saque a descoberto.

### Uma conta com vencimento programado

A conta com vencimento programado também aplica juros ao saldo. Entretanto, se o titular da conta fizer qualquer saque do capital investido, antes do prazo de vencimento, o banco deduzirá uma porcentagem do saque. Assim, por exemplo, se o depositante sacar US\$1.000 antes do prazo de vencimento e houver uma multa de 5% sobre o valor sacado, o saldo da conta diminuirá US\$1000. Entretanto, o depositante receberá apenas US\$950. Se a conta estiver no vencimento, o banco não penalizará os saques.

```
balance = balance - withdraw_amount
```

mas

```
amount_given_to_depositor = amount - (amount * penalty_rate)
```

a conta com vencimento programado não permite saque a descoberto.

5

### Conta com cheques

Ao contrário das contas poupança e com vencimento programado, a conta com cheques não aplica juros ao saldo. Em vez disso, a conta com cheques permite que o depositante emita cheques e faça transações na conta através de ATM (caixa eletrônico de auto-atendimento). Entretanto, o

banco limita o número de transações mensais a algum número fixo. Se o depositante ultrapassar essa quota mensal, o banco cobrará uma taxa por transação. Assim, por exemplo, se o depositante tiver direito a cinco transações gratuitas por mês, mas fizer oito transações a uma taxa de US\$1 por transação, o banco cobrará do depositante uma taxa de US\$3:

```
fee = (total_transactions - monthly_quota) * per_transaction_fee
```

A conta com cheques não permite saque a descoberto.

## Conta com cheque especial

Finalmente, a conta com cheque especial permite ao depositante sacar dinheiro além do saldo da conta. Entretanto, nada é de graça. Periodicamente, o banco aplicará uma taxa de juros no caso de qualquer saldo negativo. Assim, por exemplo, se o depositante acumular um saldo negativo de US\$1.000 a uma taxa de 20%, poderá pagar uma taxa de US\$200. Depois da aplicação da taxa, seu saldo será de -US\$1200:

```
balance = balance + (balance * interest_rate)
```

Note que o banco só calcula juros em contas com saldo negativo! Caso contrário, o banco acabaria distribuindo dinheiro. O Banco OO não está no ramo de distribuição de dinheiro. Nem mesmo para desenvolvedores.

Ao contrário da conta com cheques, a conta com cheque especial não coloca um limite no número de transações mensais. O banco estimularia os saques — eles poderiam cobrar juros!

## Exposição do problema

Sua tarefa é formular uma hierarquia de herança e implementar as contas conforme definido anteriormente. Você deve criar as seguintes classes de conta:

- BankAccount
- SavingsAccount
- TimeMaturityAccount
- CheckingAccount
- OverdraftAccount

BankAccount é a classe base. Ela contém as tarefas comuns a todas as contas. Essa é a única dica hierárquica que você terá! Parte do laboratório é você experimentar as hierarquias de herança.

Existem várias simplificações que você pode fazer. Para cálculos de taxas, vencimento programado e juros, suponha que outra pessoa observará o calendário. Não programe esse tipo de funcionalidade em suas classes. Em vez disso, forneça um método para outro objeto chamar. Por exemplo, SavingsAccount deve ter um método addInterest(). Um objeto externo chamará o método, quando for hora de calcular os juros. Do mesmo modo, CheckingAccount deve exportar um método accessFees(). Quando chamado, esse método calculará todas as taxas e as aplicará no saldo.

**NOTA**

Não se atrapalhe com detalhes desnecessários. Lembre-se de que você está completando este laboratório para ganhar experiência prática com herança e não para escrever o sistema de conta mais robusto possível. Assim, não se preocupe com a validação da entrada (a não ser que queira fazer isso). Você pode supor que todos os valores dos argumentos serão sempre válidos.

O Dia 4 abordou brevemente o uso de `super`. `super` não é um conceito difícil. Considere a seleção a seguir:

```
public CommissionedEmployee( String first_name, String last_name,
                             double wage, double commission ) {
    super(first_name, last_name, wage ); // chama o construtor original
                                         // para inicializar corretamente
    this.commission = commission;
}
```

Quando você chama `super` dentro de um construtor, isso permite que o construtor da progenitora seja chamado. É claro que você deve fornecer todos os argumentos exigidos pelo construtor da progenitora. A maioria das linguagens, Java incluída, exige que, se você chamar `super` dentro do construtor, então deve fazer isso antes de tudo. Na verdade, se você não chamar `super`, automaticamente a linguagem Java tentará chamar `super()` sozinha.

`super` permite que você destaque o código da progenitora, que de outro modo seria simplesmente sobreescrito. No caso dos construtores, `super` permite que a filha chame o construtor de sua progenitora.

Chamar corretamente o construtor da progenitora é algo que você não deve desprezar. Você precisa garantir que a classe seja inicializada corretamente.

Você também pode usar `super` dentro de um método.

Imagine uma classe `VeryHappyObject`:

```
public class VeryHappyObject extends HappyObject {

    // redefine o humor da classe
    protected String getMood() {
        String old_mood = super.getMood();
        return "very" + old_mood;
    }
}
```

`VeryHappyObject` sobrepõe `getMood()`. Entretanto, `super.getMood()` permite que `VeryHappyObject` chame a versão da progenitora de `getMood()`. `VeryHappyObject` especializa o método `getMood()` de sua progenitora, realizando algum processamento extra no valor retornado por `super.getMood()`.

Assim, mesmo que uma filha sobreponha o método de sua progenitora, a filha ainda poderá destacar o código existente na progenitora.

Assim como em um construtor, se você usar super.<method>() para chamar um método, então deve fornecer todos os argumentos que o método possa exigir. Você achará super útil neste laboratório.

Pare agora e complete o laboratório, caso se sinta à vontade. Se você precisar de um pouco mais de ajuda, continue a ler.

## Exposição estendida do problema

Se você ainda se acha perdido, estas interfaces devem ajudá-lo um pouco. Essas interfaces representam apenas uma maneira de completar o laboratório.

`BankAccount` contém os seguintes métodos:

```
public void depositFunds( double amount )
public double getBalance()
public double withdrawFunds( double amount )
protected void setBalance( double newBalance )
```

`SavingsAccount` deve conter os seguintes métodos:

```
public void addInterest()
public void setInterestRate( double interestRate )
public double getInterestRate()
```

`TimedMaturityAccount` contém os seguintes métodos:

```
public boolean isMature()
public void mature()
public double getFeeRate()
public void setFeeRate( double rate )
```

`TimedMaturityAccount` precisará redefinir `withdrawFunds()` para verificar o vencimento e aplicar todas as taxas necessárias.

`CheckingAccount` contém os seguintes métodos:

```
public void accessFees()
public double getFee()
public void setFee( double fee )
public int getMonthlyQuota()
public void setMonthlyQuota( int quota )
public int getTransactionCount()
```

`CheckingAccount` precisará sobrepor `withdrawFunds()` para controlar o número de transações.

OverdraftAccount contém os seguintes métodos:

```
public void chargeInterest()  
public double getCreditRate()  
public void setCreditRate( double rate )
```

OverdraftAccount pode precisar sobrepor withdrawFunds(), caso BankAccount coloque cheques a descoberto no método.

Talvez você também queira iniciar sua hierarquia com a classe Account, que desenvolveu para o Laboratório 2 no Dia 3. A única mudança que você deverá fazer é no método withdrawFunds(). Você provavelmente deve colocar proteção contra saque a descoberto nos métodos withdrawFunds().



A próxima seção discute as soluções do Laboratório 3. Não prossiga até concluir o Laboratório 3.

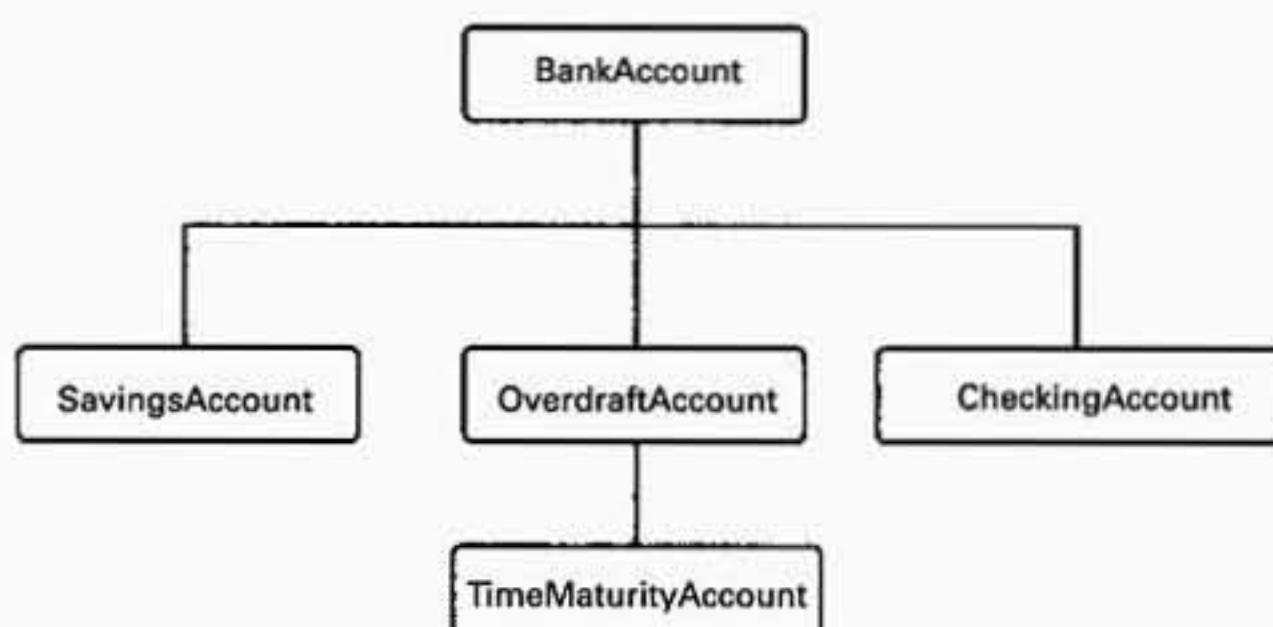
5

## Soluções e discussão

A Figura 5.2 ilustra a hierarquia de herança resultante de conta.

**FIGURA 5.2**

*A hierarquia de conta em banco.*



É importante ter essa hierarquia em mente, enquanto você considera as soluções a seguir.

A Listagem 5.6 apresenta uma possível implementação de BankAccount. Essa classe base controla o saldo e manipula depósitos e saques.

### **LISTAGEM 5.7** BankAccount.java

```
public class BankAccount {  
  
    // dados privados  
    private double balance;
```

**LISTAGEM 5.7** BankAccount.java (*continuação*)

```
// construtor
public BankAccount( double initDeposit ) {
    setBalance( initDeposit );
}
// deposita dinheiro na conta
public void depositFunds( double amount ) {
    // a classe base não aplica regras às contas
    // não valida a entrada
    setBalance( getBalance() + amount );
}
// consulta o saldo
public double getBalance() {
    return balance;
}
// configura o saldo
protected void setBalance( double newBalance ) {
    balance = newBalance;
}
// saca fundos da conta
public double withdrawFunds( double amount ) {
    if( amount >= balance ) {
        amount = balance;
    }
    setBalance( getBalance() - amount );

    return amount;
}
```

---

SavingsAccount, na Listagem 5.8, herda diretamente de BankAccount. SavingsAccount especializa BankAccount, adicionando métodos para obter e configurar a taxa de juros, assim como um método para aplicar juros no saldo da conta.

**LISTAGEM 5.8** SavingsAccount.java

```
public class SavingsAccount extends BankAccount {

    // dados privados
    private double interestRate;

    // Cria novo SavingsAccount
    public SavingsAccount(double initBalance, double interestRate ){
        super( initBalance );
        setInterestRate(interestRate );
```

**LISTAGEM 5.8** SavingsAccount.java (*continuação*)

```
}

// calcula e soma juros na conta
public void addInterest() {
    double balance = getBalance();
    double rate = getInterestRate();
    double interest = balance * rate;

    double new_balance = balance + interest;

    setBalance( new_balance );
}

// configura a taxa de juros
public void setInterestRate( double interestRate ) {
    this.interestRate = interestRate;
}

// consulta a taxa de juros
public double getInterestRate() {
    return interestRate;
}

}
```

TimeMaturityAccount, na Listagem 5.9, herda de SavingsAccount, pois juros podem ser aplicados em seu saldo. Entretanto, ela especializa sua progenitora, definindo métodos para configurar o nível de vencimento e taxas. Interessante é o fato de que essa classe redefine o método withdrawFunds(). Através de uma chamada a super.withdrawFunds(), esse método ainda usa a funcionalidade original; entretanto, ela acrescenta as verificações necessárias para ver se precisa acessar uma taxa para a transação. Se assim for, ela acessa a taxa e retorna o valor do saque, menos a taxa.

5

**LISTAGEM 5.9** TimeMaturityAccount.java

```
class TimedMaturityAccount extends SavingsAccount {

    // dados privados
    private boolean mature;
    private double feeRate;

    // Cria novo TimedMaturityAccount
    public TimedMaturityAccount( double initBalance,
                                double interestRate,
                                double feeRate ) {
        super( initBalance, interestRate );
        setFeeRate( feeRate );
    }
}
```

**LISTAGEM 5.9** TimeMaturityAccount.java (*continuação*)

```
// sobrepõe withdrawFunds de BankAccount
public double withdrawFunds( double amount ) {
    super.withdrawFunds( amount );
    if( !isMature() ) {
        double charge = amount * getFeeRate();
        amount = amount - charge;
    }
    return amount;
}
// verifica o vencimento
public boolean isMature() {
    return mature;
}
// faz vencimento
public void mature() {
    mature = true;
}
// % de taxa para saque antecipado
public double getFeeRate() {
    return feeRate;
}
// configura % de taxa para saque antecipado
public void setFeeRate( double rate ) {
    feeRate = rate;
}
}
```

---

CheckingAccount, na Listagem 5.10, herda diretamente da classe base BankAccount. Essa classe adiciona os métodos necessários para configurar a taxa por transação, configurar a quota mensal, reconfigurar a conta da transação e consultar o número de transações corrente. Essa classe também sobrepõe o método withdrawFunds() para controlar o número de transações. Assim como TimedMaturityAccount, CheckingAccount ainda usa a lógica original, chamando super.withdrawFunds().

**LISTAGEM 5.10** CheckingAccount.java

```
public class CheckingAccount extends BankAccount {

    // dados privados
    private int    monthlyQuota;
    private int    transactionCount;
    private double fee;

    // Cria novo CheckingAccount
```

**LISTAGEM 5.10** CheckingAccount.java (*continuação*)

```
public CheckingAccount( double initDeposit, int trans, double fee ) {  
    super( initDeposit );  
    setMonthlyQuota( trans );  
    setFee( fee );  
}  
// sobrepõe withdrawFunds de BankAccount  
public double withdrawFunds( double amount ) {  
    transactionCount++;  
    return super.withdrawFunds( amount );  
}  
// acessa as taxas se ultrapassou o limite de transação  
public void accessFees() {  
    int extra = getTransactionCount() - getMonthlyQuota();  
    if( extra > 0 ) {  
        double total_fee = extra * getFee();  
        double balance = getBalance() - total_fee;  
        setBalance( balance );  
    }  
    transactionCount = 0;  
}  
// alguns métodos de obtenção e configuração  
public double getFee() {  
    return fee;  
}  
public void setFee( double fee ){  
    this.fee = fee;  
}  
public int getMonthlyQuota() {  
    return monthlyQuota;  
}  
public void setMonthlyQuota( int quota ) {  
    monthlyQuota = quota;  
}  
public int getTransactionCount() {  
    return transactionCount;  
}  
}
```

5

Finalmente, OverdraftAccount, na Listagem 5.11, herda diretamente de BankAccount. Entretanto, ela também adiciona métodos para configurar a taxa de juros de saque a descoberto e para aplicar quaisquer taxas de juros.

**LISTAGEM 5.11** OverdraftAccount.java

```
public class OverdraftAccount extends BankAccount {

    // dados privados
    private double creditRate;

    // Cria novo OverdraftAccount
    public OverdraftAccount( double initDeposit, double rate ) {
        super( initDeposit );
        setCreditRate( rate );
    }

    // cobra juros sobre qualquer dinheiro emprestado
    public void chargeInterest() {
        double balance = getBalance();
        if( balance < 0 ) {
            double charge = balance * getCreditRate();
            setBalance( balance + charge );
        }
    }

    // consulta a taxa de crédito
    public double getCreditRate() {
        return creditRate;
    }

    // configura a taxa de crédito
    public void setCreditRate( double rate ) {
        creditRate = rate;
    }

    // saca fundos da conta
    public double withdrawFunds( double amount ) {
        setBalance( getBalance() - amount );

        return amount;
    }
}
```

---

Cada uma dessas classes especializa sua progenitora de uma maneira ou de outra. Algumas, como SavingsAccount, simplesmente adicionam novos métodos. Outras, como CheckingAccount, OverdraftAccount e TimedMaturityAccount, sobrepõem o comportamento padrão da progenitora, para aumentar a funcionalidade.

Este laboratório o expõe aos mecanismos da herança, assim como à herança para reutilização de implementação e à programação pela diferença.

Embora não seja mostrada aqui, você também pode usar a capacidade de conexão, pois a classe comum BankAccount se relaciona a todas as contas. Qualquer um que saiba como atuar na classe base BankAccount pode sacar, depositar e emitir um cheque nos fundos de qualquer tipo de conta. Você vai explorar a capacidade de conexão com detalhes por todo o Dia 6, “Polimorfismo: aprendendo a prever o futuro”, e Dia 7, “Polimorfismo: hora de escrever algum código”.

## Laboratório 4: estudo de caso — “é um”, “tem um” e `java.util.Stack`

Quando se é iniciante em OO, pode ser tentador ver a linguagem Java como um exemplo de projeto orientado a objetos perfeito. Você pode dizer a si mesmo, “se a linguagem Java faz isso, deve estar correto”. Infelizmente, colocar confiança inquestionável em qualquer implementação OO é perigoso.

Vamos rever a estrutura de dados de pilha clássica. Você pode colocar itens em uma pilha, extrair itens de uma pilha e olhar o primeiro elemento da pilha, sem removê-lo. Talvez você também quisesse ver se a pilha está vazia.

A linguagem Java tem uma classe Stack. A Listagem 5.12 ilustra a interface.

### LISTAGEM 5.12 `java.util.Stack`

```
public class Stack extends {  
    public boolean empty();  
    public Object peek();  
    public Object pop();  
    public Object push( Object item );  
    public int search( Object o );  
}
```

5

Você notará que a linguagem Java otimiza um pouco a definição clássica de pilha. A linguagem Java acrescenta um método `search()`. O método `push()` também retorna o item que você coloca.

Entretanto, há um problema maior. A classe Java Stack também herda de `Vector`. De um ponto de vista, essa é uma decisão inteligente. Herdando de `Vector`, Stack obtém toda a implementação contida em `Vector`. Para implementar Stack, tudo de que você precisa é envolver os métodos de Stack para chamarem internamente os métodos herdados de `Vector` corretos.

Infelizmente, a classe Java Stack é um exemplo de herança pobre. Stack passa no teste ‘é um’? “Um objeto Stack ‘é um’ `Vector`”? Não — o teste falha. `Vector` tem todos os tipos de métodos para colocar elementos em `Vector` e removê-los. Um objeto Stack só permite que você coloque elementos no topo da pilha. A classe `Vector` permite que você insira e remova elementos em qualquer parte.

Aqui, a herança permite que você interaja com a classe Stack de maneiras indefinidas para uma pilha.

## Exposição do problema

Stack passa no teste ‘tem um’. “Um objeto Stack ‘tem um’ objeto Vector”. Para este laboratório, escreva uma nova versão de Stack que empregue o tipo correto de reutilização de implementação.



A próxima seção discute as soluções do Laboratório 4. Não prossiga até concluir o Laboratório 4.

## Soluções e discussão

A Listagem 5.13 ilustra uma possível implementação de Stack

---

### LISTAGEM 5.13 A New Stack Implementation

---

```
public class Stack {  
    private java.util.ArrayList list;  
  
    public Stack(){  
        list = new java.util.ArrayList();  
    }  
  
    public boolean empty() {  
        return list.isEmpty();  
    }  
  
    public Object peek() {  
        if( !empty() ) {  
            return list.get( 0 );  
        }  
        return null;  
    }  
  
    public Object pop() {  
        if( !empty() ) {  
            return list.remove( 0 );  
        }  
        return null;  
    }  
  
    public Object push( Object item ) {  
        list.add( 0,item );  
        return item;  
    }  
}
```

**LISTAGEM 5.13 A New Stack Implementation (*continuação*)**

```
public int search( Object o ) {  
    int index = list.indexOf( o );  
    if( index != -1 ) {  
        return index + 1;  
    }  
    return -1;  
}
```

Se este laboratório nos ensinar algo, é que não devemos depositar toda nossa fé em qualquer fonte de OO. Nada é perfeito.

## Resumo

Hoje, você completou quatro laboratórios. O Laboratório 1 permitiu que você experimentasse a herança simples. Após concluir o Laboratório 1, você deve ter entendido o mecanismo básico da herança. O Laboratório 2 permitiu uma maior exploração da herança, através da classe abstrata e da herança planejada. O Laboratório 3 deve ter solidificado as lições do Dia 4. Os laboratórios 1 e 2 permitiram ver métodos e atributos redefinidos, novos e recursivos. Você também viu como, mesmo se sobrepuiser um método, ainda pode usar a implementação da progenitora.

O Laboratório 4 ilustrou a importância de considerar os testes ‘é um’ e ‘tem um’, enquanto se formam hierarquias de herança. Às vezes, a melhor ação é não herdar. Conforme o Dia 4 enfatiza, a composição é freqüentemente a forma mais limpa de reutilização. A herança só faz sentido do ponto de vista relacional ou ‘é um’. Se dois objetos não estão relacionados pelo tipo, então eles não devem herdar. A implementação compartilhada não é motivo suficiente para herdar.

Dentro de um sistema ou aplicativo, você sempre deve planejar a máxima herança possível. Entretanto, quando programa para um aplicativo específico, você está limitado a esse programa. Com o passar do tempo, você trabalhará em muitos programas diferentes. Quando você começar a notar que está programando as mesmas coisas repetidamente, as oportunidades para herança começarão a se apresentar. Você sempre deve estar atento a essas hierarquias de herança descobertas. Você deve refazer seu código, quando descobrir essas novas hierarquias.

5

## Perguntas e respostas

- P** No Laboratório 4, você mostrou como até a linguagem Java comete erros de OO. Quando eu examinar as APIs Java ou outras fontes de exemplos de OO, como poderei ter certeza de que o que estou vendo é ‘boa’ OO?

**R** É difícil dizer o que constitui a ‘boa’ OO e a ‘má’ OO, mesmo após ter muita experiência em OO. A melhor coisa que você pode fazer é aplicar o que aprendeu e nunca tomar um exemplo como garantido. Encare cada exemplo ponderadamente e, se algo não parecer correto, discuta isso com seus colegas: obtenha uma segunda opinião.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. A partir das soluções de laboratório, dê um exemplo de método redefinido, de método recursivo e de novo método.
2. Por quê você declararia uma classe como sendo abstrata?
3. No Laboratório 4, você explorou os relacionamentos ‘é um’ e ‘tem um’. Antes mesmo de aprender a respeito de herança, você viu relacionamentos ‘tem um’. Quais os relacionamentos ‘tem um’ você viu nos laboratórios do Dia 3?
4. Como esses laboratórios preservaram o encapsulamento entre a classe base e as subclasses?
5. A partir das soluções, encontre um exemplo de especialização.
6. Como as soluções do Laboratório 3 e do Laboratório 4 adotam uma estratégia diferente para reutilização de implementação?

## Exercícios

Não há exercícios hoje. Faça seus laboratórios!

# SEMANA 1

DIA **6**

## Polimorfismo: aprendendo a prever o futuro

Até aqui, você aprendeu sobre os dois primeiros pilares da programação orientada a objetos: encapsulamento e herança. Como você sabe, o encapsulamento permite construir componentes de software independentes e a herança permite reutilizar e estender esses componentes. Entretanto, ainda falta algo. O software está sempre mudando. Se os usuários exigem nova funcionalidade, erros aparecem ou o software precisa ser integrado em novos ambientes, a única constante é a mudança. O ciclo de vida do software não termina quando você distribui um produto. Você precisa de software que possa se adaptar às necessidades futuras. Não seria ótimo, se você pudesse escrever software ‘à prova do futuro’?

Um software à prova do futuro se adapta aos requisitos futuros sem alteração. Um software à prova do futuro permite que você faça alterações e adicione novos recursos facilmente. Felizmente, a POO entende que um software de sucesso não é estático. Assim, a POO usa o conceito de polimorfismo para permitir que você escreva esse software à prova do futuro.

Você passará os próximos dois dias considerando o polimorfismo, o terceiro e último pilar da programação orientada a objetos.

Hoje você aprenderá:

- O que é polimorfismo
- Quais são os diferentes tipos de polimorfismo e o que eles oferecem para seus objetos
- Dicas valiosas para o polimorfismo eficaz

- A respeito de algumas armadilhas do polimorfismo
- Como o polimorfismo atinge os objetivos da OO

## Polimorfismo

Se o encapsulamento e a herança são os socos um e dois da POO, o polimorfismo é o soco para nocaute seguinte. Sem os dois primeiros pilares, você não poderia ter o polimorfismo, e sem o polimorfismo, a POO não seria eficaz. O polimorfismo é onde o paradigma da programação orientada a objetos realmente brilha e seu domínio é absolutamente necessário para a POO eficaz.

*Polimorfismo* significa muitas formas. Em termos de programação, o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático. Assim, um nome pode assumir muitas formas e como pode representar código diferente, o mesmo nome pode representar muitos comportamentos diferentes.

**Novo Termo**

*Polimorfismo*: ter muitas formas. Em termos de programação, muitas formas significa que um único nome pode representar um código diferente, selecionado por algum mecanismo automático. Assim, o polimorfismo permite que um único nome expresse muitos comportamentos diferentes.

De sua própria maneira, o polimorfismo é o distúrbio das múltiplas personalidades do mundo do software, pois um único nome pode expressar muitos comportamentos diferentes.

Toda essa conversa sobre expressar ‘muitos comportamentos diferentes’ pode parecer um pouco abstrata. Pense no termo *abrir*. Você pode abrir uma porta, uma caixa, uma janela e uma conta no banco. A palavra *abrir* pode ser aplicada a muitos objetos diferentes no mundo real. Cada objeto interpreta ‘abrir’ de sua própria maneira. Entretanto, em cada caso, você pode simplesmente dizer ‘abrir’, para descrever a ação.

Nem todas as linguagens suportam polimorfismo. Uma linguagem que suporta polimorfismo é uma *linguagem polimórfica*. Em contraste, uma *linguagem monomórfica* não suporta polimorfismo e, em vez disso, restringe tudo a um e apenas um comportamento estático, pois cada nome é estaticamente vinculado ao seu código.

A herança fornece o aparato necessário para tornar certos tipos de polimorfismo possíveis. No Dia 4, “Herança: obtendo algo para nada”, você viu como a herança permite formar relacionamentos com capacidade de substituição. A capacidade de conexão é extremamente importante para o polimorfismo, pois ela permite que você trate um tipo específico de objeto genericamente.

Considere as classes a seguir:

```
public class PersonalityObject {  
    public String speak() {  
        return "I am an object.";  
    }  
}
```

```
public class PessimisticObject extends PersonalityObject {  
    public String speak() {  
        return "The glass is half empty.";  
    }  
}  
  
public class OptimisticObject extends PersonalityObject {  
    public String speak() {  
        return "The glass is half full.";  
    }  
}  
  
public class IntrovertedObject extends PersonalityObject {  
    public String speak() {  
        return "hi...";  
    }  
}  
  
public class ExtrovertedObject extends PersonalityObject {  
    public String speak() {  
        return "Hello, blah blah blah, did you know that blah blah blah.";  
    }  
}
```

Essas classes formam uma hierarquia de herança muito simples. A classe base, `PersonalityObject`, declara um método: `speak()`. Cada subclasse redefine `speak()` e retorna sua própria mensagem, baseada em sua personalidade. A hierarquia forma relacionamentos com capacidade de substituição entre os subtipos e seus progenitores.

Considere o método `main()` a seguir:

```
public static void main(String [] args ) {  
    PersonalityObject personality = new PersonalityObject();  
    PessimisticObject pessimistic = new PessimisticObject();  
    OptimisticObject optimistic = new OptimisticObject();  
    IntrovertedObject introverted = new IntrovertedObject();  
    ExtrovertedObject extroverted = new ExtrovertedObject();  
    // a capacidade de substituição permite fazer o seguinte  
    PersonalityObject [] personalities = new PersonalityObject[5];  
    personalities [0] = personality;  
    personalities [1] = pessimistic;  
    personalities [2] = optimistic;  
    personalities [3] = introverted;  
    personalities [4] = extroverted;  
  
    // o polimorfismo faz com que PersonalityObject pareça ter  
    // muitos comportamentos diferentes
```

```

// lembre-se - o polimorfismo é o distúrbio das múltiplas
//personalidades do mundo 00
System.out.println( "PersonalityObject[0] speaks: " +
    personalities[0].speak());
System.out.println( "PersonalityObject[1] speaks: " +
    personalities[1].speak());
System.out.println( "PersonalityObject[2] speaks: " +
    personalities[2].speak());
System.out.println( "PersonalityObject[3] speaks: " +
    personalities[3].speak());
System.out.println( "PersonalityObject[4] speaks: " +
    personalities[4].speak());
}

```

Os primeiros dois terços do método `main()` não apresentam nenhuma novidade. Conforme você viu no Dia 4, a capacidade de substituição permite tratar um objeto genericamente. Entretanto, o trecho a seguir é onde o exemplo se torna interessante:

```

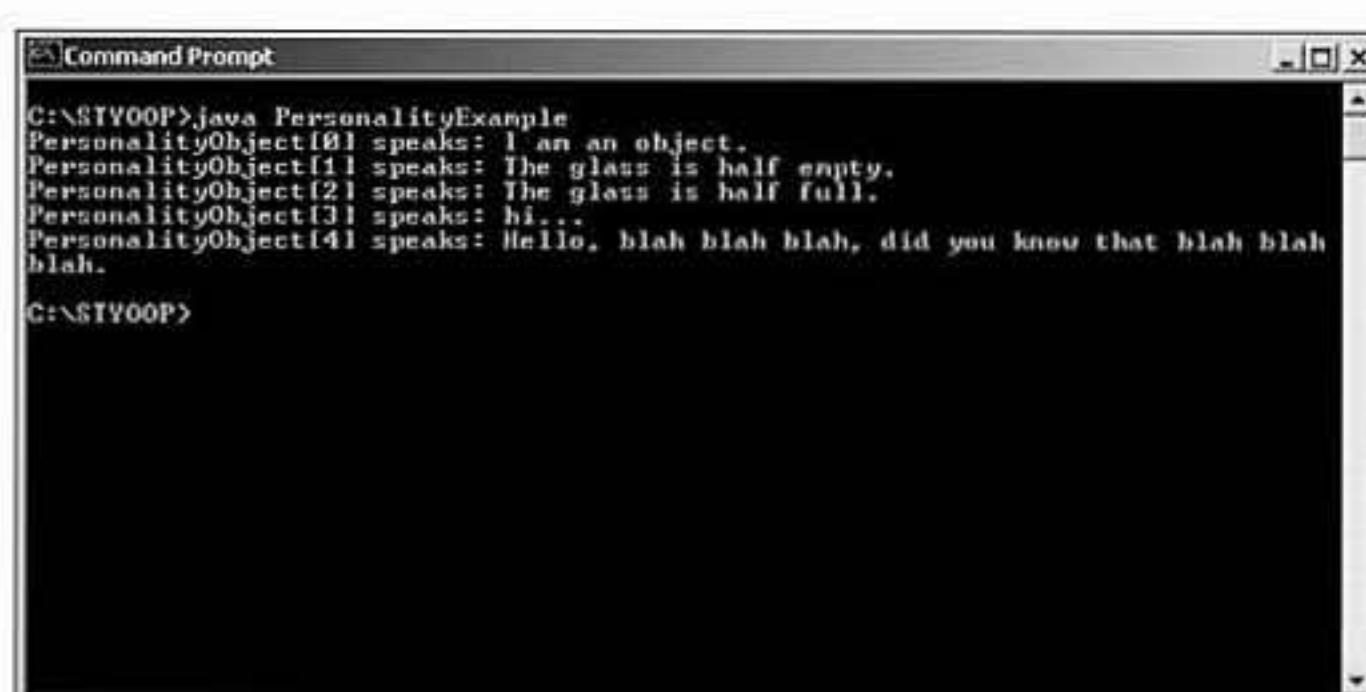
// o polimorfismo faz com que PersonalityObject pareça ter
//muitos comportamentos diferentes
// lembre-se - o polimorfismo é o distúrbio das múltiplas
//personalidades do mundo 00
System.out.println( "PersonalityObject[0] speaks: " +
    personalities[0].speak());
System.out.println( "PersonalityObject[1] speaks: " +
    personalities[1].speak());
System.out.println( "PersonalityObject[2] speaks: " +
    personalities[2].speak());
System.out.println( "PersonalityObject[3] speaks: " +
    personalities[3].speak());
System.out.println( "PersonalityObject[4] speaks: " +
    personalities[4].speak());

```

A Figure 6.1 ilustra a saída.

**FIGURA 6.1**

Demonstração do comportamento polimórfico.



Com base na saída, parece que o método `speak()` de `PersonalityObject` tem muitos comportamentos diferentes. Mesmo que `PersonalityObject` defina `speak()` para imprimir “I am an object”, `PersonalityObject` está exibindo mais de um comportamento. Mesmo que o array supostamente contenha instâncias de `PersonalityObject`, cada membro do array se comporta de forma diferente, quando o método principal chama o método `speak()`. Esse é o dilema do comportamento polimórfico; `PersonalityObject`, o *nome*, parece ter muitos comportamentos.

**Novo TERMO**

`personalities` é um exemplo de variável polimórfica. Uma *variável polimórfica* é uma variável que pode conter muitos tipos diferentes.

**NOTA**

Em uma linguagem tipada, as variáveis polimórficas estão restritas a conter valores específicos. Em uma linguagem dinamicamente tipada, uma variável polimórfica pode conter qualquer valor.

O exemplo anterior explica o mecanismo, mas ele poderia não representar adequadamente o espirito do polimorfismo. Afinal, você sabe exatamente o que o array contém.

Em vez disso, imagine que você tenha um objeto cujo método recebe um objeto `PersonalityObject` como parâmetro:

```
public void makeSpeak( PersonalityObject obj ) {  
    System.out.println( obj.speak() );  
}
```

Os relacionamentos com capacidade de substituição permitem que você passe uma instância do objeto `PersonalityObject` ou qualquer descendente dessa classe para o método `makeSpeak()`, como um argumento. Assim, ao criar descendentes especializados de `PersonalityObject`, como `ExtrovertedObject`, você não precisa mudar a lógica do método para que ele use instâncias das novas classes como argumento. Em vez disso, você pode simplesmente instanciar `ExtrovertedObject` (ou qualquer descendente) e passar a instância para o método.

O polimorfismo entra em ação quando o método `makeSpeak()` é chamado e lhe é passado um objeto como argumento. O polimorfismo garante que o método correto seja chamado através do argumento de `PersonalityObject`, chamando o método do objeto com base no tipo da classe real do argumento, em vez do tipo da classe que o método `makeSpeak()` pensa que está usando. Assim, se você passar um objeto `ExtrovertedObject`, o polimorfismo garantirá que a definição de `speak()` de `ExtrovertedObject` seja chamada e não aquela encontrada na classe base. Como resultado, `makeSpeak()` apresentará mensagens diferentes na tela, dependendo do tipo de argumento que for passado.

6

Você pode aprimorar o polimorfismo para adicionar nova funcionalidade em seu sistema, a qualquer momento. Você pode adicionar novas classes, que tenham funcionalidades jamais imaginadas quando o programa foi escrito pela primeira vez — tudo isso sem ter de mudar seu código já existente. É para isso que serve o software à prova do futuro.

Este exemplo é apenas a ponta do iceberg polimórfico. Na verdade, o exemplo representa apenas uma das muitas formas do polimorfismo. O próprio polimorfismo correto é polimórfico!

Infelizmente, ainda há pouco consenso na comunidade de OO quando se trata de polimorfismo. Em vez de entrar na controvérsia, este livro apresentará quatro formas de polimorfismo. O entendimento dessas quatro formas comuns deve dar-lhe a base de que você precisa para começar a aplicar o polimorfismo. Hoje você aprenderá sobre:

1. Polimorfismo de inclusão
2. Polimorfismo paramétrico
3. Sobreposição
4. Sobrecarga

## Polimorfismo de inclusão

O polimorfismo de inclusão, às vezes chamado de polimorfismo puro, permite que você trate objetos relacionados genericamente. Você viu o polimorfismo de inclusão pela primeira vez, no início do dia.

Considere os métodos a seguir:

```
public void makeSpeak( PessimisticObject obj ) {  
    System.out.println( obj.speak() );  
}  
  
public void makeSpeak( OptimisticObject obj ) {  
    System.out.println( obj.speak() );  
}  
  
public void makeSpeak( IntrovertedObject obj ) {  
    System.out.println( obj.speak() );  
}  
  
public void makeSpeak( ExtrovertedObject obj ) {  
    System.out.println( obj.speak() );  
}
```

`PessimisticObject`, `OptimisticObject`, `IntrovertedObject` e `ExtrovertedObject` estão todos relacionados, pois todos eles são objetos `PersonalityObject`. A capacidade de substituição e o polimorfismo de inclusão permitem que você escreva um método para manipular todos os tipos de objetos `PersonalityObject`:

```
public void makeSpeak( PersonalityObject obj ) {  
    System.out.println( obj.speak() );  
}
```

A capacidade de substituição permite que você passe qualquer objeto `PersonalityObject` para o método e o polimorfismo garante que o método correto seja chamado na instância. O polimorfismo chamará o método com base no tipo verdadeiro da instância (`OptimisticObject`, `IntrovertedObject`, `ExtrovertedObject` ou `PessimisticObject`) e não em seu tipo aparente (`PersonalityObject`).

O polimorfismo de inclusão é útil porque diminui a quantidade de código que precisa ser escrito. Em vez de ter de escrever um método para cada tipo concreto de `PersonalityObject`, você pode simplesmente escrever um método que manipule todos os tipos. O polimorfismo de inclusão e a capacidade de substituição permitem que `makeSpeak()` funcione em qualquer objeto que ‘seja um’ `PersonalityObject`.

O polimorfismo de inclusão torna mais fácil adicionar novos subtipos em seu programa, pois você não precisará adicionar um método especificamente para esse novo tipo. Você pode simplesmente reutilizar `makeSpeak()`.

O polimorfismo de inclusão também é interessante porque faz parecer que as instâncias de `PersonalityObject` exibem muitos comportamentos diferentes. A mensagem apresentada por `makeSpeak()` será diferente de acordo com a entrada do método. Através do uso cuidadoso do polimorfismo de inclusão, você pode mudar o comportamento de seu sistema, introduzindo novas subclasses. A melhor parte é que você pode obter esse novo comportamento sem ter de alterar nenhum código já existente.

O polimorfismo é o motivo pelo qual você não deve associar automaticamente herança com reutilização de implementação. Em vez disso, você deve usar herança principalmente para permitir um comportamento polimórfico através de relacionamentos com capacidade de substituição. Se você definir corretamente os relacionamentos com capacidade de substituição, a reutilização será automática. O polimorfismo de inclusão permite que você reutilize a classe base, qualquer descendente, assim como os métodos que usam a classe base.

Agora, você provavelmente já entende o mecanismo, mas por que desejaria usar polimorfismo de inclusão?

Considere a hierarquia a seguir:

```
public abstract class BaseLog {  
  
    // algumas constantes úteis; não se preocupe com a sintaxe  
    private final static String DEBUG    = "DEBUG";  
    private final static String INFO     = "INFO";  
    private final static String WARNING  = "WARNING";  
    private final static String ERROR    = "ERROR";  
    private final static String FATAL    = "FATAL";  
  
    java.text.DateFormat df = java.text.DateFormat.getTimeInstance();
```

```
public void debug( String message ) {
    log( message, DEBUG, getDate() );
}
public void info( String message ) {
    log( message, INFO, getDate() );
}
public void warning( String message ) {
    log( message, WARNING, getDate() );
}
public void error( String message ) {
    log( message, ERROR, getDate() );
}
public void fatal( String message ) {
    log( message, FATAL, getDate() );
}

// cria uma indicação de tempo
protected String getDate() {
    java.util.Date date = new java.util.Date();
    return df.format( date );
}

// permite que as subclasses definam como e onde vão gravar o log
protected abstract void log( String message, String level, String time );
}
```

BaseLog é um log abstrato que define a interface pública de um log, assim como alguma implementação. BaseLog é abstrato porque cada desenvolvedor precisa personalizar o modo como o log é gravado. Todo desenvolvedor de BaseLog deve definir o método log().

Tornando a classe abstrata, você garante que todo desenvolvedor implemente as subclasses corretamente. Tal estratégia permite que você reutilize o projeto do log entre muitos aplicativos diferentes. Quando um novo aplicativo aparecer, você poderá simplesmente fornecer a implementação necessária para esse aplicativo. Não há necessidade de criar um novo projeto de log. Em vez disso, você pode reutilizar o projeto de log delineado em BaseLog, fornecendo implementações personalizadas.

```
public class FileLog extends BaseLog {

    private java.io.PrintWriter pw;

    public FileLog( String filename )throws java.io.IOException {
        pw = new java.io.PrintWriter( new java.io.FileWriter( filename ) );
    }

    protected void log( String message, String level, String time ) {
        pw.println( level + ":" + time + ":" + message );
    }
}
```

```
    pw.flush();
}

public void close() {
    pw.close();
}

public class ScreenLog extends BaseLog {
    protected void log( String message, String level, String time ) {
        System.out.println( level + ":" + time + ":" + message );
    }
}
```

FileLog e ScreenLog herdam de BaseLog e implementam o método log(). FileLog grava em um arquivo, enquanto ScreenLog escreve na tela.

Lembrando do exemplo Employee do Dia 4, é razoável suspeitar que exista uma classe que saiba como recuperar objetos Employee a partir de um banco de dados:

```
public class EmployeeDatabaseAccessor {
    private BaseLog error_log;

    public EmployeeDatabaseAccessor( BaseLog log ) throws InitDBException {
        error_log = log;
        try {
            // a conexão com o banco de dados
        } catch( DBException ex ) {
            error_log.fatal( "cannot access database: " +
                ex.getMessage() );
        throw new InitDBException( ex.getMessage() );
    }

    public Employee retrieveEmployee( String first_name, String last_name )
        throws EmployeeNotFoundException {
        try {
            // tenta recuperar o funcionário
            return null;
        } catch( EmployeeNotFoundException ex ) {
            error_log.warning( "cannot locate employee: " + last_name +
                "," + first_name );
            throw new EmployeeNotFoundException( last_name, first_name );
        }
    }

    // etc., cada método usa error_log para registrar erros
}
```

A classe EmployeeDatabaseAccessor recebe um objeto BaseLog como argumento em seu construtor. Uma instância usará esse log para gravar todo e qualquer evento importante. Considere o método main() a seguir:

```
public static void main(String [] args ) {  
    BaseLog log = new ScreenLog();  
  
    EmployeeDatabaseAccessor eda = new EmployeeDatabaseAccessor( log );  
  
    Employee emp = eda.retrieveEmployee( "Employee", "Mr." )  
}
```

Concebivelmente, o método main() poderia passar *qualquer* subclasse de BaseLog para EmployeeDatabaseAccessor. Um aplicativo poderia fazer o mesmo. EmployeeDatabaseAccessor é à prova de futuro — no que diz respeito ao registro. Talvez no futuro você precise de um arquivo de log que funcione a cada 24 horas ou um que crie um nome de arquivo usando a data. Talvez outro log faça registros em um manipulador de erros, que receba informações da rede. Quem pode ter certeza? Entretanto, com o polimorfismo de inclusão, você está pronto.

Sem polimorfismo de inclusão, você precisaria de um construtor para cada tipo de log que quisesse fazer o accessor usar. Entretanto, isso não pára aí. Você também precisaria trocar código dentro do accessor, para que ele soubesse qual log deveria usar. Um objeto EmployeeDatabaseAccessor que não usasse polimorfismo, mas quisesse suportar muitos logs diferentes, poderia ser como segue:

```
public class EmployeeDatabaseAccessor {  
    private FileLog    file_log;  
    private ScreenLog screen_log;  
    private int        log_type;  
  
    // algumas constantes 'úteis'  
    private final static int FILE_LOG    = 0;  
    private final static int SCREEN_LOG = 1;  
  
    public EmployeeDatabaseAccessor( FileLog log ) throws InitDBException {  
        file_log = log;  
        log_type = FILE_LOG;  
        init();  
    }  
  
    public EmployeeDatabaseAccessor( ScreenLog log ) throws InitDBException {  
        screen_log = log;  
        log_type = SCREEN_LOG;  
        init();  
    }  
  
    public Employee retrieveEmployee( String first_name, String last_name )
```

```
throws EmployeeNotFoundException {  
    try {  
        // tenta recuperar o funcionário  
        return null;  
    } catch( EmployeeNotFoundException ex ) {  
        if( log_type == FILE_LOG ) {  
            file_log.warning( "cannot locate employee: " +  
                last_name + ", " + first_name );  
        } else if ( log_type == SCREEN_LOG ) {  
            screen_log.warning( "cannot locate employee: " +  
                last_name + ", " + first_name );  
        }  
        throw new EmployeeNotFoundException( last_name, first_name );  
    }  
}  
  
private void init() throws InitDBException {  
    try {  
        // inicializa a conexão com o banco de dados  
    } catch( DBException ex ) {  
        if( log_type == FILE_LOG ) {  
            file_log.fatal( "cannot access database: " +  
                ex.getMessage() );  
        } else if ( log_type == SCREEN_LOG ){  
            screen_log.fatal( "cannot access database: " +  
                ex.getMessage() );  
        }  
        throw new InitDBException( ex.getMessage() );  
    }  
}  
// etc. Cada método usa error_log para registrar erros  
}
```

Você precisará atualizar `EmployeeDatabaseAccessor` sempre que quiser adicionar suporte a um novo log. Agora, qual versão você gostaria de manter?

6

## Polimorfismo paramétrico

O polimorfismo paramétrico permite que você crie métodos e tipos genéricos. Assim como o polimorfismo de inclusão, os métodos e tipos genéricos permitem que você codifique algo uma vez e faça isso trabalhar com muitos tipos diferentes de argumentos.

### Métodos paramétricos

Embora o polimorfismo de inclusão afete o modo como você vê um objeto, o polimorfismo paramétrico afeta os métodos. O polimorfismo paramétrico permite que você programe métodos

genéricos retirando a referência de declarações de tipo de parâmetro até o momento da execução. Considere o método a seguir:

```
int add(int a, int b)
```

`add()` recebe dois inteiros e retorna a soma. Esse método é muito explícito; ele recebe dois inteiros como argumentos. Você não pode passar dois números reais para esse método ou dois objetos matriz. Se você tentar, obterá um erro em tempo de compilação.

Se você quiser somar dois números reais ou duas matrizes, então deve criar métodos para cada tipo:

```
Matrix add_matrix(Matrix a, Matrix b)  
Real add_real(Real a, Real b)
```

etc., para cada tipo que você queira somar.

Seria conveniente se você pudesse evitar ter de escrever muitos métodos. Primeiro, ter de escrever muitos métodos torna seus programas maiores. Você precisará separar um método para cada tipo. Segundo, mais código leva a mais erros e a mais manutenção. Você não quer tornar a manutenção mais difícil do que precisa ser. Terceiro, ter de escrever métodos separados não modela `add()` naturalmente. É mais natural pensar apenas em termos de `add()` e não de `add_matrix()` e `add_real()`.

O polimorfismo de inclusão apresenta uma solução para o problema. Você poderia declarar um tipo chamado `addable`, que teria um método que soubesse como somar-se a outra instância de `addable`.

O tipo poderia ser como segue:

```
public abstract class Addable {  
    public Addable add(Addable);  
}
```

O novo método seria como segue:

```
Addable add_addable(Addable a, Addable b)  
    Return a.add(b)
```



Às vezes, o exemplo anterior é referido como *polimorfismo de função*.

Está tudo bom, tudo bem. Você só precisa escrever um método para somar, entretanto o método funciona apenas para argumentos de `Addable`. Você também precisa certificar-se de que os objetos `Addable` passados para o método sejam do mesmo tipo. Tal requisito é propenso a erros e contrário ao que a interface implica. De qualquer modo, você realmente não resolveu o problema

original. Você ainda precisará escrever métodos para cada tipo que queira somar, que não seja do tipo `Addable`. Nem tudo que você desejará somar será `Addable`.

É aí que o polimorfismo paramétrico entra em ação. O polimorfismo paramétrico permite que você escreva um e apenas um método para somar todos os tipos. O polimorfismo paramétrico retarda a declaração dos tipos dos argumentos.

Considere o método reescrito para tirar proveito do polimorfismo paramétrico:

```
add([T] a, [T] b) : [T]
```

[T] é um argumento exatamente igual a a e b. O argumento [T] especifica o tipo de argumento de a e b. Declarando um método dessa maneira, você adia a definição do tipo dos argumentos até o momento da execução. Você também notará que a e b devem ter o mesmo [T].

Internamente, o método pode ser como segue:

```
[T] add([T] a, [T] b)  
return a + b;
```

O polimorfismo não é mágico. Ele ainda espera que o argumento tenha determinada estrutura. Neste caso, qualquer argumento que você passe deverá definir + para esse tipo.



*Determinada estrutura pode ser a presença de certo método ou operador corretamente definido.*

## Tipos paramétricos

Levado à sua conclusão extrema, o polimorfismo paramétrico pode estender seu alcance aos próprios tipos. Assim como os métodos podem ter parâmetros paramétricos, os tipos podem ser eles próprios paramétricos. Considere o TAD Queue definido no Dia 2:

```
Queue [T]  
void enqueue([T])  
[T] dequeue()  
boolean isEmpty()  
[T] peek()
```

6

O Queue é um tipo parametrizado. Em vez de escrever uma classe de fila para cada tipo que gostaria de enfileirar, você simplesmente especifica os tipos de elementos que gostaria de enfileirar, para conter dinamicamente em tempo de execução. Originalmente, você poderia dizer que o objeto Queue era um Queue de Object. Agora, o objeto Queue pode ser um Queue de qualquer tipo.

Assim, se você quisesse armazenar objetos Employee, faria a seguinte declaração:

```
Queue[Employee] employee_queue = new Queue[Employee];
```

Agora, quando você usar Queue, só poderá usar enqueue() e dequeue() nas instâncias de funcionário.

Se tipos paramétricos não forem possíveis, você precisaria escrever uma fila separada para inteiros, outra para reais e ainda outra para alienígenas do espaço.

Em vez disso, usando tipos parametrizados, você pode escrever o tipo uma vez, neste caso uma fila, e usá-lo para conter todos os tipos possíveis.



### NOTA

Polimorfismo paramétrico soa bem no papel, mas há um problema: suporte.

Para aqueles que estão familiarizados com Java, os exemplos anteriores podem parecer estranhos. Na versão 1.3, a linguagem Java não tem suporte nativo a tipos parametrizados ou polimorfismo paramétrico em geral. Você pode imitar tipos parametrizados, mas o preço na eficiência é bastante alto. Existem algumas extensões Java disponíveis para suporte a polimorfismo paramétrico, entretanto nenhuma delas foi oficialmente sancionada pela Sun.

A sintaxe dos exemplos anteriores é completamente inventada. Entretanto, ela demonstra as idéias adequadamente.

## Sobreposição

A sobreposição é um tipo importante de polimorfismo. Você viu como cada subclasse de `PersonalityObject` sobrepôs o método `speak()` no início deste dia. Entretanto, você viu detalhadamente um exemplo ainda mais interessante de sobreposição e polimorfismo no Dia 5. Especificamente, considere as definições de classe `MoodyObject` e `HappyObject`:

```
public class MoodyObject {  
  
    // retorna o humor  
    protected String getMood() {  
        return "moody";  
    }  
  
    // pergunta ao objeto como ele se sente  
    public void queryMood() {  
        System.out.println("I feel " + getMood() + " today!");  
    }  
}  
  
public class HappyObject extends MoodyObject {  
  
    // redefine o humor da classe  
    protected String getMood() {  
        return "happy";  
    }  
  
    // especialização  
    public void laugh() {
```

```
        System.out.println("hehehe... hahaha... HAHAHAHAHAHA!!!!");  
    }  
}
```

Aqui, você vê que HappyObject sobrepõe o método `getMood()` de MoodyObject. O interessante é que a definição de MoodyObject de `queryMood()` faz uma chamada a `getMood()` internamente.

Você notará que HappyObject não sobrepõe o método `queryMood()`. Em vez disso, HappyObject simplesmente herda o método como um método recursivo de MoodyObject. Quando você chama `queryMood()` em HappyObject, o polimorfismo da instância garante a chamada da versão sobrescrita de `getMood()` em HappyObject, internamente.

Aqui, o polimorfismo cuida dos detalhes do método a ser chamado. Isso o libera de ter que redefinir `queryMood()`, para que ele chame a versão correta de `getMood()`.

Posteriormente, você viu como poderia tornar `getMood()` abstrato na progenitora:

```
public abstract class MoodyObject {  
  
    // retorna o humor  
    protected abstract String getMood();  
  
    // pergunta ao objeto como ele se sente  
    public void queryMood() {  
        System.out.println("I feel " + getMood() + " today!");  
    }  
}
```

Os métodos abstratos são freqüentemente referidos como *métodos adiados*, pois você retarda a definição para as classes descendentes. Entretanto, assim como em qualquer outro método, a classe que define o método abstrato pode fazer chamadas ao método. Assim como os métodos sobrepostos, o polimorfismo garantirá que a versão correta do método adiado seja sempre chamada nas subclasses.

## Sobrecarga

A sobrecarga, também conhecida como *polimorfismo ad-hoc*, permite que você use o mesmo nome de método para muitos métodos diferentes. Cada método difere apenas no número e no tipo de seus parâmetros.

Considere os métodos a seguir, definidos em `java.lang.Math`:

```
public static int max(int a, int b);  
public static long max(long a, long b);  
public static float max(float a, float b);  
public static double max(double a, double b);
```

Os métodos `max()` são todos exemplos de sobrecarga. Você notará que os métodos `max()` diferem apenas no tipo de parâmetros.

A sobrecarga é útil quando um método não é definido por seus argumentos. Em vez disso, o método é um conceito independente de seus argumentos. O método transcende seus parâmetros específicos e se aplica a muitos tipos diferentes de parâmetros. Peguemos o método `max().max()` é um conceito genérico que recebe dois parâmetros e informa qual é maior. Essa definição não muda se você comparar inteiros, números com ponto flutuante, valores duplos ou a ordem da bicada de um bando de pássaros.

A operação `+` é outro exemplo de método sobrecarregado. O conceito `+` é independente de seus argumentos. Você pode somar todos os tipos de elementos.



**NOTA**  
Você não pode sobrecarregar ou sobrepor operadores na linguagem Java; entretanto, a linguagem Java tem alguma sobrecarga interna.

Se a sobrecarga não fosse possível, você teria que fazer o seguinte:

```
public static int max_int( int a, int b );
public static long max_long( long a, long b );
public static float max_float( float a, float b );
public static double max_double( double a, double b );
public static bird max_bird( bird a, bird b );
```

Sem a sobrecarga, você deve dar a cada método um nome exclusivo. Os métodos `max()` não transcederiam mais seus parâmetros. `Max` deixaria de ser um conceito abstrato. Em vez disso, você teria de definir o método em termos de seus argumentos. Ter de escrever o método `max()` dessa maneira não é um jeito natural de modelar o conceito de `max()`. Isso também fornece ao programador mais detalhes para ter em mente.

É claro que chamar cada método com um nome diferente não é polimórfico. Quando todos os métodos compartilham o mesmo nome, você obtém comportamento polimórfico, pois diferentes métodos são chamados internamente, dependendo dos tipos de parâmetros passados. Você pode simplesmente chamar `max()` e passar seus parâmetros. O polimorfismo cuidará de chamar o método correto internamente.

O modo como o polimorfismo direciona a chamada de método depende da linguagem. Algumas linguagens solucionam a chamada de método durante a compilação, enquanto outras vinculam a chamada de método dinamicamente, em tempo de execução.

## Conversão

Conversão e sobrecarga freqüentemente andam lado a lado. A conversão também pode fazer com que um método pareça como se fosse polimórfico. A conversão ocorre quando um argumento de um tipo é convertido para o tipo esperado, internamente.

Considere a definição a seguir:

```
public float add( float a, float b );  
add() recebe dois argumentos float e os soma.
```

O segmento de código a seguir cria algumas variáveis inteiras e chama o método add():

```
int iA = 1;  
int iB = 2;  
  
add(iA,iB);
```

Entretanto, o método add() solicita dois argumentos float. É aí que a conversão entra em ação.

Quando você chama add() com argumentos int, os argumentos são convertidos em valores float pelo compilador. Isso significa que, antes que os argumentos int sejam passados para add(), primeiro eles são convertidos em valores float. Os programadores Java reconhecerão essa conversão.

Assim, a conversão faz o método add() parecer polimórfico, pois o método parece funcionar para valores float e int. Conforme você viu na última seção, também seria possível ter um método add sobrecarregado, da forma:

```
public int add(int a, int b);
```

Nesse caso, add(iA,iB) não resultaria em conversão. Em vez disso, o método add() corretamente sobrecarregado seria chamado.

## Polimorfismo eficaz

Assim como todos os outros pilares, o polimorfismo eficaz não acontece por acidente. Existem alguns passos que você pode executar para garantir um polimorfismo eficaz.

O primeiro passo para o polimorfismo eficaz é ter encapsulamento e herança eficientes.

Sem encapsulamento seu código se torna facilmente dependente da implementação de suas classes. Não permita que o encapsulamento seja destruído. Se o código se tornar dependente de alguns aspectos da implementação de uma classe, você não poderá conectar uma subclasse que refaça essa implementação. Um bom encapsulamento é o primeiro passo para o polimorfismo.

**NOTA**

É importante notar que, nesse contexto, *interface* é um pouco diferente da noção de interfaces Java, embora sejam semelhantes. Aqui, o termo *interface* é usado para descrever a lista de mensagens que você pode enviar para um objeto. Todas essas mensagens compreendem a interface pública de um objeto.

Uma interface Java também define as mensagens que você pode enviar para um objeto Java. Quando uma classe Java implementa uma interface, todos os métodos da interface se tornam parte da interface pública global da classe.

Entretanto, a interface Java não é a única maneira de definir as mensagens que você pode enviar para um objeto. Na linguagem Java, qualquer método público definido na definição da classe se tornará parte da interface pública do objeto. Isso significa que, se uma classe implementar uma interface e definir métodos públicos adicionais, os dois conjuntos de métodos se tornarão parte de sua interface pública.

Usar interfaces Java ao programar é considerada uma boa prática, pois isso separa a definição da interface da implementação da classe dessa interface. Quando você separa as duas, muitas classes de outro modo não relacionadas poderiam implementar a mesma interface. Assim como a herança, os objetos que compartilham uma interface comum também podem tomar parte em relacionamentos com capacidade de substituição, mas sem ter de fazer parte da mesma hierarquia de herança.

A herança é um fator importante no polimorfismo de inclusão. Sempre tente estabelecer relacionamentos com capacidade de substituição programando o mais próximo possível da classe base. Essa prática permitirá que mais tipos de objetos participem de seu programa.

Um modo de estimular a capacidade de substituição é através de hierarquias bem pensadas. Mova as características comuns para classes abstratas e programe seus objetos para usar a classe abstrata e não uma descendente concreta específica. Desse modo, você poderá introduzir qualquer descendente em seu programa.

**DICA**

Para obter um polimorfismo eficaz, siga estas dicas:

- Siga as dicas do encapsulamento e da herança eficazes.
- Sempre programe para a interface e não para a implementação. Programando para uma interface, você define especificamente quais tipos de objetos podem participar de seu programa. Então, o polimorfismo garantirá que esses objetos participem corretamente.
- Pense e programe genericamente. Deixe o polimorfismo se preocupar com os detalhes específicos. Se você deixar o polimorfismo fazer seu trabalho, não precisará escrever muito código. Ele cuidará dos detalhes para você!
- Defina a base do polimorfismo estabelecendo e usando relacionamentos com capacidade de substituição. A capacidade de substituição e o polimorfismo garantirão que você possa adicionar novos subtipos em seu programa e que o código correto será executado, quando esses subtipos forem usados.

- Se sua linguagem fornece uma maneira de separar completamente a interface e a implementação, favoreça esse mecanismo em detrimento da herança. Um exemplo de mecanismo que permite definir e herdar interface sem implementação é a Java Interface. Separar os dois permite uma capacidade de substituição mais flexível, obtendo-se assim mais oportunidade de polimorfismo.
- Use classes abstratas para separar a interface da implementação. Todas as classes não-folha devem ser abstratas; programe apenas para essas classes abstratas.

A discussão anterior focalizou muito as linguagens fortemente tipadas, como Java. Em uma linguagem fortemente tipada, você deve declarar explicitamente o tipo de uma variável. Entretanto, algumas linguagens orientadas a objetos, como Smalltalk, não têm esse requisito. Em vez disso, tais linguagens são dinamicamente tipadas. Tipagem dinâmica significa que você não precisa indicar explicitamente o tipo de uma variável, ao criá-la. Em vez disso, o tipo é determinado dinamicamente, em tempo de execução. Assim, basicamente, toda variável é polimórfica.

O polimorfismo é um pouco mais simples em linguagens dinamicamente tipadas. As variáveis são automaticamente polimórficas, pois elas podem conter qualquer valor. Desde que o objeto tenha o método esperado, ele pode trabalhar de forma polimórfica. É claro que tudo será destruído, se você tentar chamar um método que não existe!

As linguagens tipadas são um pouco mais rigorosas. As linguagens dinamicamente tipadas permitem que você trate um objeto de forma polimórfica, desde que ele tenha o método em que você está interessado. O objeto não precisa pertencer a uma hierarquia de herança específica. As linguagens fortemente tipadas exigem que o objeto pertença a uma hierarquia de herança específica.

Entretanto, os dois casos não são tão diferentes assim. O comportamento é que realmente define um tipo; as linguagens tipadas exigem apenas a presença de todos os comportamentos definidos. Assim, os conceitos por trás do polimorfismo em linguagens fortemente tipadas e dinâmicas são os mesmos. Tudo se reduz a um objeto que sabe como executar algum comportamento.

O foco nas linguagens tipadas é deliberado. Focalizar diretamente a forte tipagem o obriga a se concentrar no tipo, sem perder os detalhes. Se você pode entender o polimorfismo em uma linguagem tipada, então certamente pode entendê-lo em uma linguagem não tipada. O inverso parece não ser verdadeiro!

A escolha de focalizar o tipo também é pragmática. A maioria das principais linguagens orientadas a objetos é fortemente tipada.

## Armadilhas polimórficas

Ao se usar polimorfismo, existem três armadilhas principais a serem evitadas.

## Armadilha 1: mover comportamentos para cima na hierarquia

Muito freqüentemente, desenvolvedores inexperientes moverão comportamentos para cima na hierarquia, para aumentar o polimorfismo. O entusiasmo em tratar tudo de maneira polimórfica pode cegar facilmente um desenvolvedor e resultar em hierarquias mal projetadas.

Se você mover um comportamento para cima demais em uma hierarquia, nem todos os descendentes poderão suportar o comportamento. Lembre-se de que os descendentes *nunca* devem retirar funcionalidade de seus ancestrais. Não destrua uma boa herança para tornar seus programas mais polimórficos.

Se você tiver vontade de mover comportamentos para cima na hierarquia, unicamente para melhorar o polimorfismo, *pare*. Você está em território perigoso.

Se você encontrar limitações demais em sua hierarquia, talvez queira revê-la. Mova elementos comuns para classes abstratas; mova funcionalidade por toda parte. Entretanto, não mova métodos para cima na hierarquia, além do nível onde eles são necessários pela primeira vez. Não se habitue a mover comportamentos por capricho, simplesmente para adicionar suporte polimórfico. Certifique-se de que você tenha outro motivo válido para a mudança. Você pode ter sorte algumas vezes, mas a prática o pegará posteriormente, e maus hábitos de programação são difíceis de perder.

Ao desenvolver suas hierarquias, é importante considerar a evolução em potencial das classes, com o passar do tempo. Você pode dividir a hierarquia em níveis funcionais. Com o passar do tempo, você pode evoluir sua hierarquia, adicionando novos níveis funcionais, quando eles forem necessários. Entretanto, você só deve especular com base nos requisitos futuros que conhece. Existe um número infinito de ‘e se’ indefinidos. Planeje apenas as eventualidades que você conhece.

## Armadilha 2: sobrecarga de desempenho

Tudo tem um preço. O verdadeiro polimorfismo sofrerá certa sobrecarga de desempenho. O polimorfismo não pode competir com um método que conhece seus argumentos estaticamente. Em vez disso, com o polimorfismo, devem haver verificações em tempo de execução. Para o polimorfismo de inclusão, a implementação real do objeto para o qual você envia mensagens deve ser determinada em tempo de execução. Todas essas verificações levam tempo para terminar e são mais lentas em comparação aos valores que conhecem seus tipos estaticamente.

As vantagens da manutenção e da flexibilidade do programa devem compensar qualquer perda de desempenho. Entretanto, se você estiver escrevendo um aplicativo em que o tempo seja importante, talvez precise ter cuidado ao usar polimorfismo. Entretanto, mantenha o desempenho em perspectiva. Crie uma implementação OO limpa, trace o perfil da implementação e otimize cuidadosamente o desempenho, onde o traçado do perfil revela problemas.

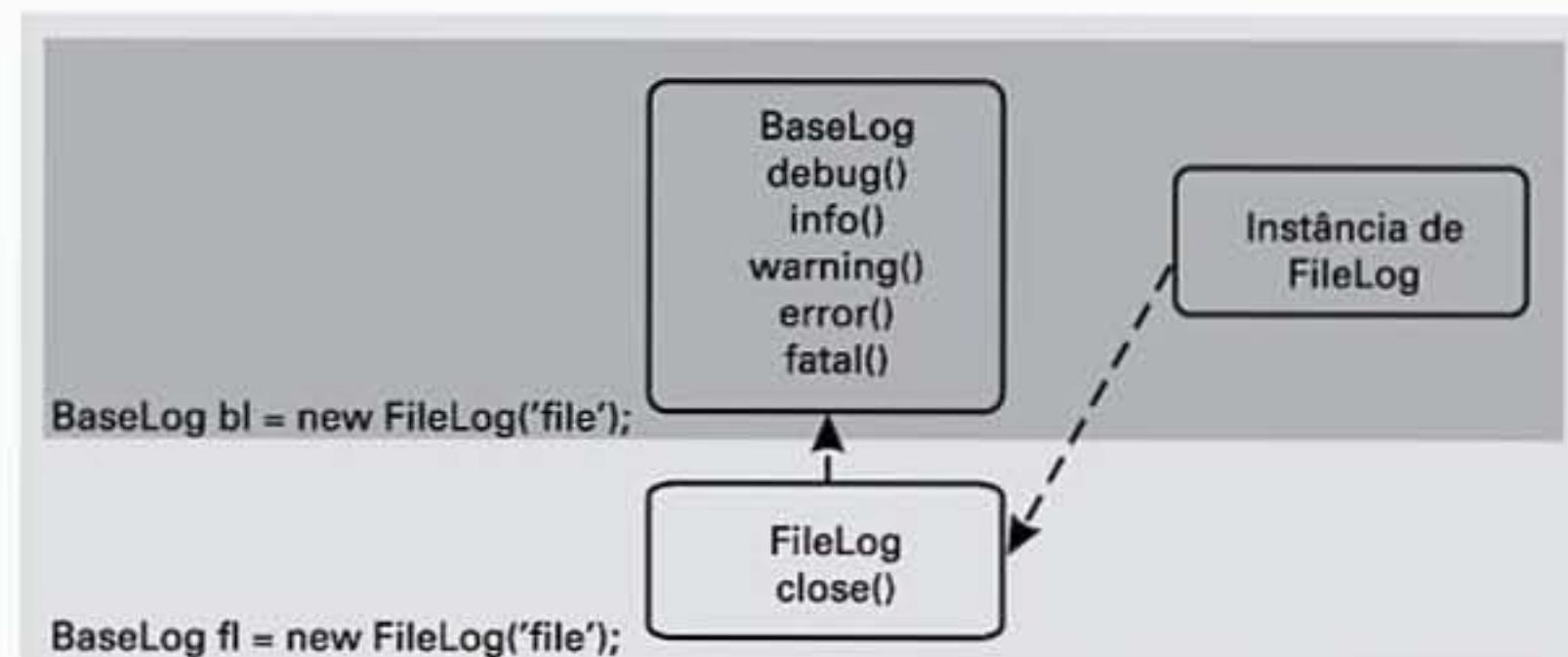
## Armadilha 3: vendas

O polimorfismo de inclusão tem uma deficiência. Embora seja verdade que você pode passar uma subclasse para um método que está esperando uma classe base, o método não pode tirar proveito de todos os novos métodos que a subclasse possa adicionar em sua interface. Por exemplo, a classe `FileLog` adiciona um método `close()` em sua interface. O método `retrieveEmployee()` de `EmployeeDatabaseAccessor` não poderá usá-lo. Um método programado para aceitar a classe base só saberá manipular a interface da classe base.

A Figura 6.2 mostra como a visualização de uma instância é relativa. Optar por ver uma instância de `FileLog` como se fosse uma instância de `BaseLog` é como colocar vendas nos olhos. Você só tem acesso aos métodos declarados em `BaseLog`. É claro que, se você tratar da instância de `FileLog` como uma instância de `FileLog`, obterá toda a funcionalidade definida ou herdada pela classe `FileLog`, como aconteceria normalmente.

**FIGURA 6.2**

*Diferentes visões  
do mesmo objeto.*



Assim, quando você adicionar novos tipos de forma polimórfica, seu código antigo não poderá usar nenhum dos métodos novos. Entretanto, o novo código (ou o código atualizado) está livre para usar qualquer coisa na interface pública.

Novamente, essa armadilha aponta o motivo pelo qual um descendente nunca deve remover comportamentos de seu progenitor. Um método que conte com o polimorfismo de inclusão só saberá como exercer os métodos definidos com o tipo que estiver programado para manipular. Se estiver faltando um comportamento, o método não funcionará.

Essa armadilha também mostra que simplesmente trocar para um novo tipo em seu programa já existente, freqüentemente não é tão fácil quanto poderia parecer. No exemplo do `FileLog`, você precisará encontrar um modo de chamar o método `close()`, quando seu programa tiver terminado com o log.

## Advertências

Existe uma advertência importante a ser lembrada ao se considerar o polimorfismo. Cada linguagem implementa o polimorfismo de uma forma diferente. Esta discussão esboçou as definições teóricas por trás do polimorfismo.

A maioria, se não todas as linguagens, suporta polimorfismo de inclusão em algum grau. Por outro lado, poucas suportam o verdadeiro polimorfismo paramétrico. A linguagem Java certamente não suporta polimorfismo paramétrico. C++ finge implementá-lo.

A maioria das linguagens tem alguma forma de sobrecarga e conversão. Entretanto, a implementação exata vai variar de uma linguagem para outra.

Assim, quando você começar a programar de forma polimórfica, lembre-se da teoria, mas prosiga cuidadosamente. Você não pode evitar as limitações de sua linguagem de implementação.

## Como o polimorfismo atende os objetivos da OO

O polimorfismo atende cada um dos objetivos da POO. O polimorfismo produz software que é:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível
6. Oportuno

O polimorfismo atende esses objetivos das seguintes maneiras:

- Natural: o polimorfismo permite que você modele o mundo mais naturalmente. Em vez de programar para casos especiais, o polimorfismo permite que você trabalhe em um nível mais genérico e conceitual.

A sobrecarga e o polimorfismo paramétrico permitem que você modele um objeto ou método em nível conceitual do que esse objeto ou método faz e não de quais tipos de parâmetros ele poderia processar. O polimorfismo de inclusão permite que você manipule tipos de objetos, em vez de implementações específicas.

Tal programação genérica é mais natural, pois ela o libera para programar no nível conceitual do problema e não nas implementações específicas.

- Confiável: o polimorfismo resulta em código confiável.

Primeiro, polimorfismo simplifica seu código. Em vez de ter de programar casos especiais de cada tipo de objeto que poderia manipular, você escreve simplesmente um caso.

Se você não pudesse programar dessa maneira, teria de atualizar seu código sempre que adicionasse uma nova subclasse. Ter de atualizar código é algo propenso a erros.

Segundo, o polimorfismo permite que você escreva menos código. Quanto menos código você escreve, menores as chances de introdução de erros.

O polimorfismo também permite que você isole partes do código das alterações das subclasse, garantindo que elas tratem apenas com os níveis da hierarquia de herança importantes para sua função.

- Reutilizável: o polimorfismo auxilia a reutilização. Para que um objeto use outro, ele só precisa conhecer a interface do segundo objeto e não os detalhes da implementação. Como resultado, a reutilização pode ocorrer mais prontamente.
- Manutenível: o polimorfismo auxilia a manutenibilidade. Conforme você já viu, o polimorfismo resulta em menos código. Assim, há menos para manter. Quando você precisa manter código, não é obrigado a manter grandes estruturas condicionais.
- Extensível: o código polimórfico é mais extensível. O polimorfismo de inclusão permite que você adicione novos subtipos em seu sistema, sem ter de alterar o sistema para usar o novo subtipo. A sobrecarga permite que você adicione novos métodos, sem ter de se preocupar com conflitos de atribuição de nomes. Finalmente, o polimorfismo paramétrico permite que você estenda automaticamente suas classes, para suportar novos tipos.
- Oportuno: o polimorfismo o ajuda a escrever código oportuno. Se você pode escrever menos código, então pode distribuir seu código mais cedo. Como o polimorfismo o estimula a programar genericamente, você pode adicionar novos tipos quase que instantaneamente em seus programas. Como resultado, a manutenção e a extensão de seus programas ocorre em um ritmo muito mais rápido.

## Resumo

Polimorfismo é ter muitas formas. O polimorfismo é um mecanismo que permite a um único nome representar códigos diferentes. Como um único nome pode representar códigos diferentes, esse nome pode expressar muitos comportamentos diferentes. O polimorfismo permite que você escreva código humorado: código que exibe diferentes comportamentos.

Para os objetivos deste livro, você aprendeu a respeito de quatro diferentes tipos de polimorfismo:

- Polimorfismo de inclusão
- Polimorfismo paramétrico
- Sobreposição
- Sobrecarga

Embora exista certo desacordo em relação ao polimorfismo na comunidade de OO, esses tipos descrevem algumas das formas mais comuns de polimorfismo. Entender esses tipos proporcionará a você uma boa base na teoria de polimorfismo.

O polimorfismo de inclusão permite que um objeto expresse muitos comportamentos diferentes, em tempo de execução. Do mesmo modo, o polimorfismo paramétrico permite que um objeto ou método opere com vários tipos de parâmetro diferentes.

A sobreposição permite que você sobreponha um método e saiba que o polimorfismo garantirá que o método correto sempre será executado.

Finalmente, a sobrecarga permite que você declare o mesmo método várias vezes. Cada declaração difere simplesmente no número e no tipo de argumentos. A conversão faz um método parecer polimórfico, convertendo argumentos nos tipos de argumentos esperados pelo método.

O polimorfismo permite que você escreva código mais curto e mais inteligível, que também é mais flexível para os requisitos futuros.

## Perguntas e respostas

**P Existem três pilares na programação orientada a objetos. Se eu não usar todos os três, meu software não será OO?**

**R** No mínimo, você sempre deve usar encapsulamento. Sem encapsulamento, você realmente não terá herança eficaz, polimorfismo ou OO em geral. Quanto aos outros dois pilares, você só deve usá-los quando fizer sentido. Não use herança ou polimorfismo apenas para que você possa dizer que os utilizou em seu programa.

A ausência de herança e polimorfismo não significa que um programa é necessariamente não-OO. Entretanto, você precisa dar uma boa olhada em seu programa, para ver se está desperdiçando uma oportunidade de usar corretamente os outros pilares.

**P Por que existe tanto desacordo em relação ao polimorfismo na comunidade de OO?**

**R** Há muito desacordo na literatura que ainda precisa ser acertado. Parece que cada autor tem seu próprio vocabulário. Grande parte desse desacordo provém do fato de que cada linguagem implementa o polimorfismo de sua própria maneira. Todas essas diferentes implementações têm dividido a comunidade.

O importante é que você entenda os quatro tipos apresentados hoje. Embora eles possam receber nomes diferentes, esses quatro tipos são bastante reconhecidos.

**P A linguagem Java vai suportar polimorfismo paramétrico?**

**R** Somente a Sun pode responder essa pergunta. Entretanto, atualmente existe um Java Specification Request (JSR-000014), que foi aceito para desenvolvimento. Ele acrescenta tipos genéricos na linguagem de programação Java. Assim, a coisa está a caminho!

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

### Teste

1. Quais são os quatro tipos de polimorfismo?
2. O que o polimorfismo de inclusão permite que você faça?
3. Como a sobrecarga e o polimorfismo paramétrico modelam mais naturalmente o mundo real?
4. Ao programar, por que você deve programar para uma interface, em vez de para uma implementação?
5. Como o polimorfismo e a sobreposição trabalham juntos?
6. Qual é outro nome para sobrecarga?
7. Defina sobrecarga.
8. Defina polimorfismo paramétrico.
9. Quais são as três armadilhas associadas ao polimorfismo?
10. Como o encapsulamento e a herança afetam o polimorfismo de inclusão?

### Exercícios

1. Dê um exemplo real de situação de programação onde você acha que poderia usar polimorfismo de inclusão. Pode ajudar pensar em algo que você tenha programado anteriormente, que pudesse tirar proveito do polimorfismo.
2. Dê um exemplo de conversão. Explique por que se trata de conversão.
3. Examine as APIs Java. Encontre um exemplo de sobrecarga e explique-o. Em seguida, encontre uma hierarquia de classes que você poderia destacar para polimorfismo de inclusão. Identifique a hierarquia e explique como você pode aplicar polimorfismo de inclusão nela.

PÁGINA EM BRANCO

# SEMANA 1

DIA 7

## Polimorfismo: hora de escrever algum código

Ontem, você aprendeu a respeito do polimorfismo. Você deve ter um bom entendimento dos quatro diferentes tipos de polimorfismo. Hoje, você ganhará experiência prática com o polimorfismo, através de vários exercícios de laboratório. No final da lição de hoje, você deverá se sentir à vontade com a teoria apresentada no Dia 6, “Polimorfismo: aprendendo a prever o futuro”.

Hoje você aprenderá:

- Como aplicar as diferentes formas de polimorfismo
- Como escrever software à prova do futuro
- Como o polimorfismo pode ajudá-lo a evitar a troca da lógica

## Laboratório 1: aplicando polimorfismo

No Dia 5, o Laboratório 2 apresentou uma hierarquia de funcionários. A Listagem 7.1 apresenta uma classe base `Employee` ligeiramente alterada.

### LISTAGEM 7.1 Employee.java

---

```
public abstract class Employee {  
  
    private String first_name;  
    private String last_name;
```

**LISTAGEM 7.1** Employee.java (*continuação*)

```
private double wage;

public Employee( String first_name, String last_name, double wage ) {
    this.first_name = first_name;
    this.last_name = last_name;
    this.wage = wage;
}

public double getWage() {
    return wage;
}

public String getFirstName() {
    return first_name;
}

public String getLastName() {
    return last_name;
}

public abstract double calculatePay();

public void printPaycheck() {
    String full_name = last_name + ", " + first_name;
    System.out.println( "Pay: " + full_name + " $" + calculatePay() );
}
```

---

A nova classe Employee tem agora um método abstrato `calculatePay()`. Cada subclasse deve definir sua própria implementação de `calculatePay()`. As listagens 7.2 e 7.3 apresentam duas possíveis subclasses.

**LISTAGEM 7.2** CommissionedEmployee.java

```
public class CommissionedEmployee extends Employee {

    private double commission; // o custo por unidade
    private int     units;      // controla o número de unidades vendidas

    public CommissionedEmployee( String first_name, String last_name,
                                double wage, double commission ) {
        super( first_name, last_name, wage );
        // chama o construtor original para inicializar corretamente
```

**LISTAGEM 7.2** CommissionedEmployee.java (*continuação*)

```
        this.commission = commission;
    }

    public double calculatePay() {
        return getWage() + (commission * units );
    }

    public void addSales( int units ) {
        this.units = this.units + units;
    }

    public int getSales() {
        return units;
    }

    public void resetSales() {
        units = 0;
    }
}
```

**LISTAGEM 7.3** HourlyEmployee.java

```
public class HourlyEmployee extends Employee {

    private int hours; // controla o número de horas trabalhadas

    public HourlyEmployee( String first_name, String last_name,
                           double wage ) {
        super( first_name, last_name, wage );
        // chama o construtor original para inicializar corretamente
    }

    public double calculatePay() {
        return getWage()* hours;
    }

    public void addHours( int hours ) {
        this.hours = this.hours + hours;
    }

    public int getHours() {
        return hours;
    }
}
```

**LISTAGEM 7.3 HourlyEmployee.java (continuação)**

```
}

public void resetHours() {
    hours = 0;
}

}
```

---

Cada subclasse fornece sua própria implementação de `calculatePay()`. `HourlyEmployee` simplesmente calcula o pagamento multiplicando as horas trabalhadas pela taxa horária. Um objeto `CommissionedEmployee` recebe o salário-base, mais um bônus por cada unidade vendida. Cada subclasse também adiciona alguns métodos próprios. Por exemplo, `HourlyEmployee` tem um método para reconfigurar as horas. Do mesmo modo, `CommissionedEmployee` tem um método para adicionar vendas.

Conforme você aprendeu no Dia 4, “Herança: hora de escrever algum código”, `CommissionedEmployee` e `HourlyEmployee` permitem que as instâncias das duas classes compartilhem um relacionamento com capacidade de substituição. Você pode usar uma instância de `CommissionedEmployee` ou uma instância de `HourlyEmployee`, em lugar de `Employee`. Entretanto, o que o polimorfismo permite fazer?

Considere a classe `Payroll`, apresentada na Listagem 7.4.

**LISTAGEM 7.4 Payroll.java**

```
public class Payroll {

    private int    total_hours;
    private int    total_sales;
    private double total_pay;

    public void payEmployees( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps[i];
            total_pay += emp.calculatePay();
            emp.printPaycheck();
        }
    }

    public void recordEmployeeInfo( CommissionedEmployee emp ) {
        total_sales += emp.getSales();
    }

    public void recordEmployeeInfo( HourlyEmployee emp ) {
        total_hours += emp.getHours();
    }
}
```

**LISTAGEM 7.4 Payroll.java (continuação)**

```
}
```

```
public void printReport() {
    System.out.println( "Payroll Report:" );
    System.out.println( "Total Hours: " + total_hours );
    System.out.println( "Total Sales: " + total_sales );
    System.out.println( "Total Paid: $" + total_pay );
}
```

```
}
```



**MÉTODOS GET E SET EM DEMASIA INDICAM UM MAL PROJETO OO.** Em geral, você quer solicitar os dados de um objeto muito raramente. Em vez disso, você deve pedir para que um objeto faça algo com seus dados.

No exemplo de funcionários, teria sido uma OO melhor se passasse para o objeto Employee um objeto Report, onde ele pudesse registrar suas horas, vendas etc. Embora isso fosse uma OO melhor, teria se desviado do exemplo.

A 'boa OO' também é relativa. Se você estiver escrevendo objetos genéricos que serão usados em muitas situações diferentes, talvez queira adicionar métodos get/set, para que possa manter a interface da classe fácil de gerenciar.

Considere o método `payEmployees( Employee [] emps )`. Os relacionamentos com capacidade de substituição permitem que você passe qualquer subclasse de Employee para o método. Geralmente, esse método trata objetos HourlyEmployee e CommissionedEmployee como simples instâncias de Employee.

O polimorfismo é o que torna esse exemplo interessante. Quando os métodos `payEmployees()` dizem

```
total_pay += emp.calculatePay()
```

o polimorfismo faz parecer que Employee tenha muitos comportamentos diferentes. Quando `emp.calculatePay()` é chamado em um objeto que é realmente um HourlyEmployee, `calculatePay()` calcula o pagamento, multiplicando a taxa horária pelo número de horas trabalhadas. Do mesmo modo, quando a instância subjacente é um objeto CommissionedEmployee, `calculatePay()` retorna o salário mais todos os bônus de vendas.

`payEmployees()` é um exemplo de *polimorfismo de inclusão*. Esse método funciona para qualquer funcionário. O método não precisa de código especial, você não precisará atualizá-lo cada vez que adicionar uma nova subclasse em seu sistema — ele simplesmente funciona para todos os objetos Employee.

Métodos como

```
recordEmployeeInfo( CommissionedEmployee emp )
```

e

```
recordEmployeeInfo( HourlyEmployee emp )
```

demonstram a sobrecarga. A *sobrecarga* permite que um método pareça ser polimórfico. Por exemplo, ela permite o seguinte:

```
Payroll payroll = new Payroll();
CommissionedEmployee emp1 = new CommissionedEmployee( "Mr.", "Sales",
                                                       25000.00, 1000.00 );
HourlyEmployee emp2 = new HourlyEmployee( "Mr.", "Minimum Wage", 6.50 );
payroll.recordEmployeeInfo( emp2 );
payroll.recordEmployeeInfo( emp1 );
```

`recordEmployeeInfo()` parece ser polimórfico, pois pode manipular os dois tipos de funcionário.

A sobrecarga é um pouco mais limitada que o polimorfismo de inclusão. Com o polimorfismo de inclusão, você viu que precisava apenas de um método, `payEmployees()`, para calcular o pagamento de qualquer objeto `Employee`. Independentemente de quantas subclasses de `Employee` você introduza, o método sempre funcionará. Esse é o poder do polimorfismo de inclusão.

Os métodos que empregam sobrecarga não são tão robustos assim. Pegue como exemplo o método `recordEmployeeInfo()`. Sempre que você adicionar uma nova classe na hierarquia `Employee`, terá de adicionar um novo método `recordEmployeeInfo()` para o novo tipo. Embora alguns métodos extras possam ser aceitáveis para uma hierarquia pequena, talvez você tenha de refazer sua hierarquia, para que possa escrever um método `recordEmployeeInfo()` genérico, quando o número de subclasses de `Employee` aumentar.

A Listagem 7.5 fornece um pequeno método principal que executa os métodos `Payroll`.

#### **LISTAGEM 7.5** PayrollDriver.java

---

```
public class PayrollDriver {
    public static void main( String [] args ) {

        // cria o sistema de folha de pagamento
        Payroll payroll = new Payroll();

        // cria e atualiza alguns funcionários
        CommissionedEmployee emp1 = new CommissionedEmployee( "Mr.", "Sales",
                                                               25000.00, 1000.00 );
        CommissionedEmployee emp2 = new CommissionedEmployee( "Ms.", "Sales",
                                                               25000.00, 1000.00 );

        emp1.addSales( 7 );
        emp2.addSales( 5 );
```

**LISTAGEM 7.5 PayrollDriver.java (continuação)**

```
HourlyEmployee emp3 = new HourlyEmployee( "Mr.", "Minimum Wage", 6.50 );
HourlyEmployee emp4 = new HourlyEmployee( "Ms.", "Minimum Wage", 6.50 );
emp3.addHours( 40 );
emp4.addHours( 46 );

// usa os métodos sobrecarregados
payroll.recordEmployeeInfo( emp2 );
payroll.recordEmployeeInfo( emp1 );
payroll.recordEmployeeInfo( emp3 );
payroll.recordEmployeeInfo( emp4 );

// coloca os funcionários em um array
Employee [] emps = new Employee[4];
emps[0] = emp1; emps[1] = emp2; emps[2] = emp3; emps[3] = emp4;

payroll.payEmployees( emps );
payroll.printReport();
}

}
```

A Figura 7.1 mostra a saída do método principal.

**FIGURA 7.1**

*A saída correta de PayrollDriver.*



The screenshot shows a Windows Command Prompt window titled 'Command Prompt'. The command 'java PayrollDriver' is entered, followed by the output of the program:

```
C:\STYOOOP>java PayrollDriver
Pay: Sales, Mr. $32000.0
Pay: Sales, Ms. $30000.0
Pay: Minimum Wage, Mr. $260.0
Pay: Minimum Wage, Ms. $299.0
Payroll Report:
Total Hours: 86
Total Sales: 12
Total Paid: $62559.0
C:\STYOOOP>
```

Se você percorrer o código e calcular manualmente o pagamento de cada funcionário, verá que `payEmployees()` paga o valor correto. Do mesmo modo, todas as informações de funcionário são corretamente gravadas.

## Exposição do problema

No Dia 5, você trabalhou com objetos MoodyObject. A Listagem 7.6 apresenta uma classe MoodyObject ligeiramente modificada.

---

### LISTAGEM 7.6 MoodyObject.java

---

```
public abstract class MoodyObject {  
  
    // retorna o humor  
    protected abstract String getMood();  
  
    // pergunta ao objeto como ele se sente  
    public void queryMood() {  
        System.out.println("I feel " + getMood() + " today!!");  
    }  
}
```

---

As listagens 7.7 e 7.8 apresentam duas subclasses: HappyObject e SadObject.

---

### LISTAGEM 7.7 HappyObject.java

---

```
public class HappyObject extends MoodyObject {  
  
    // redefine o humor da classe  
    protected String getMood() {  
        return "happy";  
    }  
  
    // especialização  
    public void laugh() {  
        System.out.println("hehehe... hahaha... HAHAHAHAHAHA!!!!");  
    }  
}
```

---

---

### LISTAGEM 7.8 SadObject.java

---

```
public class SadObject extends MoodyObject {  
  
    // redefine o humor da classe  
    protected String getMood() {  
        return "sad";  
    }  
  
    // especialização
```

**LISTAGEM 7.8** SadObject.java (*continuação*)

```
public void cry() {  
    System.out.println("'wah' 'boo hoo' 'weep' 'sob' 'weep'");  
}  
}
```

Sua tarefa é praticar o polimorfismo. Escreva uma classe PsychiatristObject. PsychiatristObject deve ter três métodos. examine() deve pegar qualquer instância de MoodyObject e perguntar como ela se sente. PsychiatristObject também deve ter um método observe() sobre-carregado. observe() deve chamar os métodos cry() ou laugh() do objeto. A classe PsychiatristObject deve tecer um comentário médico para cada comportamento.

Certifique-se de usar a classe PsychiatristDriver fornecida, para testar sua solução!



A próxima seção discutirá as soluções do Laboratório 1. Não prossiga até concluir o Laboratório 1.

## Soluções e discussão

A Listagem 7.9 apresenta uma possível classe PsychiatristObject.

**LISTAGEM 7.9** PsychiatristObject.java

```
public class PsychiatristObject {  
  
    // usa polimorfismo de inclusão para examinar todos os objetos humor  
    // genericamente  
    public void examine(MoodyObject obj) {  
        System.out.println("Tell me, object, how do you feel today?");  
        obj.queryMood();  
        System.out.println();  
    }  
  
    // usa sobrecarga para observar objetos especificamente,  
    // mas com um método chamado genericamente  
    public void observe(SadObject obj) {  
        obj.cry();  
        System.out.println(  
            "Hmm... very, very interesting. Something makes this object sad.");  
        System.out.println();  
    }  
}
```

**LISTAGEM 7.9 PsychiatristObject.java (continuação)**

```
public void observe( HappyObject obj ) {  
    obj.laugh();  
    System.out.println(  
        "Hmm... very, very interesting. This object seems very happy." );  
    System.out.println();  
}  
}
```

---

examine como (`MoodyObject obj`) trata todos os objetos `MoodyObject` genericamente. O objeto `PsychiatristObject` pergunta ao objeto `MoodyObject` como ele se sente e chama seu método `queryMood()`. O objeto `PsychiatristObject` precisa de um método `observe()` para cada tipo de objeto `MoodyObject` que gostaria de observar.

Após concluir este laboratório, você deverá começar a se sentir à vontade com os mecanismos básicos do polimorfismo.

## Laboratório 2: conta de banco — aplicando polimorfismo em um exemplo conhecido

No Laboratório 2, você vai pegar o que aprendeu no Laboratório 1 e aplicar em um problema ligeiramente mais complicado. Este laboratório focaliza a hierarquia `BankAccount`, apresentada no Dia 5. A hierarquia apresentada aqui continua quase igual àquela apresentada no Dia 5. A única diferença é que agora `BankAccount` é uma classe abstrata. Você não pode mais instanciar um objeto `BankAccount` diretamente.

Tornar `BankAccount` abstrata modela mais precisamente o funcionamento de contas bancárias. Quando você abre uma conta, abre uma conta de cheque ou uma conta de mercado financeiro. Você não abre uma conta de banco genérica. A Listagem 7.10 lista a única mudança na hierarquia.

**LISTAGEM 7.10 BankAccount.java**

```
public abstract class BankAccount {  
    // o restante é igual  
}
```

---

## Exposição do problema

Neste laboratório, você precisa escrever uma classe Bank. A classe Bank tem vários métodos.

As instâncias de Bank contêm contas. Entretanto, você precisa de uma maneira de controlar a quem pertencem as contas. `addAccount()` permite especificar um proprietário, sempre que você adiciona uma nova conta:

```
public void addAccount( String name, BankAccount account );
```

Você pode usar o nome do proprietário para acessar a conta correta posteriormente.

`totalHoldings()` permite que a classe Bank relate a quantidade total de dinheiro existente no banco:

```
public double totalHoldings();
```

`totalHoldings()` deve fazer um laço por todas as contas e totalizar o valor mantido no banco.

`totalAccounts()` permite que você consulte a instância de Bank para ver quantas contas ela possui correntemente:

```
public int totalAccounts();
```

`deposit()` permite que você deposte fundos em uma conta bancária específica:

```
public void deposit( String name, double amount );
```

`deposit()` é um método de conveniência que o libera de ter de recuperar uma conta específica, antes de poder adicionar fundos nela. Em vez disso, `deposit()` permite que você deposte fundos diretamente através do banco.

`balance()` permite que você recupere o saldo de uma conta específica:

```
public double balance( String name )
```

Assim como `deposit()`, `balance()` é um método de conveniência.

`addAccount()` armazena uma conta sob determinado nome. Existem várias maneiras de implementar essa funcionalidade. Entretanto, algumas estratégias são mais fáceis de implementar do que outras.

Para este laboratório, você deverá considerar `java.util.Hashtable`. `Hashtable` permite que você armazene e recupere pares chave/valor.

Considere esta API consolidada:

```
public Object get( Object key );
public Object put ( Object key, Object value );
public int size();
public java.util.Enumeration elementos();
```

Aqui está um exemplo de `Hashtable`:

```
java.util.Hashtable table = new java.util.Hashtable();
table.put( "LANGUAGE", "JAVA" );
String name = table.get( "LANGUAGE" );
```

Esse exemplo armazena o valor JAVA sob a chave LANGUAGE. Para recuperar o valor posteriormente, você simplesmente chama get() e passa a chave correta.

Estudando a API, você notará que os métodos get() e put() retornam Object. Assim, se você fosse armazenar uma String, obteria o valor de volta como um Object.

Na linguagem Java, todos os objetos herdam de Object. A Hashtable foi escrita para tratar com Object, de modo que funcionará para todos os objetos Java. Entretanto, e se você armazenar um objeto CheckingAccount na Hashtable e quiser tratá-lo como um objeto CheckingAccount, após recuperá-lo? Como você faria isso na linguagem Java?

A linguagem Java fornece um modo de transformar uma referência de Object de volta para seu tipo correto. O mecanismo é conhecido como *conversão*. A instrução a seguir é inválida na linguagem Java:

```
CheckingAccount account = table.get( "CHECKING_ACCOUNT" );
```

Em vez disso, você precisará realizar uma conversão, antes de poder armazenar uma referência de Object em uma variável CheckingAccount:

```
CheckingAccount account = (CheckingAccount) table.get( "CHECKING_ACCOUNT" );
```

Você precisa tomar cuidado enquanto converte. A conversão pode ser perigosa. Por exemplo, a conversão a seguir é inválida:

```
HappyObject o = new HappyObject();
table.put( "HAPPY", o );
(CheckingAccount) table.get( "HAPPY" );
```

Quando você converte, deve certificar-se de que o Object que está convertendo seja realmente do tipo TYPE. Assim, para este laboratório, quando você recuperar um objeto BankAccount da Hashtable, desejará converter para BankAccount. Como exemplo, considere a conversão a seguir:

```
BankAccount b = (BankAccount) table.get( "ACCOUNT1" );
```

Se você tentar realizar uma conversão inválida na linguagem Java, ela lançará uma exceção ClassCastException.

Assim como o Laboratório 1, o Laboratório 2 fornece um driver para ajudá-lo a testar sua solução. Certifique-se de consultar BankDriver.



A próxima seção discutirá as soluções do Laboratório 2. Não prossiga até concluir o Laboratório 2.

## Soluções e discussão

A Listagem 7.11 apresenta uma possível implementação de Bank.

### LISTAGEM 7.11 Bank.java

```
public class Bank {  
    private java.util.Hashtable accounts = new java.util.Hashtable();  
  
    public void addAccount( String name, BankAccount account ) {  
        accounts.put( name, account );  
    }  
  
    public double totalHoldings() {  
        double total = 0.0;  
  
        java.util.Enumeration enum = accounts.elements();  
        while( enum.hasMoreElements() ) {  
            BankAccount account = (BankAccount) enum.nextElement();  
            total += account.getBalance();  
        }  
        return total;  
    }  
  
    public int totalAccounts() {  
        return accounts.size();  
    }  
  
    public void deposit( String name, double amount ) {  
        BankAccount account = retrieveAccount( name );  
        if( account != null ) {  
            account.depositFunds( amount );  
        }  
    }  
  
    public double balance( String name ) {  
        BankAccount account = retrieveAccount( name );  
        if( account != null ) {  
            return account.getBalance();  
        }  
        return 0.0;  
    }  
  
    private BankAccount retrieveAccount( String name ) {  
        return (BankAccount) accounts.get( name );  
    }  
}
```

Internamente, essa solução usa `java.util.Hashtable` para conter todos os objetos `BankAccount`. Em vez de fornecer seu próprio mecanismo de armazenamento e recuperação, essa implementação tira proveito da reutilização, utilizando as classes fornecidas pela linguagem Java.

`balance()`, `deposit()`, `addAccount()` e `totalHoldings()`, todos demonstram o polimorfismo de inclusão. Esses métodos funcionarão para qualquer subclasse de `BankAccount` que você possa criar.

Após concluir este laboratório, você deverá ter uma idéia melhor dos mecanismos de polimorfismo. No Dia 5, os objetos `BankAccount` mostraram a conveniência da herança. A herança permitiu que você criasse subclasses rapidamente, programando apenas o que era diferente entre as contas. O polimorfismo simplifica ainda mais seu código, fornecendo um mecanismo para programação genérica.

## Laboratório 3: conta de banco — usando polimorfismo para escrever código à prova do futuro

Por toda a discussão sobre polimorfismo, você ouviu o termo software à prova do futuro. O que é exatamente software à prova do futuro? Software à prova do futuro é simplesmente um software que se adapta facilmente à mudança nos requisitos.

Os requisitos mudam o tempo todo. Quando você começa a escrever um programa pela primeira vez, os requisitos podem mudar, enquanto você aprende mais a respeito do problema que está resolvendo. Uma vez escrito, seus usuários esperarão e exigirão novos recursos de seu software. Se você criar software à prova do futuro, não terá de reescrevê-lo completamente, cada vez que obtiver um novo requisito.

Vamos considerar um exemplo de mudança de requisitos. A Listagem 7.12 apresenta uma nova classe `MoodyObject`: `CarefreeObject`.

### LISTAGEM 7.12 CarefreeObject.java

---

```
public class CarefreeObject extends MoodyObject {
    // redefine o humor da classe
    protected String getMood() {
        return "carefree";
    }

    // especialização
    public void whistle() {
        System.out.println("whistle, whistle, whistle...");
    }
}
```

---

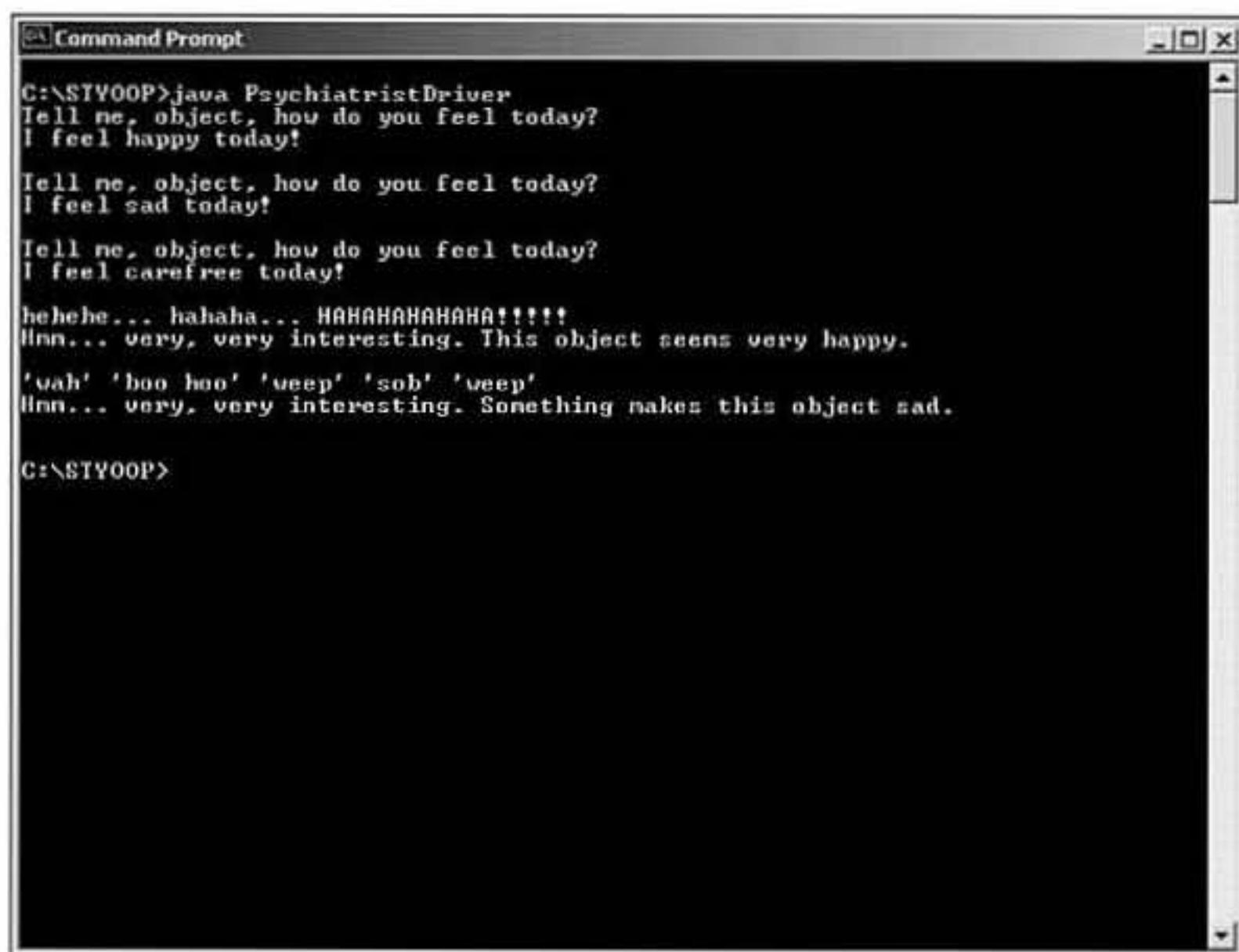
A Listagem 7.13 mostra o PsychiatristDriver atualizado.

**LISTAGEM 7.13** PsychiatristDriver.java

```
public class PsychiatristDriver {  
  
    public static void main( String [] args ) {  
        HappyObject happy = new HappyObject();  
        SadObject sad = new SadObject();  
        CarefreeObject carefree = new CarefreeObject();  
        PsychiatristObject psychiatrist = new PsychiatristObject();  
  
        // usa polimorfismo de inclusão  
        psychiatrist.examine( happy );  
        psychiatrist.examine( sad );  
        psychiatrist.examine ( carefree );  
  
        // usa sobrecarga para que possamos observar os objetos  
        psychiatrist.observe( happy );  
        psychiatrist.observe( sad );  
    }  
}
```

A Figura 7.2 mostra a saída que você verá ao executar PsychiatristDriver.

**FIGURA 7.2**  
*A saída correta de PsychiatristDriver.*



```
C:\STV0OP>java PsychiatristDriver  
Tell me, object, how do you feel today?  
I feel happy today!  
Tell me, object, how do you feel today?  
I feel sad today!  
Tell me, object, how do you feel today?  
I feel carefree today!  
hehehe... hahaha... HAHAHAHAHAHAT!!!!  
Hnn... very, very interesting. This object seems very happy.  
'wah' 'boo hoo' 'weep' 'sob' 'weep'  
Hnn... very, very interesting. Something makes this object sad.  
C:\STV0OP>
```

Aqui, você vê que PsychiatristObject é à prova do futuro. A qualquer momento, você pode adicionar novos objetos MoodyObject, todos com seu próprio comportamento personalizado. PsychiatristObject pode simplesmente usar os novos subtipos.



Você pode observar que esse exemplo se concentra no método `examine()`. O polimorfismo de inclusão possibilita um software realmente à prova do futuro. Entretanto, se PsychiatristObject quiser usar `observe()` em um novo subtipo, você precisará atualizar PsychiatristObject também.

MoodyObject é um exemplo simples. Entretanto, tente imaginar como você poderia estender essa idéia para programas mais complexos!

## Exposição do problema

Sua tarefa é testemunhar a programação à prova do futuro em primeira mão. No último laboratório, você escreveu uma classe Bank. A classe Bank pode trabalhar com qualquer subtipo de BankAccount. Sua tarefa é criar um novo tipo de BankAccount: a RewardsAccount.

Assim como em SavingsAccount, a RewardsAccount aplica juros no saldo. Entretanto, para aumentar o número e o tamanho dos depósitos, o banco gostaria de introduzir um sistema de recompensas.

A RewardsAccount controla o número de depósitos em relação a certa quantidade de dólares: o nível de depósito para recompensa. Por exemplo, digamos que o nível de depósito para recompensa seja de US\$500 dólares. Sempre que o depositante deposita US\$500 ou mais, ele receberá um ponto de recompensa.



Mantenha RewardsAccount simples. Se o nível de depósito para recompensa for de US\$500 e o depositante depositar US\$500, ele receberá um ponto de recompensa. Se o depositante depositar US\$3.000, ele ainda deverá receber apenas um ponto de recompensa.

Junto com os métodos definidos por BankAccount, RewardsAccount também deve fornecer um mecanismo para recuperar e reconfigurar o número de pontos de recompensa recebidos. RewardsAccount também precisa de um modo para configurar e obter o nível de depósito para recompensa.

Para este laboratório, talvez você queira voltar ao Dia 5 e reler as descrições de SavingsAccount e BankAccount. Este laboratório também inclui

um RewardsAccountDriver e um BankDriver atualizados. Certifique-se de usá-los para testar sua solução. Você também desejará ver BankDriver. O BankDriver demonstra como você pode adicionar um novo tipo de objeto em seu programa, sem ter de atualizar qualquer um dos outros objetos.



A próxima seção discutirá as soluções do Laboratório 3. Não prossiga até concluir o Laboratório 3.

## Soluções e discussão

A Listagem 7.14 apresenta uma possível solução para RewardsAccount.

### LISTAGEM 7.14 RewardsAccount.java

```
public class RewardsAccount extends SavingsAccount {  
    private double min_reward_balance;  
    private int qualifying_deposits;  
  
    public RewardsAccount( double initDeposit, double interest, double min ) {  
        super( initDeposit, interest );  
        min_reward_balance = min;  
    }  
  
    public void depositFunds( double amount ) {  
        super.depositFunds( amount );  
        if( amount >= min_reward_balance ){  
            qualifying_deposits++;  
        }  
    }  
  
    public int getRewardsEarned() {  
        return qualifying_deposits;  
    }  
  
    public void resetRewards() {  
        qualifying_deposits = 0;  
    }  
  
    public double getMinimumRewardBalance() {  
        return min_reward_balance;  
    }  
  
    public void setMinimumRewardBalance( double min ) {  
        min_reward_balance = min;  
    }  
}
```

RewardsAccount sobrepõe depositFunds(), de modo que pode verificar o saldo e os pontos de recompensa. A classe também adiciona métodos para recuperar o saldo das recompensas, reconfigurar o saldo, assim como obter e configurar o nível de depósito para recompensa.

A Listagem 7.15 apresenta o BankDriver atualizado.

---

**LISTAGEM 7.15** BankDriver.java

```
public class BankDriver {  
  
    public static void main( String [] args ) {  
        CheckingAccount ca = new CheckingAccount( 5000.00, 5, 2.50 );  
        OverdraftAccount oa = new OverdraftAccount( 10000.00, 0.18 );  
        SavingsAccount sa = new SavingsAccount( 500.00, 0.02 );  
        TimedMaturityAccount tma = new TimedMaturityAccount(  
            10000.00, 0.06, 0.05 );  
  
        Bank bank = new Bank();  
        bank.addAccount( "CHECKING", ca );  
        bank.addAccount( "OVERDRAFT", oa );  
        bank.addAccount( "SAVINGS", sa );  
        bank.addAccount( "TMA", tma );  
  
        System.out.println( "Total holdings(should be $25500.0): $" +  
            bank.totalHoldings() );  
        System.out.println( "Total accounts(should be 4): " +  
            Bank.totalAccounts() );  
  
        RewardsAccount ra = new RewardsAccount( 5000.00, .05, 500.00 );  
        bank.addAccount( "REWARDS", ra );  
  
        System.out.println( "Total holdings(should be $30500.0): $" +  
            bank.totalHoldings() );  
        System.out.println( "Total accounts(should be 5): " +  
            bank.totalAccounts() );  
  
        bank.deposit( "CHECKING", 250.00 );  
        double new_balance = bank.balance( "CHECKING" );  
        System.out.println( "CHECKING new balance (should be 5250.0): $" +  
            new_balance );  
    }  
}
```

---

Para usar a nova classe de conta existem dois passos que você deve dar.

Para o primeiro passo, você deve criar seu novo subtipo. Após criar seu novo subtipo, você precisa alterar seu programa para criar novas instâncias do objeto.

No caso de RewardsAccount, você deve atualizar o método `main()` de BankAccount, para criar instâncias de RewardsAccount. Entretanto, você não precisa mudar mais nada!

Quando você estiver escrevendo um programa real, percorrerá os mesmos passos para introduzir novos subtipos em seu programa. Primeiro, você precisará criar o novo subtipo.

Segundo, você precisará alterar seu código para que ele possa criar seu novo subtipo. Mas, é isso. Você não precisa alterar o restante de seu programa.

Posteriormente, você verá maneiras de tornar seus programas tão flexíveis, que poderá nem precisar alterar qualquer código de seu programa para que ele encontre e comece a usar novos subtipos.

## Laboratório 4: estudo de caso — estruturas condicionais Java e polimorfismo

A linguagem Java, assim como muitas outras, fornece um mecanismo de troca. Considere o método `day_of_the_week()` a seguir:

```
public void day_of_the_week(int day) {
    switch ( day ) {
        case 1:
            System.out.println( "Sunday" );
            break;
        case 2:
            System.out.println( "Monday" );
            break;
        case 3:
            System.out.println( "Tuesday" );
            break;
        case 4:
            System.out.println( "Wednesday" );
            break;
        case 5:
            System.out.println( "Thursday" );
            break;
        case 6:
            System.out.println( "Friday" );
            break;
        case 7:
            System.out.println( "Saturday" );
            break;
        default:
            System.out.println( day + " is not a valid day." );
            break;
    }
}
```

O método recebe um parâmetro: um valor inteiro representando o dia da semana. Então, o método testa todos os dias válidos da semana. Se o argumento day corresponder a um dos dias, o método imprimirá o nome do dia.

Geralmente falando, você usa estruturas condicionais (case ou if/else) para realizar lógica condicional. Com a lógica condicional, você procura certa condição nos dados. Se essa condição for satisfeita, você faz algo. Se outra condição for satisfeita, você faz algo inteiramente diferente. Todos os que têm base procedural devem estar familiarizados com tal estratégia de programação.

Se você se sente à vontade com estruturas condicionais, é hora de desaprender um pouco. A lógica condicional geralmente é considerada uma má prática de OO. Na verdade ela é tão má, que muitas linguagens OO não fornecem tais estruturas. A lógica condicional tem uma vantagem; ela o ajuda a detectar um projeto mal feito!

As estruturas condicionais são quase sempre más por natureza. Entretanto, elas freqüentemente se insinuam à sua frente, pois aparecem em muitas formas. Considere um método day\_of\_the\_week() ligeiramente diferente.

```
public void day_of_the_week( int day ) {
    if ( day == 1 ) {
        System.out.println( "Sunday" );
    } else if ( day == 2 ) {
        System.out.println( "Monday" );
    } else if ( day == 3 ) {
        System.out.println( "Tuesday" );
    } else if ( day == 4 ) {
        System.out.println( "Wednesday" );
    } else if ( day == 5 ) {
        System.out.println( "Thursday" );
    } else if ( day == 6 ) {
        System.out.println( "Friday" );
    } else if ( day == 7 ) {
        System.out.println( "Saturday" );
    } else {
        System.out.println( day + " is not a valid day." );
    }
}
```

Então, o que há de errado com as estruturas condicionais?

As estruturas condicionais são contrárias aos conceitos de OO. Na OO, você não solicita os dados de um objeto e depois faz algo com esses dados. Em vez disso, você solicita para que um objeto faça algo com seus dados. No caso do método day\_of\_the\_week(), você provavelmente obtém day de algum objeto. Você não deve estar processando dados brutos. Em vez disso, você deve solicitar ao objeto uma representação de string. As estruturas condicionais o obrigam a misturar responsabilidades. Cada lugar que usa os dados terá de aplicar a mesma lógica condicional.

Existem ocasiões em que as condicionais são absolutamente necessárias. Então, como você detecta ‘más’ estruturas condicionais?

Existem meios de saber quando uma estrutura condicional ‘boa’ fica ‘ruim’. Se você se encontrar atualizando uma estrutura case ou blocos if/else, sempre que adicionar um novo subtipo, as chances são de que a estrutura condicional é ‘ruim’. Não apenas isso é uma má prática de OO, como também é um pesadelo de manutenção. Você terá que certificar-se de atualizar cada condicional que teste os dados. Exigirá muito tempo para garantir que você não se esqueceu de atualizar algo!

## Corrigindo uma estrutura condicional

Considere o método a seguir:

```
public int calculate( String operação, int operand1, int operand2 ) {  
    if ( operation.equals( "+" ) ) {  
        return operand1 + operand2;  
    } else if ( operation.equals( "*" ) ) {  
        return operand1 * operand2;  
    } else if ( operation.equals( "/" ) ) {  
        return operand1 / operand2;  
    } else if ( operation.equals( "-" ) ) {  
        return operand1 - operand2;  
    } else {  
        System.out.println( "invalid operation: " + operation );  
        return 0;  
    }  
}
```

Tal método poderia aparecer em um problema de calculadora. O método `calculate()` recebe a operação, assim como os dois operandos, como argumentos. Então, ele efetua o cálculo solicitado.

Sendo assim, como você poderia corrigir o problema? Você pode corrigi-lo com objetos, é claro!

Quando você começar a eliminar lógica condicional, comece com os dados que está testando. Transforme os dados em um objeto.

Neste caso, você deve criar objetos adição, subtração, multiplicação e divisão. Todos esses objetos são operações. Assim, todos eles devem herdar de uma classe base comum. Conforme você já viu por todo este dia, a capacidade de substituição e o polimorfismo permitirão que você faça algumas coisas inteligentes com esses objetos.

Todos os objetos que você deve criar são operações; portanto, você sabe que precisa de uma classe base `Operation`. Mas o que uma classe `Operation` faz? Uma classe `Operation` calcula algum valor, dados dois operandos. A Listagem 7.16 apresenta uma classe `Operation`.

**LISTAGEM 7.16** Operation.java

```
public abstract class Operation {  
    public abstract int calculate( int operand1, int operand2 );  
}
```

---

As listagens 7.17, 7.18, 7.19 e 7.20 apresentam os vários objetos operação.

**LISTAGEM 7.17** Add.java

```
public class Add extends Operation {  
    public int calculate( int operand1, int operand2 ) {  
        return operand1 + operand2;  
    }  
}
```

---

**LISTAGEM 7.18** Subtract.java

```
public class Subtract extends Operation {  
    public int calculate( int operand1, int operand2 ) {  
        return operand1 - operand2;  
    }  
}
```

---

**LISTAGEM 7.19** Multiply.java

```
public class Multiply extends Operation {  
    public int calculate( int operand1, int operand2 ) {  
        return operand1 * operand2;  
    }  
}
```

---

**LISTAGEM 7.20** Divide.java

```
public class Divide extends Operation {  
    public int calculate( int operand1, int operand2 ) {  
        return operand1 / operand2;  
    }  
}
```

---

Cada operação implementa o método `calculate()` de sua própria maneira. Agora que você tem um objeto para cada operação, pode reescrever o método `calculate()` original.

```
public int calculate( Operation operation, int operand1, int operand2 ) {  
    return operation.calculate( operand1, operand2 );  
}
```

Transformando a operação em um objeto, você ganhou muita flexibilidade. No passado, você teria de atualizar o método sempre que quisesse adicionar uma nova operação. Agora, você pode simplesmente criar a nova operação e passá-la para o método. Você não precisa alterar o método de maneira alguma, para que ele funcione com a nova operação — ele simplesmente funciona.

## Exposição do problema

A linguagem Java fornece um operador chamado `instanceof`. O operador `instanceof` permite que você verifique o tipo subjacente de uma referência.

```
String s = "somestring";  
Object obj = s;  
System.out.println( (obj instanceof String) );
```

Esse segmento de código imprime `true`. `obj` contém uma instância de `String`. A maioria das linguagens de POO fornece um mecanismo semelhante.

Agora, considere a nova classe `Payroll`, na Listagem 7.21.

### Listagem 7.21 Payroll.java

```
public class Payroll {  
  
    private int      total_hours;  
    private int      total_sales;  
    private double   total_pay;  
  
    public void payEmployees( Employee [] emps ) {  
        for( int i = 0; i < emps.length; i ++ ) {  
            Employee emp = emps[i];  
            total_pay += emp.calculatePay();  
            emp.printPaycheck();  
        }  
    }  
  
    public void calculateBonus( Employee [] emps ) {  
        for( int i = 0; i < emps.length; i ++ ) {  
            Employee emp = emps[i];  
            if( emp instanceof HourlyEmployee ) {  
                System.out.println("Pay bonus to " + emp.getLastName() +  
                                  ", " + emp.getFirstName() + " $100.00.");  
            } else if ( emp instanceof CommissionedEmployee ) {  
                int bonus = ( (CommissionedEmployee) emp ).getSales() * 100;  
            }  
        }  
    }  
}
```

**LISTAGEM 7.21** Payroll.java (*continuação*)

```
        System.out.println("Pay bonus to " + emp.getLastName() + "," +
                           emp.getFirstName() + " $" + bonus );
    } else {
        System.out.println( "unknown employee type" );
    }
}
public void recordEmployeeInfo( CommissionedEmployee emp ) {
    total_sales += emp.getSales();
}

public void recordEmployeeInfo( HourlyEmployee emp ) {
    total_hours += emp.getHours();
}

public void printReport() {
    System.out.println( "Payroll Report:" );
    System.out.println( "Total Hours: " + total_hours );
    System.out.println( "Total Sales: " + total_sales );
    System.out.println( "Total Paid: $" + total_pay );
}
}
```

---

Essa classe Payroll tem um método calculateBonus(). Esse método recebe um array de objetos Employee, descobre qual é o tipo de cada um e calcula um bônus. Os objetos HourlyEmployee recebem um bônus fixo de US\$100, enquanto os objetos CommissionedEmployee recebem US\$100 por cada venda.

Sua tarefa é eliminar a lógica condicional encontrada em calculateBonus(). Comece atacando os dados que o método está testando. Neste caso, ele está testando em um objeto. Então, o que está errado?

Em vez de solicitar o bônus ao objeto, o método solicita a ele alguns dados e, em seguida, calcula um bônus, usando esses dados. Em vez disso, o método deve solicitar os dados ao objeto.

Você pode fazer download do código-fonte das classes Payroll, Employee, HourlyEmployee e CommissionedEmployee. Também existe um PayrollDriver fornecido, para que você possa testar facilmente sua solução.



A próxima seção discutirá as soluções do Laboratório 4. Não prossiga até concluir o Laboratório 4.

## Soluções e discussão

Para resolver esse problema, você deve adicionar um método `calculateBonus()` diretamente em cada objeto `Employee`. Isso poderia parecer como se você estivesse caindo na armadilha 1 do Dia 6. Entretanto, está correto mover o método para a classe base, pois todas as subclasses sabem como calcular seu bônus. Na verdade, isso já deveria estar na classe há muito tempo.

As listagens 7.22, 7.23 e 7.24 apresentam as alterações exigidas.

---

### LISTAGEM 7.22 Employee.java

---

```
public abstract class Employee {  
    public abstract double calculateBonus();  
    // reduzido por brevidade, o restante diz o mesmo  
}
```

---

---

### LISTAGEM 7.23 HourlyEmployee.java

---

```
public class HourlyEmployee extends Employee {  
    public double calculateBonus() {  
        return 100.00;  
    }  
    // reduzido por brevidade, o restante diz o mesmo  
}
```

---

---

### LISTAGEM 7.24 CommissionedEmployee.java

---

```
public class CommissionedEmployee extends Employee {  
    public double calculateBonus() {  
        return 100.00 * getSales();  
    }  
    // reduzido por brevidade, o restante diz o mesmo  
}
```

---

Com essas alterações, você pode atualizar a classe `Payroll`, como a Listagem 7.25 demonstra.

---

### LISTAGEM 7.25 Payroll.java

---

```
public class Payroll {  
    public void calculateBonus( Employee [] emps ) {  
        for(int i = 0; i < emps.length; i ++ ) {  
            Employee emp = emps[i];  
            System.out.println("Pay bonus to " + emp.getLastName() + ", " +  
                emp.getFirstName() + " $" + emp.calculateBonus())  
    }  
}
```

**LISTAGEM 7.25 Payroll.java (continuação)**

```
);  
    }  
}  
// reduzido por brevidade, o restante diz o mesmo  
}
```

---

Vamos! Acabou-se a lógica condicional irritante!

**DICA**

Dicas sobre estruturas condicionais:

- Evite o uso de estruturas condicionais case ou if/else.
- Considere os blocos if/else grandes com ceticismo.
- Cuidado com alterações em cascata. Se uma alteração exige muitas mudanças condicionais, talvez você tenha um problema.
- instanceof é um sinal de perigo muito grande.
- if/else, case e instanceof são “culpados até prova em contrário”.

**DICA**

Dicas para a eliminação de estruturas condicionais:

- Transforme os dados em objetos.
- Se os dados já são um objeto, adicione um método no objeto.
- Evite verificações de instanceof; use polimorfismo em vez disso.

## Resumo

Hoje, você completou quatro laboratórios. O Laboratório 1 lhe deu a chance de experimentar alguns dos mecanismos básicos do polimorfismo. O Laboratório 2 permitiu aplicar o que você aprendeu no Laboratório 1, em um exemplo mais complicado. O Laboratório 3 deve ter finalmente respondido a pergunta, “o que é exatamente software à prova do futuro?”. O Laboratório 3 resume o motivo pelo qual você desejará usar polimorfismo. Finalmente, o Laboratório 4 forneceu algo para tomar cuidado enquanto estiver programando. Ele também mostrou como o polimorfismo pode ser útil, quando utilizado corretamente.

Juntos, todos esses laboratórios reforçam as lições sobre polimorfismo. Eles fornecem o que você precisa saber para tirar vantagem do conceito corretamente. Esperamos que, após concluir esses laboratórios, você veja seus programas do ponto de vista do polimorfismo. A verdadeira programação OO exige uma maneira diferente de pensar a respeito do software. As vantagens da OO surgem realmente, quando você pode pensar de forma polimórfica.

## Perguntas e respostas

- P.** Parece que o polimorfismo de inclusão é mais conveniente do que a sobrecarga, pois eu só preciso escrever um método e fazer com que ele funcione com muitos tipos diferentes. Por que eu usaria sobrecarga em vez disso?
- R.** Existem ocasiões em que a sobrecarga é uma escolha melhor. Um método que usa inclusão só funciona se estiver processando objetos relacionados. A sobrecarga permite que você reutilize um nome de método dentre um grupo de métodos, cujos argumentos podem não estar relacionados. Você não pode fazer isso com a inclusão (embora possa usar uma combinação de inclusão e sobrecarga).

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. A partir das soluções de laboratório, dê um exemplo de método sobrecarregado.
2. Qual problema a sobreposição apresenta?
3. Quais passos você precisa dar para alterar o comportamento em uma hierarquia polimórfica?
4. A partir dos laboratórios, encontre um exemplo de polimorfismo de inclusão.
5. Como você elimina lógica condicional?
6. Qual é a vantagem do polimorfismo de inclusão em relação à sobrecarga?
7. Na OO, qual é o relacionamento entre objetos e dados?
8. O que há de errado com as estruturas condicionais?
9. Qual é uma boa indicação de que uma estrutura condicional é ‘ruim’?
10. Explique o polimorfismo com suas próprias palavras.

## Exercícios

Não há exercícios hoje. Faça seus laboratórios!

# SEMANA 1

## Em revisão

Na semana um você não apenas aprendeu os fundamentos da programação orientada a objetos, mas também como e quando deve aplicá-los.

Você aprendeu que os três pilares da programação orientada a objetos são o *encapsulamento*, a *herança* e o *polimorfismo*. O encapsulamento permite que você construa software independente. O encapsulamento é conseguido através de abstração, ocultação da implementação e divisão da responsabilidade. A herança permite que você reutilize e estenda código já existente. Você aprendeu que existem três tipos de herança: para reutilização de implementação, para diferença e para substituição de tipo. O polimorfismo permite que um único nome represente código diferente. Os quatro tipos diferentes de polimorfismo são: polimorfismo de inclusão, polimorfismo paramétrico, sobrecarga e sobreposição.

O uso dos três pilares da programação orientada a objetos permite que você crie código que é:

- Natural
- Confiável
- Reutilizável
- Manutenível
- Extensível
- Oportuno

Embora essas informações tenham sido apresentadas nos dias 2, 4 e 6, os laboratórios dos dias 3, 5 e 7 são o que realmente reúne tudo. A experiência prática nesses laboratórios aumentou o seu entendimento de como escrever código orientado a objetos que atinge os objetivos mencionados.

# SEMANA 2

8

9

10

11

12

13

14

- 8 Introdução à UML
- 9 Introdução à AOO (Análise Orientada a Objetos)
- 10 Introdução ao POO (Projeto Orientado a Objetos)
- 11 Reutilizando projetos através de padrões de projeto
- 12 Padrões avançados de projeto
- 13 OO e programação de interface com o usuário
- 14 Construindo software confiável através de testes

# Panorama

Na primeira semana, você aprendeu os fundamentos da escrita de código orientado a objetos. Embora esse seja um primeiro passo na criação de um programa orientado a objetos, ainda há muito mais para aprender, antes que você possa começar a codificar.

Nesta semana, você ultrapassará a simples codificação e abordará o inteiro processo de desenvolvimento de software . Os passos do processo de desenvolvimento de software que você abordará são: análise, projeto, implementação e teste.

A análise orientada a objetos (AOO) é o primeiro passo no processo de desenvolvimento. A AOO permite que você entenda o problema que está tentando resolver. Após concluir a AOO, você deve conhecer os requisitos de seu programa, assim como toda a terminologia específica do domínio.

Após ter analisado o problema, você poderá começar a projetar sua solução. O Dia 10 descreverá o projeto orientado a objetos, o processo de pegar o modelo de domínio e criar o modelo de objetos que você usará durante a implementação. Os dias 11 e 12 apresentarão alguns atalhos de projeto.

O próximo passo é a implementação; escrever o código. É aí que você utiliza as informações apresentadas na primeira semana.

O teste é o estágio final do processo de desenvolvimento. É importante testar durante todo o estágio de implementação, assim como no final, para poder garantir um sistema livre de defeitos.

Esses assuntos complementarão o conhecimento que você adquiriu na primeira semana e permitirão que você tenha uma idéia e acompanhe o processo de desenvolvimento até ter um programa orientado a objetos totalmente desenvolvido.

# SEMANA 2

DIA 8

## Introdução à UML

Na semana anterior, você aprendeu as teorias básicas da programação orientada a objetos. Entretanto, simplesmente conhecer algumas técnicas e definições não o preparará adequadamente para aplicá-las. Você simplesmente mostra ferramentas a alguém, explica seu uso e propósito e depois manda essa pessoa construir uma casa? É claro que não! A programação não é diferente. A programação de sucesso só surge com experiência e boa metodologia. Nesta semana, você vai aprender a aplicar corretamente as ferramentas de OO que viu na semana anterior.

Hoje, você vai explorar a UML (*Unified Modeling Language*), assim como alguns dos aspectos mais elegantes do relacionamento entre objetos. As lições de hoje fornecerão a linguagem comum que você usará enquanto aprende a analisar seus problemas e a projetar soluções OO.

Hoje você aprenderá:

- Por que deve se preocupar com a Unified Modeling Language
- Como modelar suas classes usando UML
- Como modelar os vários relacionamentos entre classes
- Como reunir tudo

## Introdução à Unified Modeling Language

Quando um construtor constrói uma casa, ele não faz isso aleatoriamente. Em vez disso, o construtor constrói a casa de acordo com um conjunto de cópias heliográficas detalhadas. Essas cópias heliográficas dispõem o projeto da casa explicitamente. Nada é deixado ao acaso.

Agora, quantas vezes você ou alguém que você conheça construiu um programa aleatoriamente? Quantas vezes essa prática trouxe problemas para você?

A UML (*Unified Modeling Language*) tenta trazer as cópias heliográficas para o mundo do software. A UML é uma linguagem de modelagem padrão. A linguagem consiste em várias notações gráficas que você pode usar para descrever a arquitetura inteira de seu software. Os programadores, arquitetos e analistas de software usam *linguagens de modelagem* para descrever graficamente o projeto do software.

**Novo TERMO** Uma *linguagem de modelagem* é uma notação gráfica para descrever projeto de software. A linguagem também inclui várias regras para distinguir entre desenhos corretos e incorretos. São essas regras que tornam a UML uma linguagem de modelagem e não apenas um punhado de símbolos para desenho.

Uma linguagem de modelagem não é igual a um processo ou metodologia. Uma *metodologia* diz a você como projetar o software. Em vez disso, uma linguagem de modelagem ilustra o projeto que você criará enquanto segue uma metodologia.

**Novo TERMO** Uma *metodologia* define um procedimento para projetar software. As linguagens de modelagem capturam esse projeto graficamente.

A UML não é a única linguagem de modelagem. Entretanto, ela é um padrão amplamente aceito. Ao modelar software, é importante fazer isso em uma linguagem comum. Desse modo, outros desenvolvedores podem rápida e facilmente entender seus diagramas de projeto. Na verdade, os criadores da UML reuniram suas três linguagens de modelagem concorrentes — por isso, o *U(nified — unificada)* em UML. A UML fornece um vocabulário comum que os desenvolvedores podem usar para transmitir seus projetos.

**Novo TERMO** A *UML* é uma linguagem de modelagem padrão. A UML consiste na notação para descrever cada aspecto de um projeto de software.



Não é o objetivo deste livro apresentar uma introdução exaustiva da UML. Em vez disso, este livro apresentará as partes práticas que você pode usar imediatamente para descrever seu software.

É importante notar que uma linguagem de modelagem não diz nada a respeito de como chegar a seu projeto. Metodologias ou processos é que mostram as diretrizes de como analisar e projetar software.

**Novo TERMO** Uma *metodologia* ou *processo* descreve como projetar software. Uma metodologia freqüentemente contém uma linguagem de modelagem.



A UML apresenta um rico conjunto de ferramentas de modelagem. Como resultado, existem muitas informações que você pode colocar em seus modelos. Cuidado para não tentar usar toda notação existente ao modelar. Use apenas notação suficiente para transmitir seu projeto.

Lembre-se sempre de que o objetivo de seu modelo é transmitir seu projeto. Faça o que precisar para transmiti-lo e fique com isso.

## Modelando suas classes

Na semana anterior, você viu muito código. Quando você se aprofunda nele, o código está no nível mais baixo da documentação de seu software. Se seu código funciona, você tem certeza de ter seu projeto documentado.

Embora o código seja a documentação mais completa de seu projeto, pode ser extremamente difícil para outros mexerem nele — especialmente se não estiverem familiarizados com o código. A ‘documentação’ também é útil para alguém que não conheça a linguagem de implementação.

Em vez disso, você precisa de uma notação que lhe permita documentar seu projeto, para que outros possam entendê-lo imediatamente. Desse modo, outros poderão ver a estrutura de classes de alto nível e apenas se aprofundar nos detalhes, quando isso for necessário. De certa forma, uma notação gráfica o isola dos detalhes, para que você possa eximir-se de entender a estrutura de alto nível de um programa.



Criar documentação separada do código exige o comprometimento de mantê-la em sincronismo com o código.

Uma maneira pela qual a UML o ajuda a transmitir seu projeto é fornecendo um rico conjunto de notação para descrever suas classes. Usando essa notação, outros podem ver facilmente as principais classes que compõem o projeto de seu programa. Conforme você verá, a UML permite definir as classes, assim como descrever os relacionamentos de alto nível entre as classes.

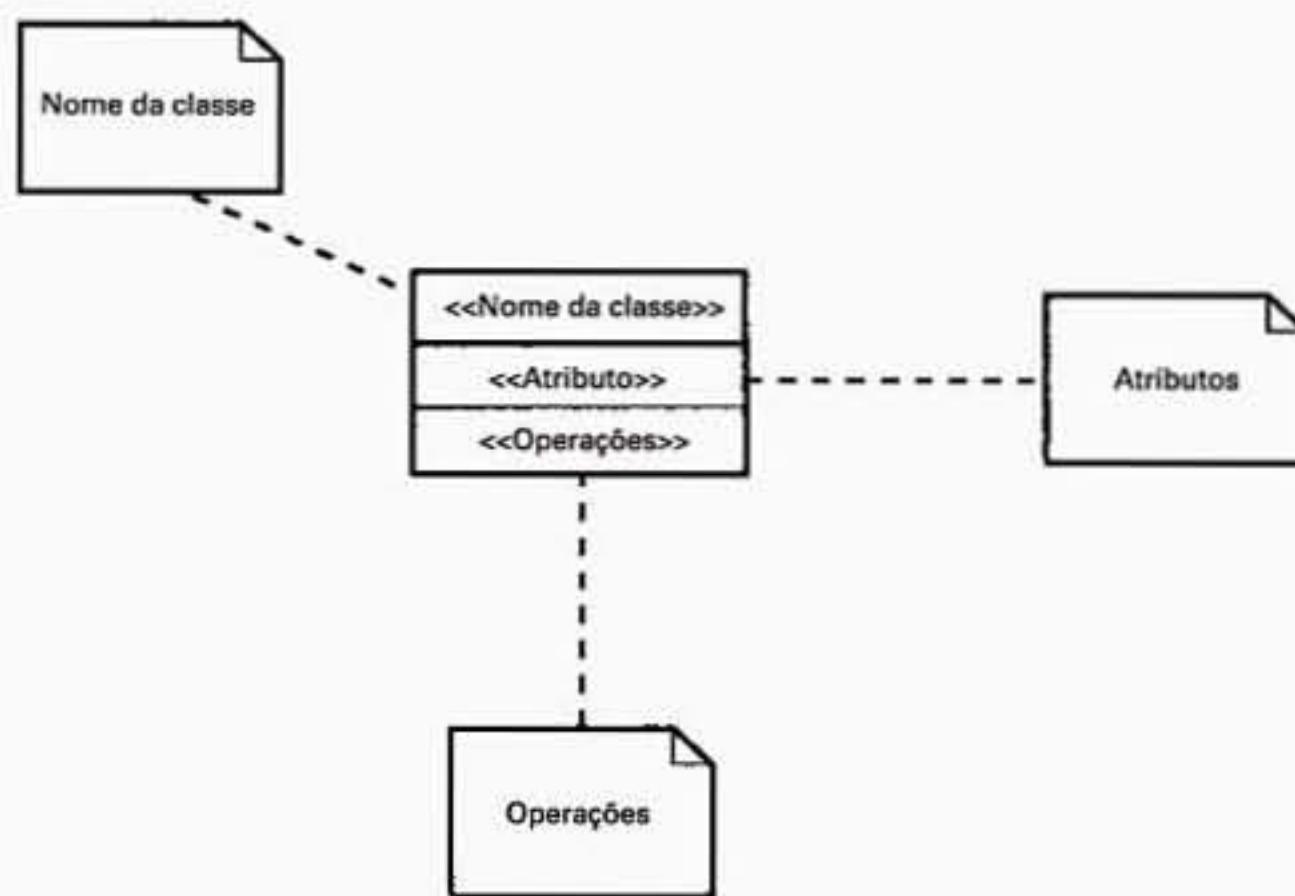
### Notação básica de classe

A UML fornece um rico conjunto de notação para modelar classes. Na UML, uma caixa representa a classe. A caixa superior sempre contém o nome da classe. A caixa do centro contém todos os atributos e a inferior contém as operações. Notas sobre seu modelo aparecem em caixas com cantos dobrados. A Figura 8.1 resume a estrutura básica de classes.



A UML faz diferença entre operação e métodos. Na UML, uma operação é um serviço que você pode solicitar de qualquer objeto de uma classe, enquanto um método é uma implementação específica da operação. As lições de hoje acompanharão a utilização da UML.

**FIGURA 8.1**  
*A notação de classe da UML.*



Dentro do modelo, você pode usar os caracteres `-`, `#` e `+`. Esses caracteres transmitem a *visibilidade* de um atributo ou de uma operação. O hífen (`-`) significa privado, o jogo da velha (`#`) significa protegido e o sinal de adição (`+`) significa público (veja a Figura 8.2).

**FIGURA 8.2**  
*A notação da UML para especificar visibilidade.*

Visibilidade
<code>+ public_attr</code> <code># protected_attr</code> <code>- private_attr</code>
<code>+ public_opr( )</code> <code># protected_opr( )</code> <code>- private_opr( )</code>

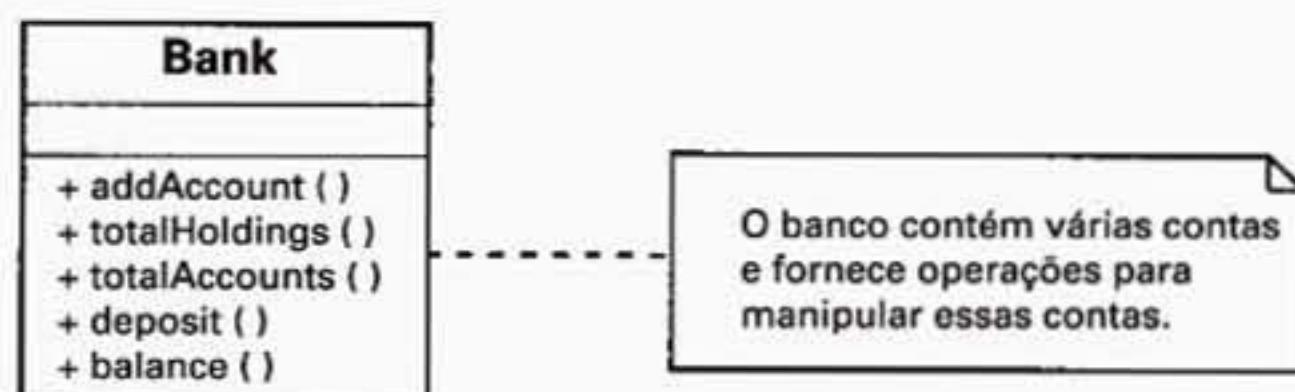
A Figura 8.3 ilustra a completa classe BankAccount dos dias 5 e 7.

**FIGURA 8.3**  
*Uma classe totalmente descrita.*

BankAccount
<code>- balance : double</code>
<code>+ depositFunds (amount : double) : void</code> <code>+ getBalance () : double</code> <code># setBalance () : void</code> <code>+ withdrawFunds (amount : double) : double</code>

Às vezes, uma nota ajudará a transmitir um significado que, de outro modo, ficaria perdido ou seria ignorado, como a nota da Figura 8.4.

**FIGURA 8.4**  
*Um exemplo detalhado de nota.*



Essas notas são a modelagem análoga à anotação adesiva do mundo real.

## Notação avançada de classe

A UML também define algumas outras notações, mais avançadas. O uso correto dessa notação o ajuda a criar modelos mais descriptivos.

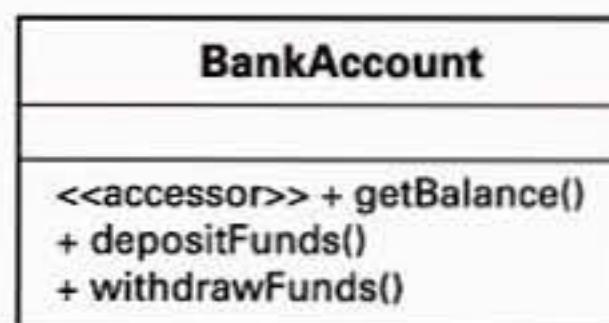
A UML o ajuda a ser mais descriptivo, permitindo que você amplie o vocabulário da própria linguagem, através do uso de *estereótipos*.

**Novo Termo** Um *estereótipo* é um elemento da UML que permite que você amplie o vocabulário da própria linguagem UML. Um estereótipo consiste em uma palavra ou frase incluída entre sinais de menor e maior duplos (<>). Você coloca um estereótipo acima ou ao lado de um elemento existente.

Por exemplo, a Figura 8.1 mostra o estereótipo <>Atributo>>. Esse estereótipo ilustra onde acrescentar atributos em um retângulo de classe. A Figura 8.5 ilustra outro estereótipo que informa um pouco sobre a operação.

**FIGURA 8.5**

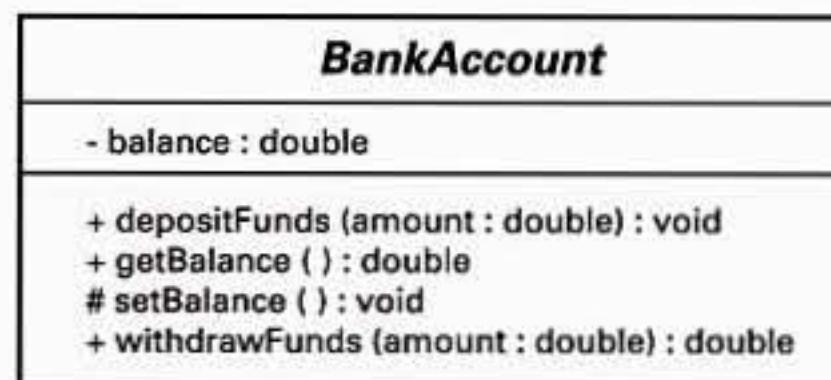
Um estereótipo que qualifica a operação.



Finalmente, você pode se lembrar que a classe *BankAccount* foi originalmente definida como uma classe concreta. Entretanto, o Dia 7 redefiniu a classe *BankAccount* como uma classe abstrata. A UML fornece uma notação para transmitir que uma classe é abstrata: o nome da classe abstrata é escrito em itálico. No caso de *BankAccount*, o nome deve ser escrito em itálico, conforme ilustrado na Figura 8.6.

**FIGURA 8.6**

O objeto *BankAccount* abstrato.



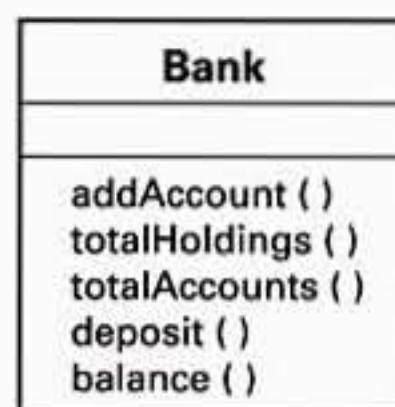
## Modelando suas classes de acordo com seus objetivos

As duas seções anteriores apresentaram muitas escolhas diferentes de notação. Dadas todas essas opções, como você sabe quais notações deve usar?

Você sempre precisa voltar às perguntas, “o que eu estou tentando transmitir?” e “para quem eu estou tentando transmitir isso?” O objetivo de um modelo é transmitir seu projeto o mais eficientemente (e simplesmente) possível.

Talvez seu objetivo seja transmitir a interface pública de uma classe. A Figura 8.7 poderia basta. Ela transmite adequadamente a interface pública de Bank, sem sobrecarregá-lo com os detalhes de argumentos de método ou atributos ocultos. Tal notação bastará, se você quiser simplesmente transmitir o que outros objetos poderiam fazer com instâncias de Bank.

**FIGURA 8.7**  
*Uma notação simples para Bank.*



Entretanto, tome a Figura 8.3 como outro exemplo. Essa figura documenta completamente todos os atributos e operações (público, protegido e privado) da classe BankAccount. Você poderia modelar uma classe com esse detalhe, se quisesse transmitir a definição inteira da classe para outro desenvolvedor. Ou talvez, quando você progredir em sua carreira OO, possa se tornar um arquiteto. Você poderia dar tal modelo para um desenvolvedor, para que ele pudesse criar a classe.

Então, a resposta da pergunta “como eu sei quais notações devo usar?” é que isso depende. Quando uma pessoa não-técnica pergunta a você o que faz, você responde de uma maneira que essa pessoa entenda. Quando um colega pergunta o que você faz, você geralmente dá uma resposta técnica. Modelar seu projeto não é diferente. Use o vocabulário que for apropriado para o que você estiver tentando fazer.



#### Dicas para a modelagem eficiente:

- Sempre faça a você mesmo a pergunta “o que eu estou tentando transmitir?” A resposta o ajudará a decidir exatamente o que você precisa modelar.
- Sempre faça a você mesmo a pergunta “para quem eu estou tentando transmitir a informação?” A resposta ditará o modo como você vai modelar.
- Sempre tente produzir o modelo mais simples que ainda tenha êxito em transmitir seu projeto.
- Não fique preso à linguagem de modelagem. Embora você não deva ser vago demais na semântica, não deve deixar que o fato de seguir a notação perfeitamente o impeça de concluir seus diagramas. Os perigos de paralisia ao modelar são reais — especialmente quando você está começando. Não se preocupe se seu modelo não estiver 100% perfeito. Preocupe-se somente se seu modelo não transmite corretamente o projeto.
- Finalmente, lembre-se de que a UML (ou qualquer linguagem de modelagem) é simplesmente uma ferramenta para ajudá-lo a transmitir o projeto. Ela não é um instrumento em si mesma. No final do dia, você ainda precisará produzir código.

## Modelando um relacionamento de classe

As classes não existem no vácuo. Em vez disso, elas têm relacionamentos complexos entre si. Esses relacionamentos descrevem como as classes interagem umas com as outras.

**Novo TERMO**

Um *relacionamento* descreve como as classes interagem entre si. Na UML, um relacionamento é uma conexão entre dois ou mais elementos da notação.

8

A UML reconhece três tipos de alto nível de relacionamentos de objeto:

- Dependência
- Associação
- Generalização

Embora a UML possa fornecer notação para cada um desses relacionamentos, os relacionamentos não são específicos da UML. Em vez disso, a UML simplesmente fornece um mecanismo e vocabulário comum para descrever os relacionamentos. Entender os relacionamentos, independentemente da UML, é importante em seu estudo de OO. Na verdade, se você simplesmente ignorar a notação e entender os relacionamentos, estará bem adiantado nos estudos.

### Dependência

Dependência é o relacionamento mais simples entre objetos. A dependência indica que um objeto depende da especificação de outro objeto.

**NOTA**

*Especificação* é uma maneira diferente de dizer interface ou comportamento.

**Novo TERMO**

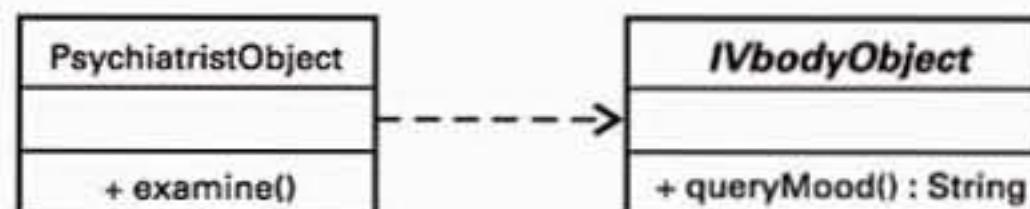
Em um *relacionamento de dependência*, um objeto é dependente da especificação de outro objeto. Se a especificação mudar, você precisará atualizar o objeto dependente.

Lembre-se dos laboratórios do Dia 7. Você pode dizer que `PsychiatristObject` depende de `MoodyObject`, por dois motivos. Primeiro, o método `examine()` de `PsychiatristObject` recebe um `MoodyObject` como argumento. Segundo, o método `examine()` chama o método `queryMood()` de `MoodyObject`. Se o nome ou lista de argumentos do método `queryMood()` mudar, você precisará atualizar o modo como `PsychiatristObject` chama o método. Do mesmo modo, se o nome da classe `MoodyObject` mudar, você terá de atualizar a lista de argumentos do método `examine()`.

A Figura 8.8 ilustra a notação UML do relacionamento de dependência entre `PsychiatristObject` e `MoodyObject`.

**FIGURA 8.8**

*Um relacionamento de dependência simples.*

**NOTA**

Tome nota do que a Figura 8.8 não diz. O elemento `PsychiatristObject` não contém cada método encontrado em `PsychiatristObject`. O mesmo vale para `MoodyObject`. Em vez disso, esse modelo de dependência contém apenas os recursos necessários para descrever o relacionamento de dependência.

Lembre-se de que a notação UML serve para transmitir informações. Ela não está lá para que você tente usar cada truque de modelagem do livro de UML!

Através da POO, você sempre tenta minimizar o máximo possível as dependências. Entretanto, é impossível remover todas as dependências entre seus objetos. Nem todas as dependências são criadas de modo igual. As dependências de interface geralmente estão corretas, enquanto as dependências de implementação quase nunca são aceitáveis.

**DICA****Quando você deve modelar dependências?**

Normalmente, você modela dependências quando quer mostrar que um objeto usa outro. Um lugar comum onde um objeto usa outro é através de um argumento de método. Por exemplo, o método `examine()` de `PsychiatristObject` recebe um `MoodyObject` como argumento. Você pode dizer que `PsychiatristObject` usa `MoodyObject`.

## Associação

Os relacionamentos de associação vão um pouco mais fundo do que os relacionamentos de dependência. As associações são relacionamentos estruturais. Uma associação indica que um objeto contém — ou que está conectado a — outro objeto.

**Novo Termo**

Uma *associação* indica que um objeto contém outro objeto. Nos termos da UML, quando se está em um relacionamento de associação, um objeto está conectado a outro.

Como os objetos estão conectados, você pode passar de um objeto para outro. Considere a associação entre uma pessoa e um banco, como ilustrado na Figura 8.9.

**FIGURA 8.9**

*Uma associação entre uma pessoa e um banco.*



A Figura 8.9 mostra que uma pessoa *empresta de* um banco. Na notação UML, toda associação tem um nome. Neste caso, a associação é chamada de *empresta de*. A seta indica a direção da associação.

**NOVO TERMO**

O nome da associação é um nome que descreve o relacionamento.

Cada objeto em uma associação também tem um papel, conforme indicado na Figura 8.10.

8

**FIGURA 8.10**

*Os papéis na associação.*

**NOVO TERMO**

Na associação, o papel de Pessoa é devedor e o papel de Banco é credor.

**NOVO TERMO**

O papel da associação é a parte que um objeto desempenha em um relacionamento.

Finalmente, a multiplicidade indica quantos objetos podem tomar parte em uma associação.

**NOVO TERMO**

A *multiplicidade* indica quantos objetos podem tomar parte na instância de uma associação.

A Figura 8.11 ilustra a multiplicidade da associação entre Pessoa e Banco.

**FIGURA 8.11**

*Multiplicidade.*



Essa notação nos informa que um banco pode ter um ou mais devedores e que uma pessoa pode utilizar 0 ou mais bancos.

**NOTA**

Você especifica suas multiplicidades através de um único número, uma lista ou com um asterisco (\*).

Um único número significa que determinado número de objetos — não mais e não menos — podem participar da associação. Assim, por exemplo, um 6 significa que seis objetos e somente seis objetos podem participar da associação.

\* significa que qualquer número de objetos pode participar da associação.

Uma lista define um intervalo de objetos que podem participar da associação. Por exemplo, 1..4 indica que de 1 a 4 objetos podem participar da associação. 3..\* indica que três ou mais objetos podem participar.

**DICA**

**Quando você deve modelar associações?**

Você deve modelar associações quando um objeto contiver outro objeto — o relacionamento *tem um*. Você também pode modelar uma associação quando um objeto usa outro. Uma associação permite que você modele quem faz o que em um relacionamento.

A UML também define dois tipos de associação: *agregação* e *composição*. Esses dois subtipos de associação o ajudam a refinar mais seus modelos.

## Agregação

Uma agregação é um tipo especial de associação. Uma agregação modela um relacionamento *tem um* (ou *parte de*, no jargão da UML) entre pares. Esse relacionamento significa que um objeto contém outro. *Pares* significa que um objeto não é mais importante do que o outro.

### Novo TERMO

Um *relacionamento todo/parte* descreve o relacionamento entre objetos onde um objeto contém outro.

### Novo TERMO

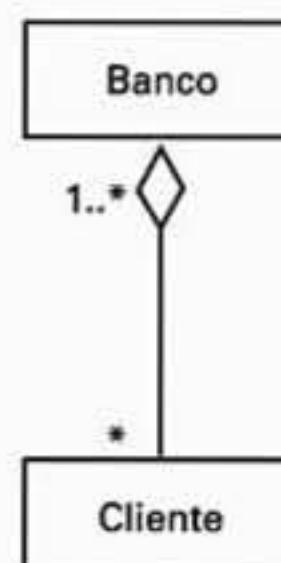
Uma *agregação* é um tipo especial de associação que modela o relacionamento ‘tem um’ de relacionamentos todo/parte entre pares.

*Importância*, no contexto de uma agregação, significa que os objetos podem existir independentemente uns dos outros. Nenhum objeto é mais importante do que o outro no relacionamento.

Considere a agregação ilustrada pela Figura 8.12.

**FIGURA 8.12**

*Aggregação entre um banco e seus clientes.*



Aqui, você vê que um Banco pode conter qualquer número de objetos Cliente. O losango aberto ajuda seu modelo a indicar qual objeto é o todo e qual é a parte. Aqui, o losango diz que Banco é o todo. Banco é o objeto que ‘tem um’ no relacionamento. Banco contém objetos Cliente. Em termos de programação, isso poderia significar que Banco contém um array de objetos Cliente.

### NOTA

Um losango aberto simboliza agregação. O losango toca o objeto que é considerado o *todo* do relacionamento: a classe que se refere à outra classe. O todo é constituído de *partes*. No exemplo anterior, Banco é o todo e os objetos Cliente são as partes.

Outro exemplo de agregação é um carro e seu motor. Um carro ‘tem um’ motor. Nessa agregação, o carro é o todo e a parte é o motor.

Como Banco e Cliente são independentes, eles são pares. Você pode dizer que o objeto Banco e o objeto Cliente são pares, porque os objetos Cliente podem existir independentemente do objeto Banco. Isso significa que, se o banco encerrar suas operações, os clientes não desaparecerão

com o banco. Em vez disso, os clientes podem se tornar clientes de outro banco. Do mesmo modo, um cliente pode sacar seus fundos e o banco continuará.

A agregação entre objetos funciona como esses exemplos reais. Um objeto pode conter outro objeto independente. Queue ou Vector é um exemplo de objeto que pode conter outros objetos, através da agregação.



Quando você deve modelar a agregação?

Você deve modelar uma agregação quando o objetivo de seu modelo for descrever a estrutura de um relacionamento de pares. Uma agregação mostra explicitamente o relacionamento estrutural todo/parte.

Entretanto, se você estiver mais interessado em modelar quem faz o que em um relacionamento, é melhor usar uma associação simples: sem o losango.

## **Composição**

A composição é um pouco mais rigorosa do que a agregação. A composição não é um relacionamento entre pares. Os objetos não são independentes uns dos outros.

A Figura 8.13 ilustra um relacionamento de composição.

**FIGURA 8.13**

## *Composição entre um banco e suas filiais*



Aqui, você vê que Banco pode conter muitos objetos Filial. O losango fechado diz que esse é um relacionamento de composição. O losango também diz quem ‘tem um’. Neste caso, Banco ‘tem um’, ou contém, objetos Filial.



## NOTA

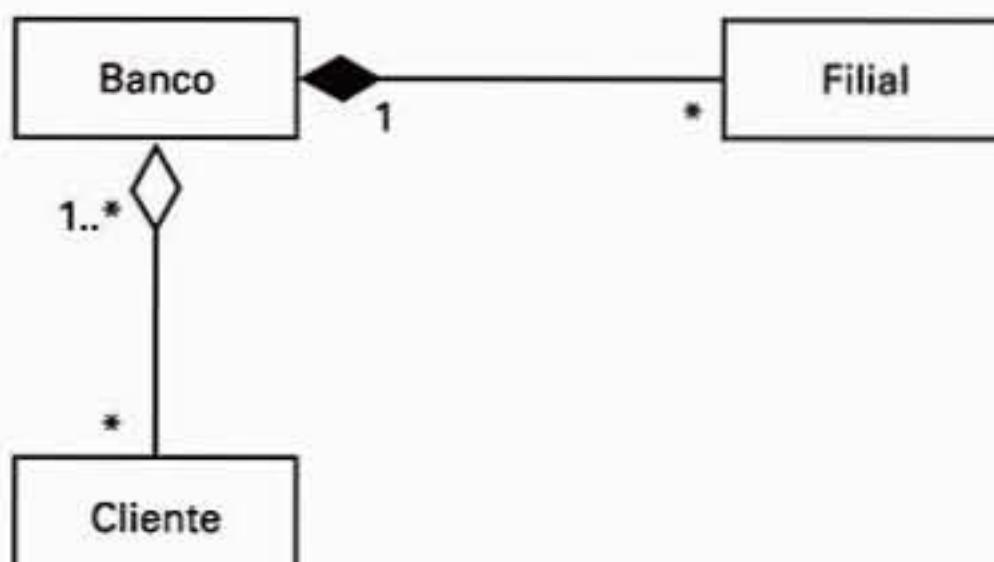
Um losango fechado simboliza a composição. O losango toca o objeto que é considerado o todo do relacionamento. O todo é constituído de partes. No exemplo anterior, Banco é o todo e os objetos Filial são as partes.

Como esse é um relacionamento de composição, os objetos Filial não podem existir independentemente do objeto Banco. A composição diz que, se o banco encerrar suas atividades, as filiais também fecharão. Entretanto, o inverso não é necessariamente verdade. Se uma filial fechar, o banco poderá permanecer funcionando.

Um objeto pode participar de uma agregação e de um relacionamento de composição ao mesmo tempo. A Figura 8.14 modela tal relacionamento.

**FIGURA 8.14**

*Banco em um relacionamento de agregação e de composição, simultaneamente.*

**DICA****Quando você deve modelar uma composição?**

Assim como a agregação, você deve modelar uma composição quando o objetivo de seu modelo for descrever a estrutura de um relacionamento. Uma composição mostra explicitamente o relacionamento estrutural todo/parte.

Ao contrário da agregação, a composição não modela relacionamentos todo/parte de pares. Em vez disso, a parte é dependente do todo. Voltando ao exemplo Banco, isso significa que quando o banco encerrar suas atividades, as filiais também fecharão.

Em termos de programação, isso significa que quando o objeto Banco for destruído, os objetos Filial também serão destruídos.

Novamente, se o objetivo de seu modelo for capturar os papéis dos objetos na associação, você deve usar uma associação simples.

**NOTA**

Lembre-se de que agregação e composição são simplesmente refinamentos ou subtipos da associação. Isso significa que você pode modelar agregação e composição como uma associação simples. Tudo depende do que você estiver tentando modelar em seu diagrama.

## Generalização

Um relacionamento de generalização é um relacionamento entre o geral e o específico. É a herança.

**NOVO TERMO**

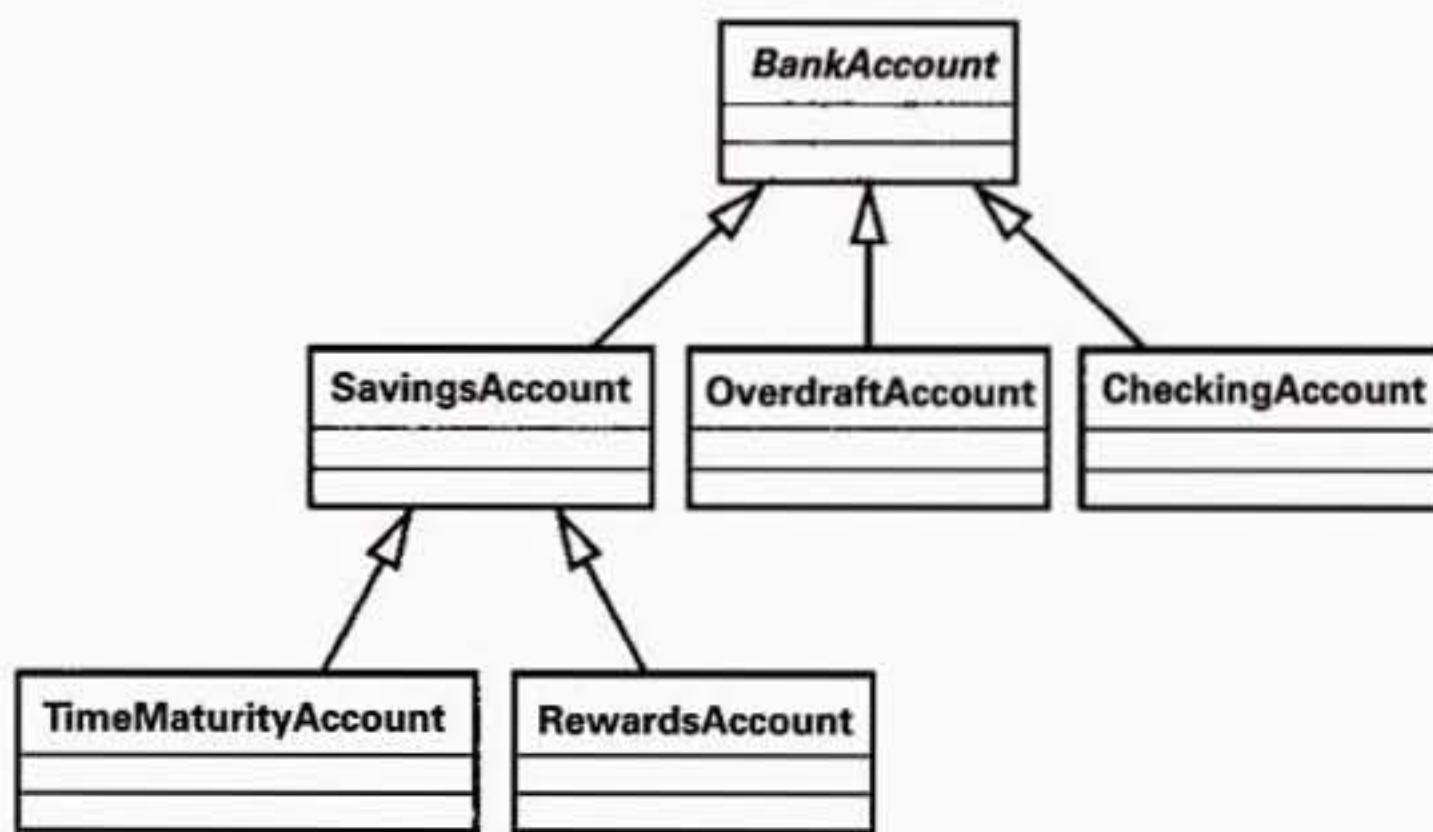
Um *relacionamento de generalização* indica um relacionamento entre o geral e o específico. Se você tem um relacionamento de generalização, então sabe que pode substituir uma classe filha pela classe progenitora.

A generalização incorpora o relacionamento ‘é um’ sobre o qual você aprendeu no Dia 4. Conforme foi aprendido no Dia 4, os relacionamentos ‘é um’ permitem que você defina relacionamentos com capacidade de substituição.

Através de relacionamentos com capacidade de substituição, você pode usar descendentes em vez de seus ancestrais, ou filhos em vez de seus progenitores.

A UML fornece uma notação para modelar generalização. A Figura 8.15 ilustra como você modelaria a hierarquia de herança BankAccount.

**FIGURA 8.15**  
A hierarquia de herança  
BankAccount.



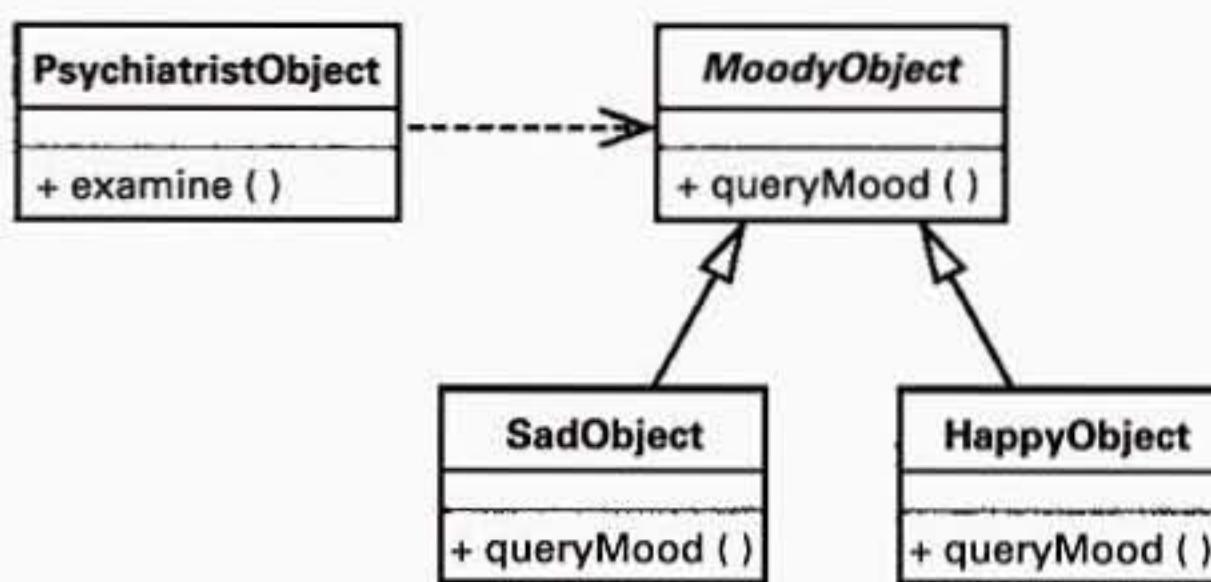
Uma linha cheia com uma seta fechada e vazada indica um relacionamento de generalização.

## Reunindo tudo

Agora que você já viu a modelagem básica de classe e os relacionamentos, pode começar a montar modelos bastante expressivos. A Figura 8.8 apresentou um exemplo de dependência simples. Usando o que aprendeu durante todo o dia, você pode tornar esse modelo um pouco mais expressivo. A Figura 8.16 expande o relacionamento modelado na Figura 8.8.

Figura página 198 em baixo

**FIGURA 8.16**  
Um modelo de dependência  
mais expressivo.

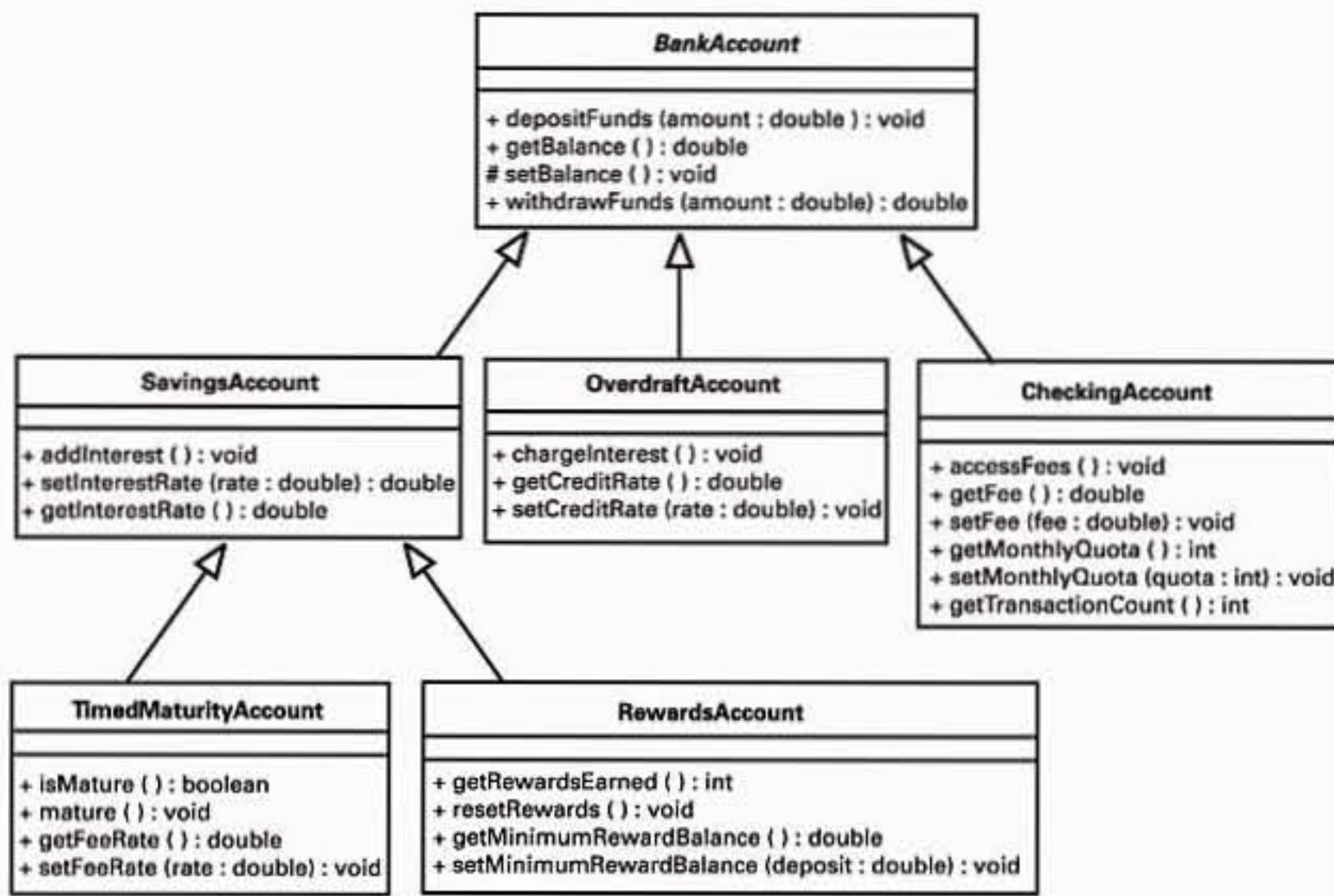


A Figura 8.16 acrescenta uma generalização para que você possa ver quais objetos pode substituir por MoodyObject nesse relacionamento.

Do mesmo modo, a Figura 8.17 expande a hierarquia de herança apresentada na Figura 8.15.

**FIGURA 8.17**

*Uma hierarquia de herança BankAccount mais detalhada.*



Examinando esse modelo, você pode ver exatamente o que cada classe acrescenta na hierarquia. Tal modelo poderia ajudar outros desenvolvedores a ver o que cada classe oferece, acima e além de suas descendentes.

Todos esses modelos têm um elemento comum. Cada modelo contém apenas informações suficientes, apenas notação suficiente, para transmitir a idéia. O objetivo desses modelos não é usar cada notação disponível.

Todos esses modelos também combinam diferentes elementos da UML. Como uma linguagem de programação, a UML permite que você combine suas várias partes de maneiras exclusivas. Através da combinação de vários elementos, você pode criar modelos muito expressivos.

## Resumo

Hoje, você aprendeu os fundamentos da modelagem de classe e relacionamentos. Após praticar os exercícios de hoje, você deverá conseguir começar a desenhar modelos de classe simples, usando a UML.

A UML fornece notações para modelar classes, assim como os relacionamentos entre objetos. A UML fornece notações para descrever três tipos de relacionamentos:

- Dependência
- Associação
- Generalização

A UML também reconhece dois subtipos de associação: agregação e composição. Combinando todos esses elementos, você pode gerar diagramas de classe expressivos. Seu domínio da UML é importante para documentar e transmitir seus projetos para outros.

## Perguntas e respostas

8

- P. Você pode misturar os três tipos de relacionamentos dentro do mesmo modelo?**
- R. Sim. Seu modelo pode ilustrar qualquer combinação dos relacionamentos delineados neste dia. O modelo existe para descrever os relacionamentos entre suas classes. Você deve modelar os relacionamentos entre suas classes.
- P. Como você usa a UML? Existem ferramentas específicas?**
- R. Você pode usar a UML como quiser. Você pode desenhar seus diagramas em uma ferramenta de modelagem, em um quadro negro ou em um guardanapo de papel. Depende da situação. Se você estiver em uma discussão interativa sobre o projeto, provavelmente desejará usar um quadro negro, pois atualizar um computador pode ser complicado.
- As ferramentas de modelagem por computador são melhor usadas quando você quer documentar formalmente um projeto.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. O que é UML?
2. Qual é a diferença entre uma metodologia e uma linguagem de modelagem?
3. Que tipo de relacionamento existe entre Employee e Payroll, no Laboratório 1 do Dia 7?
4. Examine cuidadosamente o modelo da Figura 8.15. Usando apenas o modelo, o que você pode dizer a respeito de MoodyObject?
5. Examine os laboratórios do Dia 7. Encontre um exemplo de dependência.
6. Na UML, o que os sinais a seguir simbolizam: +, #, -?
7. O Dia 2 apresentou a seguinte interface:

```
public interface Queue {  
    public void enqueue( Object obj );  
    public Object dequeue();
```

```
public boolean isEmpty();  
public Object peek();  
}
```

Que tipo de relacionamento Queue tem com os elementos que contém?

8. No Dia 3, Laboratório 3, a classe Deck criava várias cartas. Que tipo de relacionamento ‘tem um’ isso representa?
9. Como você ilustra que uma classe ou método é abstrato?
10. Qual é o objetivo final da modelagem? Quais consequências esse objetivo tem?
11. Explique associação, agregação e composição.
12. Explique quando você deve usar associação, agregação e composição.

## Exercícios

1. Modele a classe Queue definida na questão 7.
2. Modele um relacionamento de composição abelha/colméia.
3. Modele o relacionamento entre Bank e BankAccount do Laboratório 2, Dia 7.
4. Modele a associação entre um comprador e um comerciante. Especifique os papéis, a multiplicidade e o nome da dependência.
5. Modele a hierarquia de funcionários do Laboratório 2 do Dia 5. Através de seu modelo, transmita o que cada classe adiciona acima e além de suas descendentes.
6. Veja o Dia 6. Modele a hierarquia de herança PersonalityObject.

# SEMANA 2

## DIA 9

### Introdução à AOO (Análise Orientada a Objetos)

Ontem, você aprendeu a visualizar seus projetos de classe através de modelos de classe. Você viu como os modelos de classe podem ajudar outros desenvolvedores a entender melhor seu projeto, destacando os diferentes tipos de objetos e relacionamentos que eles encontrarão em seu software. As linguagens de modelagem, como a UML, fornecem a você e a seus colegas desenvolvedores uma linguagem comum para falar a respeito de projeto.

Entretanto, a questão ainda permanece; como você projeta software orientado a objetos? Os modelos simplesmente capturam um instantâneo de seu projeto. Eles não o ajudam a entender seus problemas ou a formular uma solução. Em vez disso, os modelos são simplesmente o resultado final do projeto de software. Como você chega lá?

Nos próximos dois dias, você vai aprender a respeito da AOO (Análise Orientada a Objetos) e POO (Projeto Orientado a Objetos). AOO é uma estratégia orientada a objetos para entender um problema. Você usa AOO para ajudar a entender o núcleo do problema que deseja resolver. Após entender seu problema, você pode começar a projetar uma solução. É aí que o Projeto Orientado a Objetos (POO) entra em ação. No restante da lição de hoje, você vai aprender a respeito de AOO.

Hoje você aprenderá:

- Sobre o processo de desenvolvimento de software
- Como a AOO o ajuda a entender seus problemas de software

- Como chegar a um entendimento de seu problema usando casos de uso
- Como usar a UML para visualizar sua análise
- Como construir seu modelo de domínio
- O que fazer com tudo que você cria durante a AOO

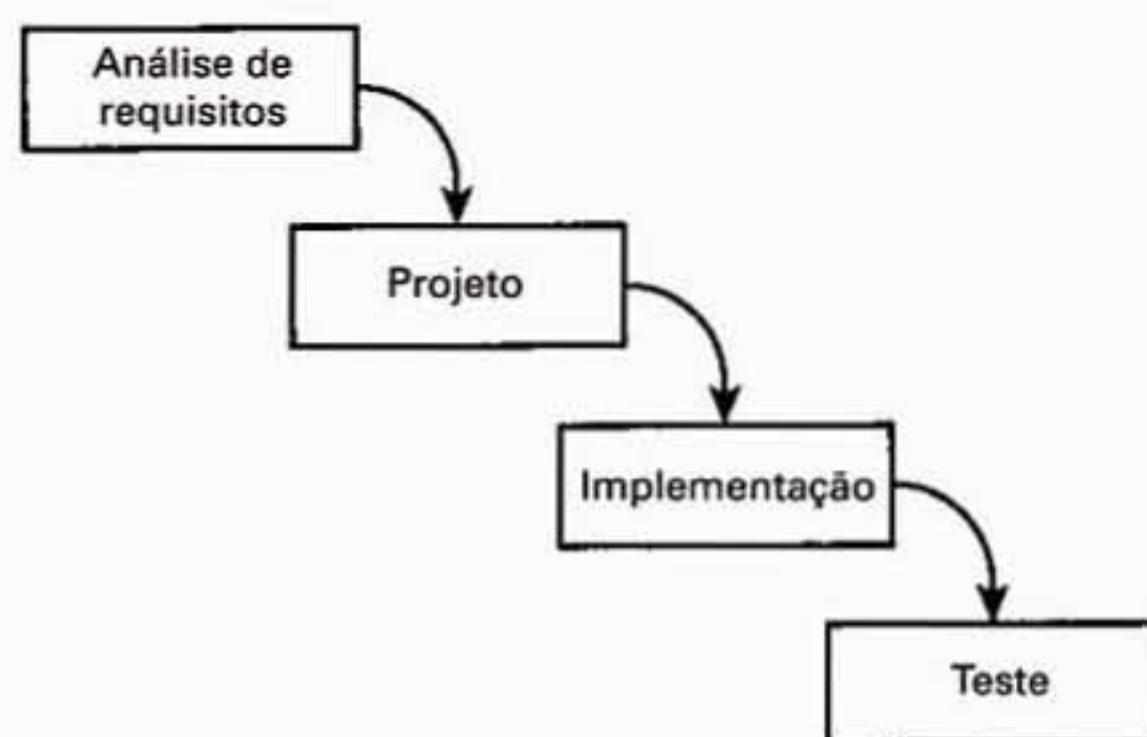
## O processo de desenvolvimento de software

Existem tantas maneiras de desenvolver software quanto existem desenvolvedores. Entretanto, uma equipe de desenvolvimento de software precisa de uma estratégia unificada para desenvolver software. Nada será feito, se cada desenvolvedor fizer sua própria atividade. As metodologias de software definem uma maneira comum de encarar o desenvolvimento de software. Uma metodologia freqüentemente conterá uma linguagem de modelagem (como a UML) e um processo.

**Novo Termo** Um *processo de software* mostra os vários estágios do desenvolvimento de software.

Um exemplo familiar de processo de software é o processo de cascata.

**FIGURA 9.1**  
*O processo de cascata.*



Conforme a Figura 9.1 ilustra, o processo de cascata é seqüencial e unidirecional. O processo é constituído de quatro estágios distintos:

1. Análise de requisitos
2. Projeto
3. Implementação
4. Teste

Quando segue o processo de cascata, você vai de um estágio para o próximo. Entretanto, uma vez que você complete um estágio, não há volta — exatamente como descer uma cascata ou um penhasco escarpado! O processo de cascata tenta evitar alteração, proibindo mudar quando um estágio está concluído. Tal estratégia protege os desenvolvedores de requisitos que mudam

constantemente. Entretanto, tal processo rígido freqüentemente resulta em software que não é o que você ou seu cliente quer.

Quando você analisa um problema, projeta uma solução e começa a implementar, seu entendimento do problema é continuamente aprofundado. O melhor entendimento de seu problema pode muito bem invalidar uma análise ou projeto anterior. Os requisitos podem até mudar enquanto você desenvolve (talvez um concorrente tenha acrescentado um novo recurso em seu produto). Infelizmente, o processo de cascata não pode enfrentar a realidade do moderno desenvolvimento de software — requisitos que mudam constantemente.

Embora este livro não tente impor nenhuma metodologia específica, há um processo que tem se mostrado muito eficiente para desenvolvimento orientado a objetos: o processo iterativo. Este livro impõe esse processo!

9

## O processo iterativo

O processo iterativo é o oposto do processo de cascata. O processo iterativo permite alterações em qualquer ponto do processo de desenvolvimento. O processo iterativo permite alteração adotando uma estratégia iterativa e incremental para o desenvolvimento de software.

**Novo Termo**

Um *processo iterativo* é uma estratégia *iterativa* e *incremental* para desenvolvimento de software. Outro modo de pensar a respeito do processo é como uma estratégia ‘evolutiva’. Cada iteração aperfeiçoa e elabora gradualmente um produto básico em um produto amadurecido.

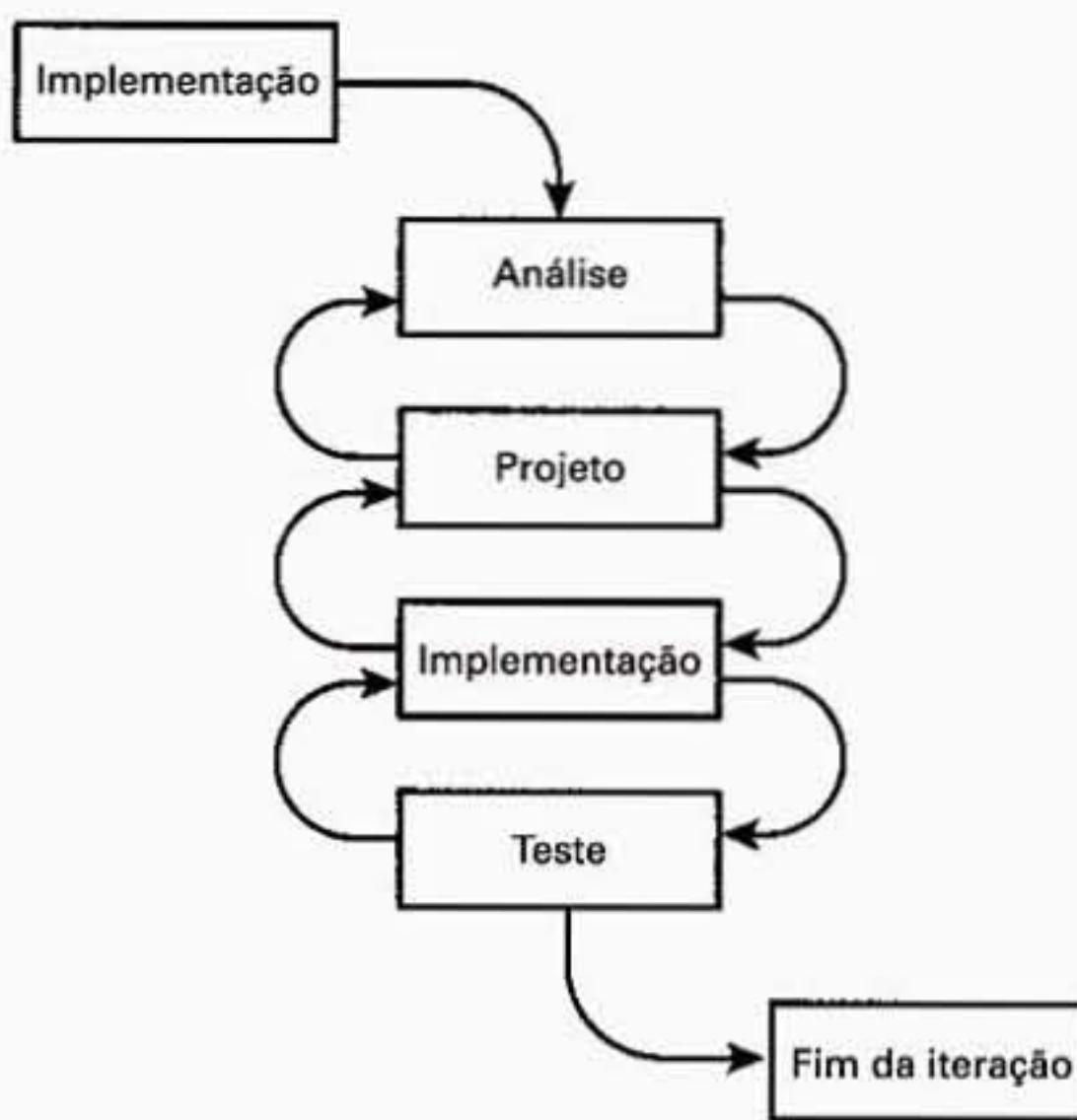
### Uma estratégia iterativa

Ao contrário do processo de cascata, o processo iterativo permite que você continuamente volte e refine cada estágio do desenvolvimento. Por exemplo, se você descobrir que o projeto simplesmente não funciona ao executar a implementação, pode voltar e fazer um projeto adicional e uma nova análise. É esse refinamento contínuo que torna o processo iterativo. A Figura 9.2 ilustra a estratégia.

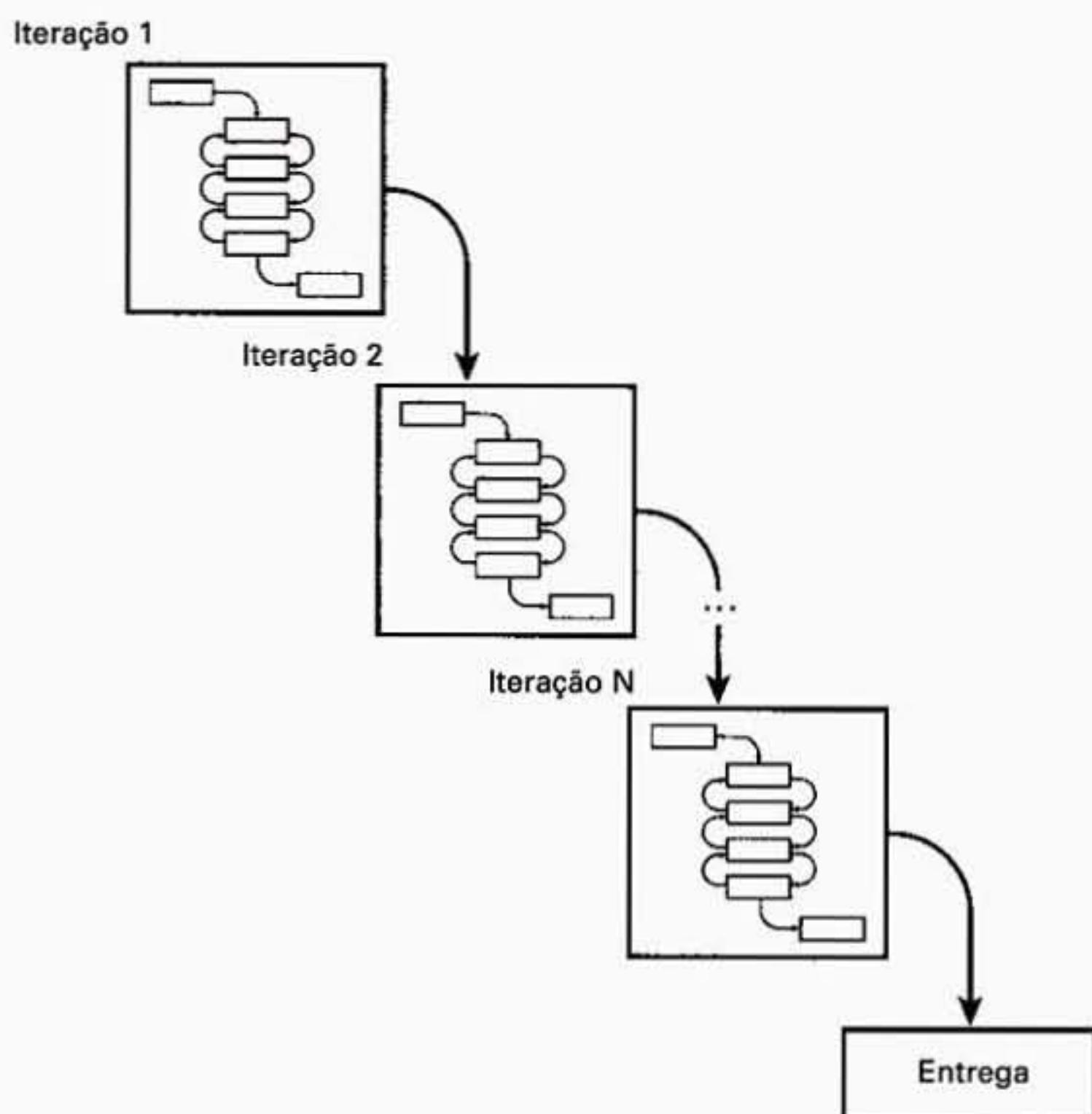
### Uma estratégia incremental

Ao seguir um processo iterativo, você não conclui simplesmente uma iteração grande que constrói o programa inteiro. Em vez disso, o processo iterativo divide o trabalho de desenvolvimento em várias iterações pequenas. A Figura 9.3 ilustra essa estratégia incremental.

**FIGURA 9.2**  
*Uma iteração.*



**FIGURA 9.3**  
*O processo iterativo.*



Cada iteração do processo introduz uma pequena melhoria incremental no programa. Essa melhoria pode ser um novo recurso ou um refinamento de um recurso já existente.

De qualquer modo, a iteração tem um objetivo específico e, no final da iteração, você tem uma melhoria notável na funcionalidade.

Imagine que você esteja criando um MP3 player. Durante uma iteração do projeto, você pode terminar o componente que reproduz um arquivo MP3. Para determinar se o componente funciona, você pode codificá-lo de modo que abra e reproduza um arquivo de música específico. Na próxima iteração, você pode adicionar a capacidade de escolher qual arquivo vai ser reproduzido. Em cada iteração, você tem um progresso mensurável. No final da primeira iteração, você pode ouvir o componente reproduzir uma música. No final da iteração seguinte, você tem um mecanismo que permite escolher dinamicamente uma música para tocar.

Seguindo uma estratégia iterativa, você vê o progresso constantemente. Por outro lado, se você tentar fazer tudo simultaneamente, poderá ser difícil ver qualquer forma mensurável de progresso. Em vez disso, o projeto parecerá constantemente atolado em um único lugar — nunca há qualquer resultado. Se um projeto nunca for adiante, o moral vai baixar e se tornará difícil determinar o que precisa ser feito em seguida. Moral baixa e confusão sobre o que fazer em seguida fragmentará e matará um projeto.

9



**ALERTA**  
Os processos iterativos precisam ser cuidadosamente monitorados para se garantir que eles não sejam simplesmente reduzidos a ‘cavar’ uma solução. A AOO e o POO fornecem tal verificação de sanidade.

O progresso constante fornece a você retorno constante. Você pode usar esse retorno como um modo de garantir se está no caminho certo. Se você tentar completar o projeto inteiro de uma vez, não saberá se criou a solução correta até terminar. Voltar e corrigir algo que não foi feito corretamente será muito mais dispendioso se você precisar voltar e reescrever o programa inteiro! A iteração, por outro lado, torna muito mais barato voltar e corrigir algo. Como você recebe retorno constante, é mais provável que identifique um problema mais cedo. Se você identificar seus problemas mais cedo, será mais fácil refazer uma iteração ou duas para corrigi-lo. É sempre mais desejável reescrever uma iteração do que reescrever um programa inteiro! Se você mantiver suas iterações pequenas, não perderá muito tempo, caso tenha de se desfazer de alguma delas.



**ALERTA**  
Se um problema chegar à base da iteração original, uma estratégia iterativa não poderá salvá-lo. Tal problema fundamental pode ser dispendioso demais para corrigir e pode danificar a qualidade do produto.

## Uma metodologia de alto nível

Este livro apresenta uma metodologia de desenvolvimento orientada a objetos informal. A metodologia seleciona e escolhe as técnicas que se mostraram eficazes a partir de outras metodologias. A metodologia consiste em um processo iterativo, no qual uma iteração tem quatro estágios:

- Análise
- Projeto
- Implementação
- Teste

**NOTA**

Após o estágio de teste, você também pode ter estágios de lançamento e manutenção. Esses são estágios importantes no ciclo de vida de um projeto de software. Entretanto, para os propósitos da lição de hoje, esses estágios serão omitidos. Hoje, você vai focalizar análise, projeto, implementação e teste.

As metodologias ‘reais’ freqüentemente enumeram estágios adicionais. Entretanto, quando você está aprendendo pela primeira vez, esses quatro estágios são aqueles que mais importam. Por isso, este livro se concentra nesses quatro estágios. O restante deste dia abordará a análise orientada a objetos.

## AOO (Análise Orientada a Objetos)

AOO (Análise Orientada a Objetos) é o processo usado para entender o problema que você está tentando resolver. Após completar a análise, você deverá entender os requisitos do problema, assim como o vocabulário do domínio do problema.

**NOVO TERMO**

*Análise orientada a objetos* é um processo que usa uma estratégia orientada a objetos para ajudá-lo a entender o problema que está tentando resolver. No final da análise, você deverá entender o domínio do problema e seus requisitos em termos de classes e interações de objetos.

Para projetar uma solução para um problema, você precisa entender como os usuários utilizarão o sistema. A resposta dessa pergunta são os requisitos do sistema. Os requisitos informam a você o que os usuários querem fazer com o sistema e quais tipos de respostas eles esperam receber.

**NOVO TERMO**

*Sistema* é o termo da AOO para um conjunto de objetos que interagem. Você pode dizer que esses objetos constituem um sistema ou modelo do problema.

Esses objetos são instâncias de classes derivadas de objetos concretos ou abstratos no domínio do problema que está sob estudo.

A análise também ajuda a se familiarizar com o domínio do problema. Estudando o domínio, você começa a identificar os objetos de que precisa para modelar corretamente o sistema.

A AOO, conforme o nome sugere, é uma estratégia orientada a objetos para análise de requisitos. A AOO utiliza uma estratégia baseada em OO, modelando o problema através de objetos e suas interações. Existem dois modelos principais. O modelo de caso de uso descreve como um usuário interage com o sistema. O modelo de domínio captura o vocabulário principal do siste-

ma. Usando o modelo de domínio, você começa a identificar os objetos que pertencem ao seu sistema. Um modelo de domínio corretamente construído pode resolver muitos problemas no mesmo domínio.

## Usando casos de estudo para descobrir o uso do sistema

Ao começar a analisar um problema, você primeiro precisa entender como seus usuários utilizam ou interagirão com o sistema. Esses usos compreendem os requisitos do sistema e prescrevem o sistema que você cria. Atendendo os requisitos de seus usuários, você produz um sistema útil.

**Novo Termo**

Os *requisitos* são os recursos ou características que o sistema deve ter para resolver determinado problema.

9

**Novo Termo**

Um modo de descobrir esses usos é através de *análise de casos de uso*. Através da análise você definirá vários casos de uso. Um caso de uso descreve como um usuário vai interagir com o sistema.

**Novo Termo**

*Análise de casos de uso* é o processo de descoberta de *casos de uso* através da criação de cenários e histórias com usuários em potencial ou existentes de um sistema.

**Novo Termo**

Um *caso de uso* descreve a interação entre o usuário do sistema e o sistema — como o usuário utilizará o sistema do seu próprio ponto de vista.

A criação de casos de uso é um processo iterativo. Existem vários passos que você deve dar durante cada iteração, para formalizar seus casos de uso. Para definir seus casos de uso, você deve:

1. Identificar os atores.
2. Criar uma lista preliminar de casos de uso.
3. Refinar e nomear os casos de uso.
4. Definir a seqüência de eventos de cada caso de uso.
5. Modelar seus casos de uso.

**NOTA**

Você não cria casos de uso no vácuo! Enquanto deriva seus casos de uso, você deve consultar aqueles que utilizarão o sistema — seus clientes. A participação do cliente é absolutamente fundamental para se descobrir os casos de uso (a não ser que você esteja escrevendo o software para si mesmo).

Seus clientes são os especialistas do domínio. Eles conhecem bem seu espaço de atuação e sabem do que precisam em seu software. Sempre se certifique de contar com o conhecimento deles e usá-lo para orientar os requisitos de seu software.

Fazer os usuários comporem histórias sobre seu dia ideal de interação com o sistema pode ser uma boa maneira de quebrar o gelo nessa atividade.



Antes de continuar com o dia, é importante dizer que os exemplos não tentam realizar uma análise completa de um site da Web on-line. Em vez disso, os exemplos ensinam os passos que você dará enquanto realizar uma análise real. Assim, muitos casos de uso serão omitidos.

Na próxima semana, você trabalhará com uma análise orientada a objetos completa.

## Identifique os atores

O primeiro passo na definição de seus casos de uso é definir os atores que usarão o sistema.

### Novo TERMO

Um *ator* é tudo que interage com o sistema. Pode ser um usuário humano, outro sistema de computador ou um chimpanzé.

Você precisa pedir aos seus clientes para que descrevam os usuários do sistema. As perguntas podem incluir as seguintes:

- Quem principalmente usará o sistema?
- Existem outros sistemas que usarão o sistema? Por exemplo, existem quaisquer usuários que não são seres humanos?
- O sistema se comunicará com qualquer outro sistema? Por exemplo, há um banco de dados já existente que você precise integrar?
- O sistema responde ao estímulo gerado por alguém que não seja usuário? Por exemplo, o sistema precisa fazer algo em certo dia de cada mês? Um estímulo pode ser proveniente de fontes normalmente não consideradas ao se pensar do ponto de vista puramente do usuário.

Considere uma loja da Web on-line. Uma loja on-line permite que usuários convidados naveguem pelo catálogo de produtos, verifique o preço dos itens e solicite mais informações. A loja também permite que usuários registrados comprem itens, assim como controla seus pedidos e mantém informações dos usuários.

A partir dessa breve descrição, você pode identificar dois atores: usuários convidados e usuários registrados. Cada um desses dois atores interage com o sistema.

A Figura 9.4 ilustra a notação UML para um ator: um desenho de pessoa com um nome. Você deve dar a cada um de seus atores um nome não ambíguo.

**FIGURA 9.4**  
*Os atores na UML.*





É importante evitar confusão ao nomear seus atores. Dê a cada ator um nome que identifique exclusivamente o ator.

Uma boa atribuição de nomes é fundamental. Os nomes devem ser simples e fáceis de lembrar.

É importante notar que determinado usuário do sistema pode assumir o papel de muitos atores diferentes. Um ator é um papel. Por exemplo, um usuário poderia entrar no site como convidado, mas posteriormente se conectar como registrado para poder fazer uma compra.



Um usuário pode assumir muitos *papéis* diferentes enquanto interage com um sistema. Um ator descreve o *papel* que o usuário pode assumir enquanto interage com o sistema.



Quando você começar a definir seus casos de uso, crie uma lista preliminar de atores. Não se atrapalhe ao identificar os atores. Será difícil descobrir todos os atores na primeira vez.

Em vez disso, encontre atores suficientes para começar e adicione os outros à medida que os descobrir.

9

Os atores são os instigadores de casos de uso. Agora que você já identificou alguns atores, pode começar a definir os casos de uso que eles executam.

## Crie uma lista preliminar de casos de uso

Para definir seus casos de uso, você precisa fazer algumas perguntas. Comece com sua lista de atores conhecidos. Você precisa perguntar o que cada ator faz com o sistema.

No caso da loja da Web on-line, você tem usuários registrados e usuários convidados. O que cada um desses atores faz?

Os usuários convidados podem fazer o seguinte:

1. Navegar pelo catálogo de produtos.
2. Pesquisar o catálogo de produtos.
3. Procurar um item específico.
4. Pesquisar o site.
5. Adicionar itens em um carrinho de compras e especificar a quantidade.
6. Ver o preço dos itens selecionados.
7. Mudar a quantidade de itens em seu carrinho.
8. Ver a lista de produtos popular e nova.
9. Navegar pela lista de itens desejados de outros usuários.
10. Solicitar mais informações sobre produto.

Os usuários registrados podem fazer o seguinte:

1. Tudo que o usuário convidado pode fazer.
2. Fazer uma compra.
3. Adicionar itens em sua lista de itens desejados.
4. Ver uma lista personalizada recomendada.
5. Manter sua conta.
6. Assinar notificações.
7. Tirar proveito de ofertas especiais personalizadas.
8. Controlar seus pedidos.
9. Assinar várias listas de distribuição.
10. Cancelar um pedido.



Provavelmente existem muito mais casos de uso. Entretanto, para nossos propósitos aqui e por brevidade, isso é suficiente para começar.

Quando tentar identificar casos de uso, você também deverá fazer a pergunta, “como um ator muda seu papel?”

No caso da loja on-line, um usuário convidado pode se tornar um usuário registrado, das seguintes maneiras:

- O usuário convidado pode se conectar com o site.
- O usuário convidado pode se registrar no site.

Um usuário registrado se torna um usuário convidado, como segue:

- Um usuário registrado pode se desconectar do site.

Até aqui, essas perguntas são orientadas pela interação. Você também pode adotar uma estratégia orientada por resultados para a descoberta. Por exemplo, você pode dizer que um usuário registrado recebe uma notificação. Um segundo ponto de vista pode ajudá-lo a descobrir casos de uso que você poderia ter ignorado, se simplesmente ficasse com o primeiro ponto de vista.

Finalmente, considere as várias entidades que os usuários manipulam. Aqui, você vê produtos, informações sobre a conta e várias listas de produto e descontos. Como todas essas entidades entram no sistema? Quem adiciona novos produtos e edita ou exclui produtos antigos?

Esse sistema precisará de um terceiro ator, o administrador. Passando pelo processo anteriormente delineado, você pode verificar que os administradores podem fazer o seguinte:

1. Adicionar, editar e excluir produtos.
2. Adicionar, editar e excluir incentivos.
3. Atualizar informações de conta.

As perguntas podem levar a outras perguntas. Por exemplo, quem atualiza a lista de produtos populares? Quem envia notificações e correspondências para as listas de distribuição? Um quarto ator, o próprio sistema, executa todas essas ações.

## Refine e nomeie os casos de uso

Agora que você tem uma lista preliminar de casos de uso, precisa refinar a lista. Em particular, você desejará procurar oportunidades de dividir ou combinar os casos de uso.

9

### Dividindo casos de uso

Cada caso de uso deve executar um objetivo principal. Quando você encontrar um caso de uso que estiver fazendo muita coisa, desejará dividi-lo em dois ou mais casos de uso. Considere o caso de uso a seguir:

Os usuários convidados podem adicionar itens em um carrinho de compras e especificar a quantidade.

Você deve dividir esse caso de uso em dois:

- Os usuários convidados podem adicionar itens em um carrinho de compras.
- Os usuários convidados podem especificar a quantidade de um item.

Você pode fazer a divisão de casos de uso, devido à maneira como eles se relacionam entre si. Os casos de uso são muito parecidos com as classes. Um caso de uso pode conter outro. Assim, se uma instância de caso de uso exige que outra faça seu trabalho, ela pode usá-la.

Um caso de uso também pode estender o comportamento de outro caso de uso. Como resultado, você pode colocar comportamento comum em um caso de uso e, então, desenvolver outros casos de uso que sejam especializações do original. Pegue o exemplo “os usuários registrados podem fazer uma compra”. Um caso de uso pode especializar o pedido, criando um caso de uso pedido para presente. Um pedido para presente poderia ser entregue sem recibo.

### Combinando casos de uso

Você não quer casos de uso redundantes. Um modo de evitar a redundância é ficar atento às variantes do caso de uso. Quando você as encontrar, deverá combinar as variantes em um único caso de uso.

#### Novo Término

Uma *variante* de caso de uso é uma versão especializada de outro caso de uso mais geral.

Considere os dois casos de uso a seguir:

- Os usuários convidados podem pesquisar o catálogo de produtos.
- Os usuários convidados podem procurar um item específico.

Aqui, o segundo é simplesmente uma variante do primeiro caso de uso mais geral.

Neste caso, o caso de uso difere apenas nos parâmetros de pesquisa. É melhor ter simplesmente um caso de uso e documentar a variante nos modelos de caso de uso que você construirá posteriormente.

Uma variante é muito parecida com uma instância de uma classe. Lembre do exemplo BankAccount. Um objeto BankAccount com um saldo de US\$10.000 pode ter mais dinheiro que um BankAccount com US\$100. Entretanto, ambos ainda são objetos BankAccount. Tudo que diferencia um objeto BankAccount de outro é o valor de seus atributos. Os casos de uso funcionam da mesma maneira.

### **Os casos de uso resultantes**

Após concluir o refinamento de seus casos de uso, você deve nomear cada caso de uso. Assim como na atribuição de nomes de atores, você deve se esforçar por nomear seus casos de uso de maneira que evite confusão.

Aqui estão os casos de uso resultantes para usuários convidados e usuários registrados, após a divisão e a combinação:

1. Navegar pelo catálogo de produtos.
2. Pesquisar o catálogo de produtos.
3. Pesquisar o site.
4. Adicionar item no carrinho de compras.
5. Ver o preço dos itens.
6. Mudar a quantidade de item.
7. Ver a lista de produtos destacada.
8. Navegar em uma lista de itens desejados.
9. Solicitar informações sobre produto.
10. Pedir.
11. Manter o pedido.
12. Adicionar itens na lista de itens desejados.
13. Atualizar a conta.
14. Assinar a correspondência.
15. Aplicar incentivos.

16. Conectar.
17. Desconectar.
18. Registrar.

Neste ponto, você tem uma lista de casos de uso bem desenvolvida. Agora, basta especificar totalmente cada caso de uso.

## Defina a seqüência de eventos de cada caso de uso

A breve lista de casos de uso só informa parte da história. Internamente, muito mais poderia estar ocorrendo dentro de um caso de uso. Pegue um pedido como exemplo. Um usuário não pode fazer um pedido em um passo. Em vez disso, ele deve usar uma seqüência de passos para concluir um pedido com êxito (como fornecer um método de pagamento).

A seqüência de passos que um usuário usa para completar um caso de uso é conhecida como *cenário*. Um caso de uso é constituído de vários cenários.

9

**Novo Termo** Um *cenário* é uma seqüência ou fluxo de eventos entre o usuário e o sistema.

Como parte de sua análise de casos de uso, você deve especificar os cenários de cada caso de uso.

Vamos desenvolver o caso de uso *Pedido*. Primeiro, comece descrevendo o caso de uso em um parágrafo:

O usuário registrado prossegue com a totalização e pagamento, para adquirir os itens de seu carrinho de compras. Uma vez na página de totalização e pagamento, o usuário fornece informações de entrega. Uma vez fornecidas, o sistema totaliza e apresenta o pedido. Se tudo estiver correto, o cliente poderá optar por continuar com o pedido. Quando o usuário continua com o pedido, o sistema consulta suas informações de pagamento. Uma vez fornecidas, o sistema autoriza o pagamento. Então, ele exibe uma página de conformação de pedido final, para os registros do usuário, e envia um e-mail de confirmação.

Existem alguns aspectos interessantes nesse caso de uso. Primeiro, ele não diz nada sobre a implementação subjacente. Segundo, você pode usá-lo para identificar as condições prévias e posteriores do caso de uso.

**Novo Termo**

*Condições prévias* são aquelas condições que devem ser satisfeitas para que um caso de uso comece. *Condições posteriores* são os resultados de um caso de uso.

**NOTA**

Um dos problemas desse tipo de sistema é que você provavelmente não está reunindo casos de uso dos usuários do sistema, mas das pessoas que querem que você os escreva. Lembre-se de que os modernos aplicativos da Web e outros aplicativos que se deparam com o cliente, como quiosques, podem exigir que você trabalhe com grupos convergentes.

Aqui, a condição prévia é que o usuário já colocou itens no carrinho. O caso de uso Pedido pede os itens do carrinho. A condição posterior é um pedido. Após completar o caso de uso, o sistema conterá um pedido para o usuário.

Nesse ponto, ajuda considerar todos os caminhos alternativos no caso de uso de pedido. Talvez a autorização de pagamento falhe ou o usuário decida cancelar o pedido, antes do término. Você precisa capturar esses caminhos alternativos.

Após se sentir à vontade com o caso de uso, você deve escrevê-lo formalmente. Um modo de escrever o caso de uso é listar os passos seqüencialmente. Após os passos, você deve listar as condições prévias, as condições posteriores e os caminhos alternativos. Considere novamente o caso de uso *Pedido*:

- Pedido
  1. O usuário registrado passa para a totalização e pagamento.
  2. O usuário registrado fornece informações de entrega.
  3. O sistema exibe o total do pedido.
  4. O usuário registrado fornece informações de pagamento.
  5. O sistema autoriza o pagamento.
  6. O sistema confirma o pedido.
  7. O sistema envia um e-mail de confirmação.
- Condições prévias
  - Um carrinho de compras não vazio.
- Condições posteriores
  - Um pedido no sistema.
- Alternativa: cancela pedido

Durante os passos 1 a 4, o usuário opta por cancelar o pedido. O usuário volta para a home page.

- Alternativa: a autorização falhou

No passo 5, o sistema falha em autorizar as informações de pagamento. O usuário pode reintroduzir as informações ou cancelar o pedido.

Você precisará completar o mesmo processo para cada caso de uso. Definir formalmente os cenários o ajuda a ver o fluxo de eventos no sistema, assim como a solidificar seu entendimento do sistema.

**DICA**

Ao escrever seus casos de uso, inclua apenas as informações que fizerem sentido. Assim como na modelagem de classe, seu objetivo é transmitir algum tipo de informação. Inclua apenas as informações para transmitir o que você está tentando fazer.

Inclua apenas as condições prévias necessárias para que o caso de uso comece. Não inclua informações extras e desnecessárias.

Certifique-se de consultar outros textos sobre casos de uso. Existem muitas maneiras de escrever um caso de uso (não há um padrão).

Ao escrever seus casos de uso pela primeira vez, considere o uso de uma ficha de arquivo com um lápis. Desse modo, você não terá de estar diante de um computador, enquanto gera seus casos de uso iniciais. Dependendo de quem forem seus clientes, pode ser difícil trabalhar com eles diante de um computador.

## Diagramas de caso de uso

9

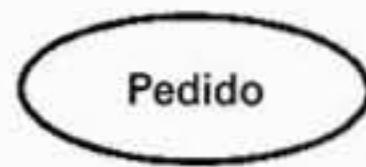
Assim como a UML fornece uma maneira de documentar e transmitir projeto de classe, também existem maneiras formais de capturar seus casos de uso. De especial interesse são os diagramas de caso de uso, diagramas de interação e diagramas de atividade. Cada um ajuda a visualizar os vários casos de uso.

Os diagramas de caso de uso modelam os relacionamentos entre casos de uso e os relacionamentos entre casos de uso e atores. Embora a descrição textual de um caso de uso possa ajudá-lo a entender um caso de uso isolado, um diagrama o ajuda a ver como os casos de uso se relacionam uns com os outros.

A Figura 9.4 mostra como modelar atores. A Figura 9.5 ilustra a notação UML para um caso de uso: uma elipse rotulada.

**FIGURA 9.5**

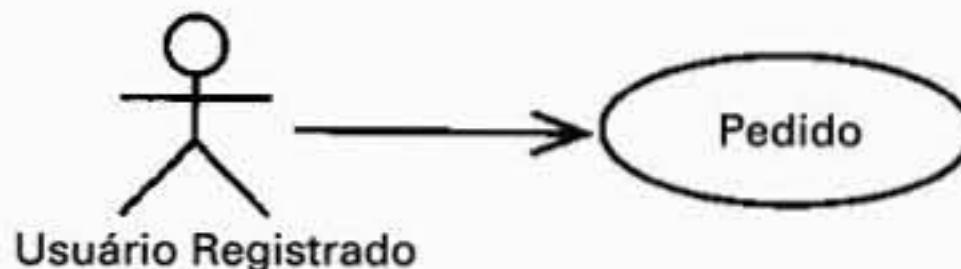
O caso de uso na UML.



Coloque um ator e um caso de uso juntos no mesmo diagrama e você terá um diagrama de caso de uso. A Figura 9.6 é o diagrama de caso de uso Pedido.

**FIGURA 9.6**

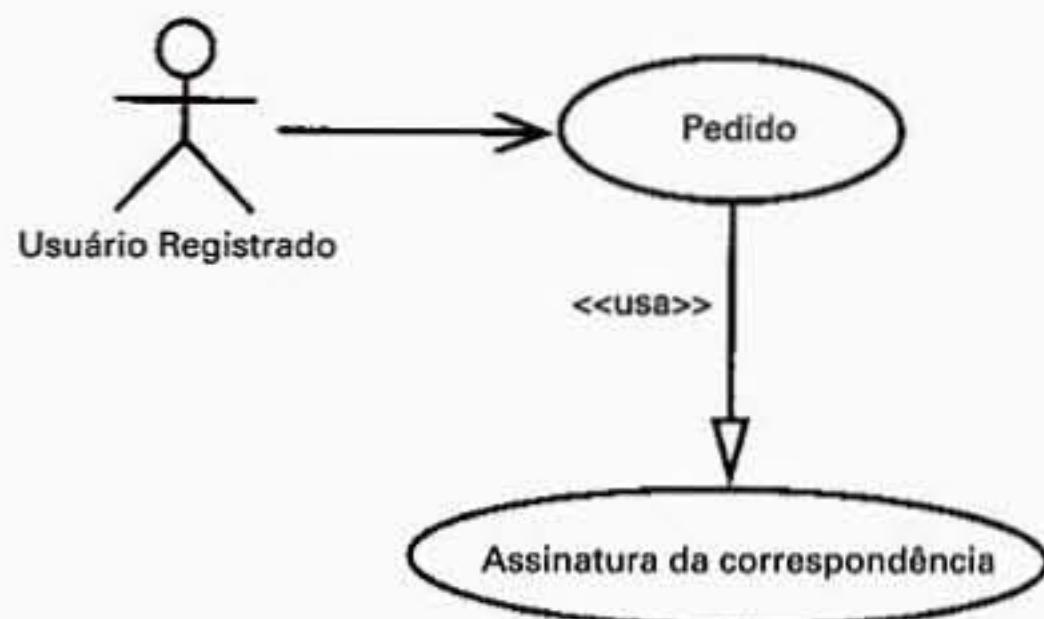
O caso de uso Pedido.



Esse diagrama é muito simples; entretanto, examinando-o, você pode ver que o usuário registrado executa o caso de uso *Pedido*.

Os diagramas podem ser um pouco mais complicados. O diagrama também pode mostrar os relacionamentos existentes entre os próprios casos de uso. Conforme você já leu, um caso de uso pode conter e usar outro. A Figura 9.7 ilustra tal relacionamento.

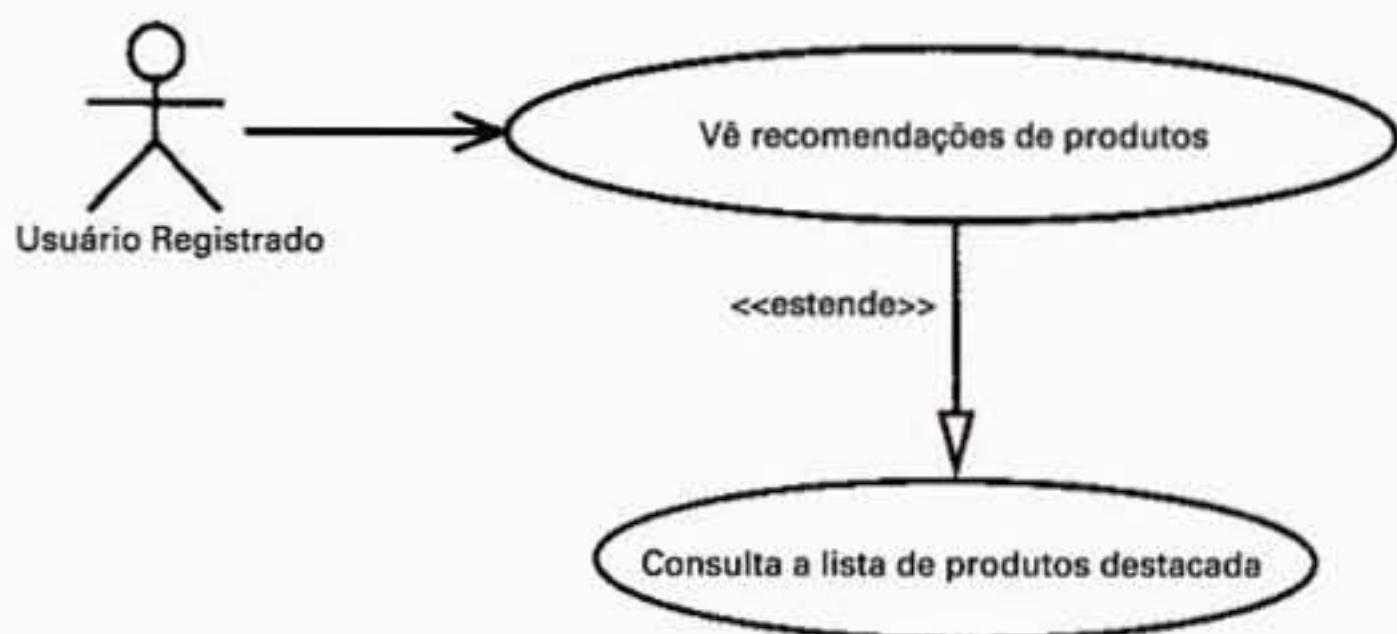
**FIGURA 9.7**  
*Um relacionamento usa.*



Aqui, você vê que o caso de uso Registro usa o caso de uso Assinatura da correspondência. Como parte do processo de registro, o usuário pode optar por receber e-mails e notificações.

A Figura 9.8 ilustra o segundo tipo de relacionamento, o relacionamento estende.

**FIGURA 9.8**  
*Um relacionamento estende.*



*Vê recomendações de produto* estende a genérica *Consulta a lista de produtos destacada*, apresentando ao usuário registrado uma lista de produtos personalizados para suas preferências de compras. A normal *Vê recomendações de produto*, conforme vista por um usuário convidado, pode simplesmente mostrar os itens mais vendidos ou mais solicitados. Essa extensão apresenta ao usuário produtos nos quais seu perfil sugere que ele poderia estar interessado.

Assim como nas classes, é possível ter um caso de uso abstrato. Um caso de uso abstrato é um caso de uso que outros casos de uso utilizam ou estendem, mas que nunca é usado diretamente por um ator em si. As abstrações normalmente são descobertas após você ter feito sua análise de caso de uso inicial. Enquanto você estuda seus casos de uso, pode encontrar meios de extrair características comuns e colocá-las em casos de uso abstratos.

## Diagramas de interação

Os diagramas de caso de uso ajudam a modelar os relacionamentos entre casos de uso. Os diagramas de interação ajudam a capturar as interações entre os vários atores participantes do sistema.

Vamos expandir os casos de uso que vimos anteriormente. Vamos adicionar um novo ator, o representante de serviço ao cliente. Freqüentemente, um usuário registrado pode se esquecer de sua senha. O representante de serviço ao cliente está lá para ajudar o usuário a reaver o acesso à sua conta. Vamos criar um novo caso de uso, *Senha Esquecida*:

Um usuário registrado liga para o representante de serviço ao cliente e informa ao representante que perdeu sua senha. O representante de serviço ao cliente pega o nome completo do usuário e extrai as informações de conta do usuário. O representante de serviço ao cliente faz então várias perguntas ao usuário registrado, para estabelecer sua identidade. Após passar por várias interpelações, o representante de serviço ao cliente exclui a senha antiga e cria uma nova. Então, o usuário recebe a nova senha por e-mail.

9

Esse caso de uso também pode ser descrito como segue:

- Senha Esquecida
  1. O usuário registrado liga para o representante de serviço ao cliente.
  2. O usuário registrado fornece o nome completo.
  3. O representante de serviço ao cliente recupera as informações do cliente.
  4. O usuário registrado responde a várias perguntas de identificação.
  5. O representante de serviço ao cliente cria uma nova senha.
  6. O usuário recebe a nova senha por e-mail.
- Condições prévias
  - O usuário esqueceu sua senha.
- Condições posteriores
  - Uma nova senha é enviada por e-mail ao usuário.
- Alternativa: a identificação falhou

O usuário pode falhar em responder corretamente as perguntas de identificação no passo 4. Se assim for, a chamada terminará.
- Alternativa: usuário não encontrado

No passo 2, o nome fornecido pode não ser o de um usuário conhecido. Se assim for, o representante de serviço ao cliente se oferecerá para registrar o usuário chamador.

Existem dois tipos de diagramas de interação: diagramas de seqüência e diagramas de colaboração. Vamos explorar cada um deles.

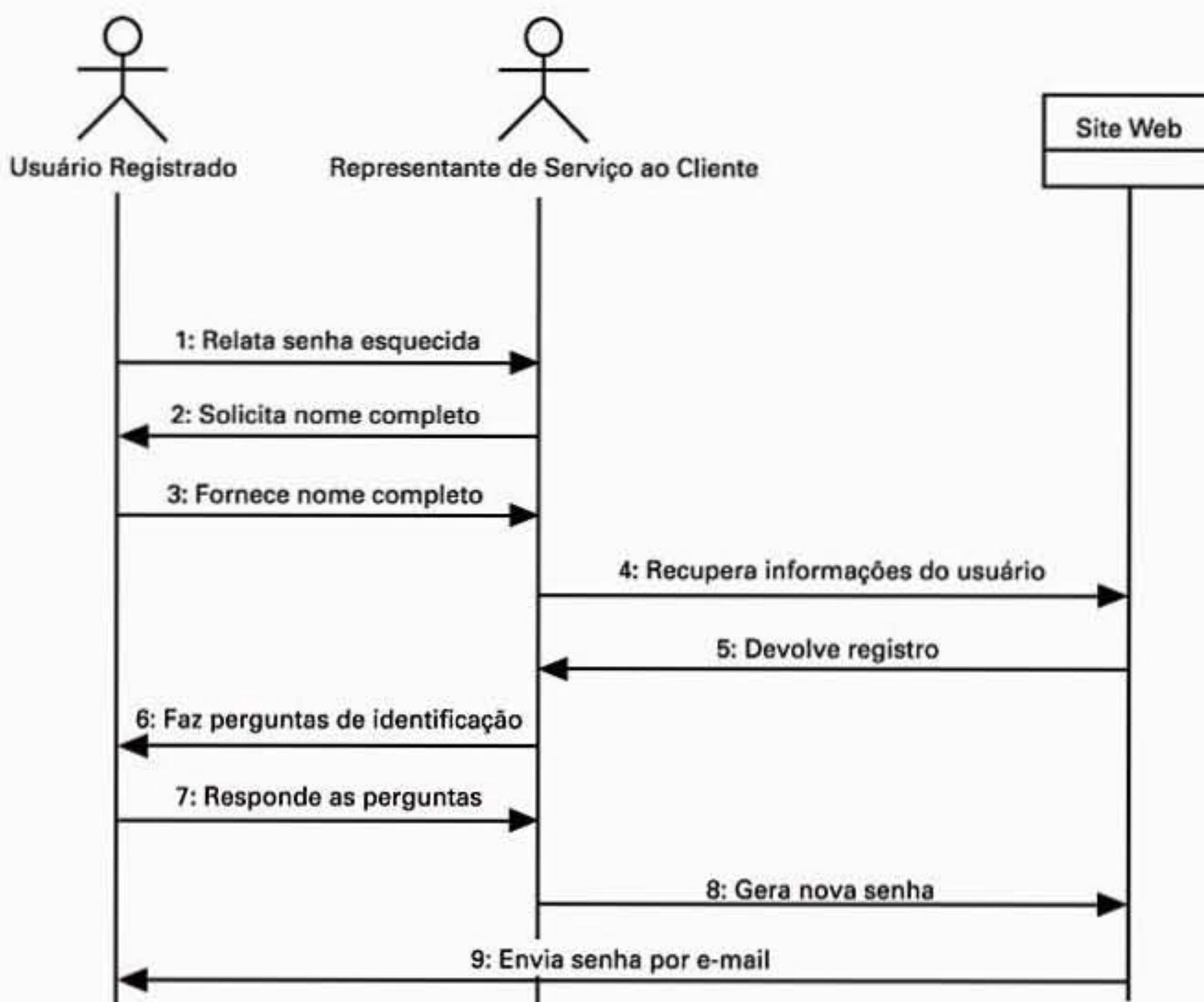
## Diagramas de seqüência

Um diagrama de seqüência modela as interações entre o usuário registrado, o representante de serviço ao cliente e o site Web, com o passar do tempo. Você deve usar diagramas de seqüência quando quiser chamar a atenção para a seqüência de eventos de um caso de uso, com o passar do tempo. A Figura 9.9 apresenta um diagrama de seqüência para o caso de uso *Senha Esquecida*.

Conforme você pode ver na ilustração, um diagrama de seqüência representa os eventos entre cada ator e o sistema (o site Web). Cada participante do caso de uso é representado no início do diagrama como uma caixa ou como um desenho de pessoa (mas você pode chamar ambos de caixa).

Uma linha tracejada, conhecida como *linha da vida*, sai de cada caixa. A linha da vida representa o tempo de vida da caixa durante o caso de uso. Assim, se um dos atores fosse embora durante o caso de uso, a linha terminaria na última seta que termina ou se origina no ator. Quando um ator deixa um caso de uso, você pode dizer que seu tempo de vida terminou.

**FIGURA 9.9**  
A diagrama de seqüência Senha Esquecida.



### Novo Término

Uma *linha da vida* é uma linha tracejada que sai de uma caixa em um diagrama de seqüência. A linha da vida representa o tempo de vida do objeto representado pela caixa.

As setas se originam na linha da vida para indicar que o ator enviou uma mensagem para outro ator ou para o sistema. Quando você desce na linha da vida, pode ver as mensagens conforme

elas se originam seqüencialmente, com o passar do tempo. O tempo corre de cima para baixo em um diagrama de seqüência. Assim, subindo na linha da vida, você pode reproduzir a seqüência de eventos de trás para frente.

### Diagramas de colaboração

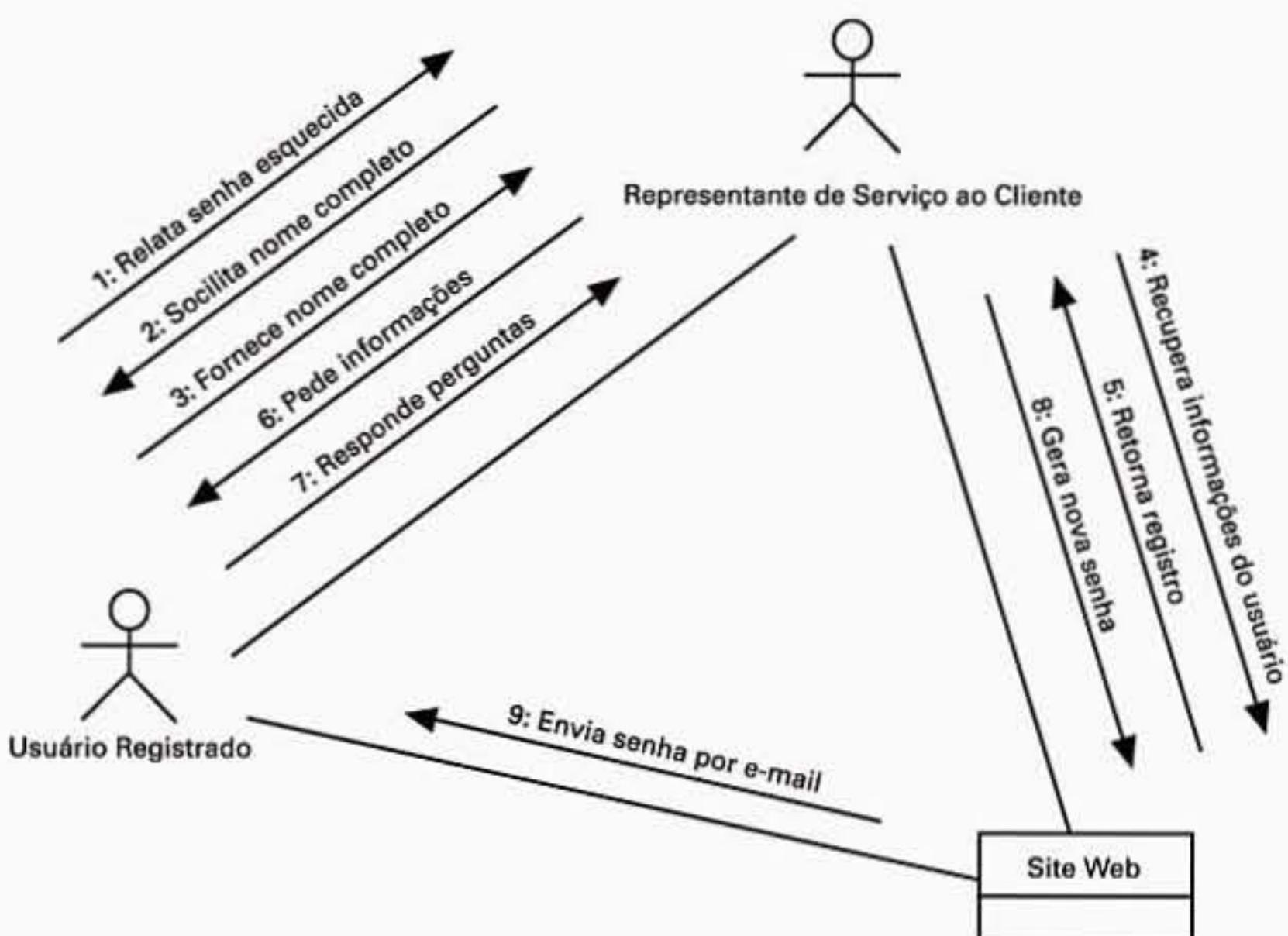
Você deve usar diagramas de seqüência se tiver a intenção de chamar a atenção para a seqüência de eventos com o passar do tempo. Se você quiser modelar os relacionamentos entre os atores e o sistema, então deve criar um diagrama de colaboração.

A Figura 9.10 modela o caso de uso *Senha Esquecida* como um diagrama de colaboração.

Em um diagrama de colaboração, você modela uma interação conectando os participantes com uma linha. Acima da linha, você rotula cada evento que as entidades geram, junto com a direção do evento (para quem ele é dirigido). Ele também ajuda a numerar os eventos, para que você saiba em qual ordem eles aparecem.

9

**FIGURA 9.10**  
O diagrama de  
colaboração  
*Senha Esquecida*.



DICA

Use diagramas de seqüência para modelar a seqüência de eventos em um cenário, com o passar do tempo.

Use diagramas de colaboração para modelar os relacionamentos entre os atores em um cenário.

## Diagramas de atividade

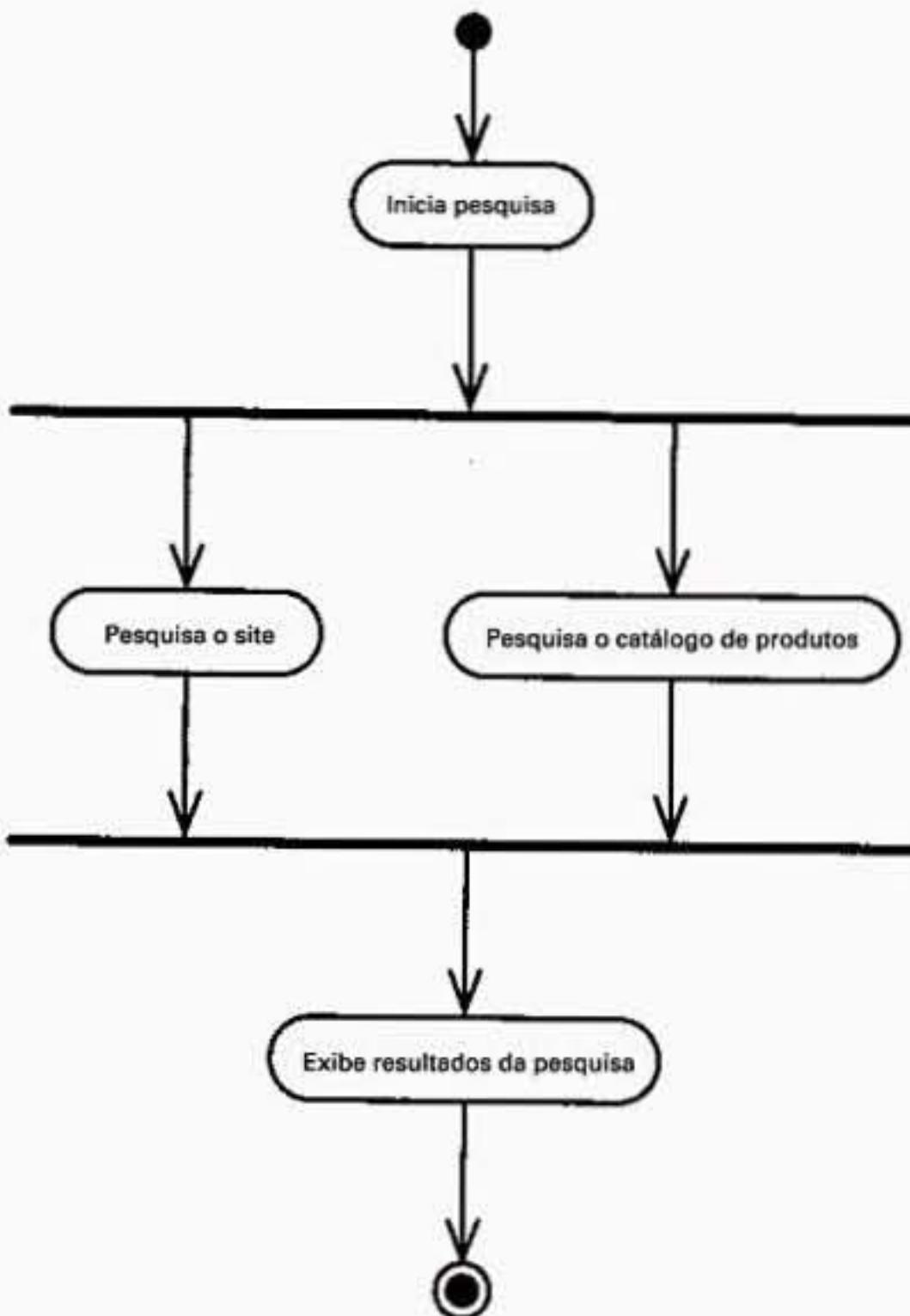
Os diagramas de interação modelam bem as ações seqüenciais. Entretanto, eles não podem modelar processos que podem ser executados em paralelo. Os diagramas de atividade o ajudam a modelar processos que podem ser executados em paralelo.

Considere outro caso de uso *Pesquisa*. Esse caso de uso pesquisa o site Web e o catálogo de produtos simultaneamente, usando o caso de uso *Pesquisa o catálogo de produtos* e o caso de uso *Pesquisa o site*. Não há motivo pelo qual essas duas pesquisas não possam ser executadas simultaneamente. O usuário ficaria impaciente se tivesse de esperar que todas as pesquisas terminassem seqüencialmente.

A Figura 9.11 modela esses processos através de um diagrama de atividade.

Uma elipse representa cada estado do processo. A barra preta grossa representa um ponto onde os processos devem ser sincronizados — ou reunidos —, antes que o fluxo de execução possa ser retomado. Aqui, você vê que as duas pesquisas são executadas em paralelo e depois reunidas, antes que o site possa exibir os resultados.

**FIGURA 9.11**  
*O diagrama de atividade Pesquisa.*



Vamos ver de perto os diagramas de interação e os diagramas de atividade nos próximos dias. Entretanto, ambos se mostram úteis ao se analisar um sistema.

## Construindo o modelo de domínio

Através da análise de caso de uso, você captura as interações do sistema. Entretanto, os casos de uso também o ajudam a capturar o vocabulário do sistema. Esse vocabulário constitui o domínio do problema. O vocabulário do domínio identifica os principais objetos do sistema.

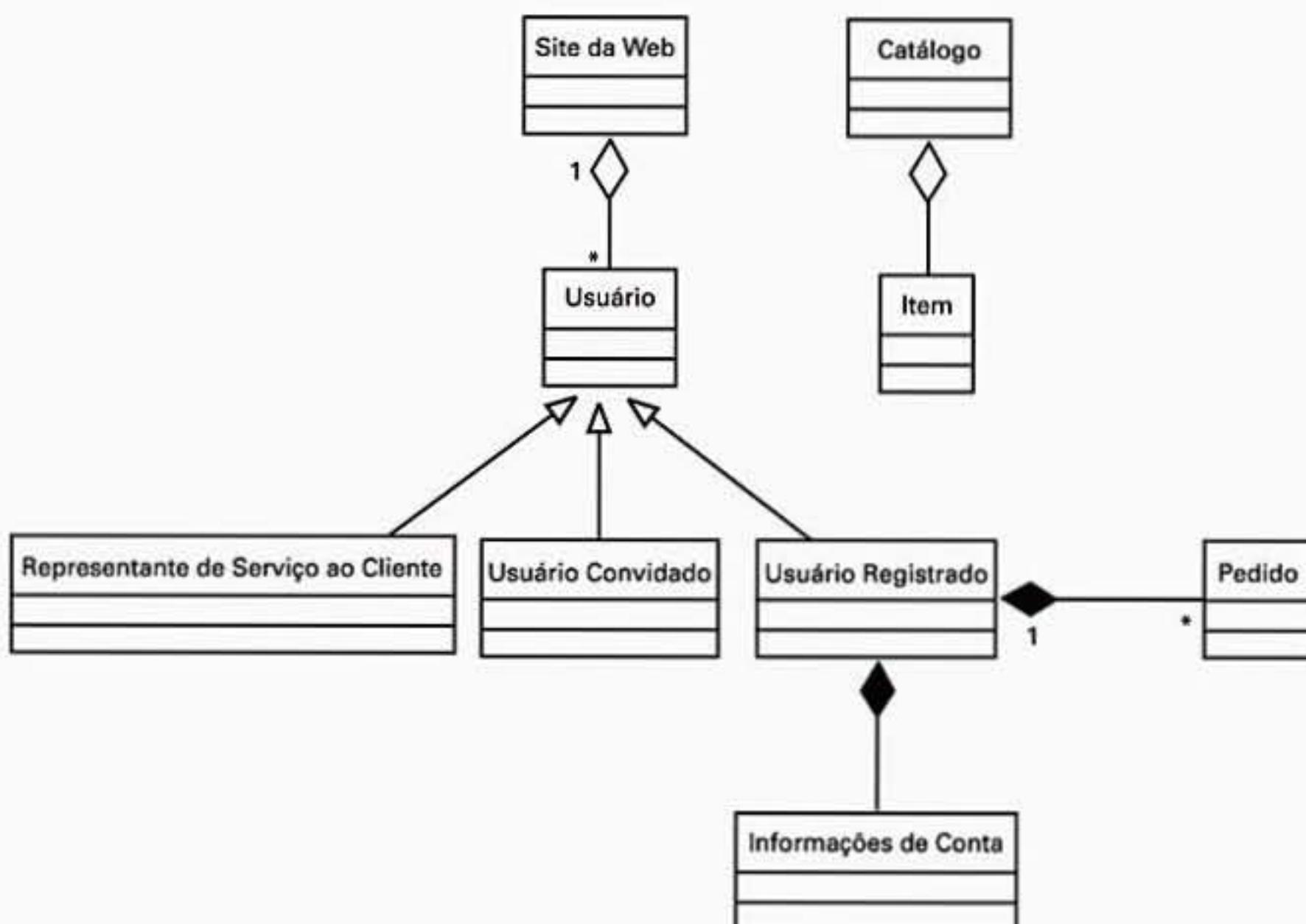
O modelo de domínio lista os objetos que você precisa para modelar corretamente o sistema. Pege a loja on-line. Através dos casos de uso, você pode identificar muitos objetos. A Figura 9.12 visualiza alguns desses objetos.

Neste ponto, você pode modelar os relacionamentos entre os objetos do domínio. A Figura 9.13 resume alguns desses relacionamentos.

**FIGURA 9.12**  
*Os objetos do domínio.*



**FIGURA 9.13**  
*Os relacionamentos dos objetos do domínio.*



O modelo de domínio é importante por vários motivos. Primeiro, o modelo de domínio modela seu problema independentemente de quaisquer preocupações com implementação. Em vez disso, ele modela o sistema em um nível conceitual. Essa independência proporciona a flexibilidade para usar o modelo de domínio que você constrói para resolver muitos problemas diferentes dentro do domínio.

Segundo, o modelo de domínio constrói a base do modelo de objeto que, finalmente, se tornará seu sistema. A implementação final pode adicionar novas classes e remover outras. Entretanto, o domínio fornece algo para que você comece e construa seu projeto — um esqueleto.

Finalmente, um modelo de domínio bem definido estabelece claramente um vocabulário comum para seu problema. Encontrando-se um vocabulário comum, todos os envolvidos no projeto poderão encará-lo a partir de uma posição e um entendimento iguais.

## E agora?

Você reuniu casos de uso. Você criou diagramas de interação. Você até iniciou um modelo de domínio. O que fazer em seguida?

Os casos de uso têm três utilizações principais. O primeiro uso trata da funcionalidade. Os casos de uso informam a você como o sistema funcionará. Os casos de uso informam quem usará o sistema, o que esses usuários farão com ele e o que esperam receber do sistema. A análise de caso de uso o ajuda a aprender a respeito do sistema que você pretende construir.

Segundo, os casos de uso fornecem uma lista de tarefas ‘a fazer’, à medida que você desenvolve o sistema. Você pode começar priorizando cada caso de uso e fornecendo uma estimativa do tempo que cada um demorará para terminar. Em seguida, você pode planejar o tempo de seu desenvolvimento em torno dos casos de uso. A conclusão de um caso de uso pode se tornar um marco. Os casos de uso também se tornam itens de barganha. Freqüentemente, as restrições de tempo o obrigarão a sacrificar um caso de uso por outro.

Finalmente, os casos de uso o ajudam a construir seu modelo de domínio. O modelo de domínio servirá como o esqueleto de seu novo sistema. (E se você o tiver feito corretamente, poderá reutilizar esse modelo em qualquer lugar!)

Quando você perceber que o modelo de domínio e a análise de caso de uso estão quase terminados, pode começar a fazer o protótipo das diferentes partes do sistema. Entretanto, não faça o protótipo de tudo. Você só deve fazer o protótipo dos aspectos do sistema que parecem confusos ou arriscados. Fazer o protótipo pode aprofundar o conhecimento, assim como descobrir se uma idéia é possível, identificando e reduzindo os riscos.



#### Dicas para a AOO eficaz

- Evite a paralisia da análise. A paralisia da análise ocorre quando você tenta realizar a análise perfeita. Você nunca vai adiante, pois fica tentando entender perfeitamente o problema. Às vezes, o entendimento total não é possível sem algum projeto e implementação.
- Faça iteração. Faça iteração de tudo. Quando você começar a análise, gere uma lista preliminar de casos de uso. Priorize os casos de uso e, em seguida, apresente-os através de iterações. Cada iteração deve abranger certa quantidade de análise, projeto e implementação. A quantidade de implementação aumentará à medida que o projeto prosseguir: muito pouco no início, muito mais durante os estágios posteriores.
- Inclua os especialistas no domínio em sua análise — mesmo que o especialista seja um cliente. A não ser que você seja especialista no domínio, precisará da entrada para modelar o sistema corretamente.
- Não introduza implementação em sua análise. Não deixe a implementação entrar furtivamente em sua análise.

9

## Resumo

A análise orientada a objetos aplica objetos ao processo de análise do problema. A AOO o ajuda a descobrir os requisitos do sistema que você pretende construir.

Os casos de uso o ajudam a identificar como os usuários vão interagir com o sistema. Os casos de uso descrevem a interação, assim como o que os usuários esperam receber do sistema.

Modelos como os diagramas de interação e os diagramas de atividade ajudam a visualizar essas interações. Cada tipo de modelo vê o sistema de um ponto de vista ligeiramente diferente. Assim, você precisará lembrar das diferenças e usar cada tipo quando for apropriado.

Os casos de uso o ajudam a definir seu modelo de domínio. O modelo de domínio serve como esqueleto do sistema que você finalmente construirá. O modelo de domínio tem a vantagem de ser livre de qualquer implementação ou uso específico. Como resultado, você pode aplicar seu modelo de domínio em muitos problemas diferentes.

É importante perceber que a AOO é realmente um modo orientado a objetos de ver um problema. Os casos de uso nada mais são do que objetos. Um caso de uso pode se relacionar com outros casos de uso, através do uso ou da generalização. As variantes dos casos de uso não são diferentes da diferença entre instâncias de classe. Os atores são objetos também.

A AOO decompõe um problema em vários casos de uso e objetos de domínio. Uma vez dividido, você pode fazer iterações até obter a solução final.

## Perguntas e respostas

**P O que acontece se você esquece um caso de uso?**

**R** Se você verificar que se esqueceu de um caso de uso, volte e acrescente-o. Se você precisar do caso de uso imediatamente, então deve acrescentá-lo imediatamente. Se ele puder esperar, tome nota e explore-o durante a próxima iteração.

**P Ao elaborar um caso de uso, você sempre precisa fazer diagramas de seqüência, colaboração e atividade?**

**R** Não. Nem sempre você precisa fazer todos os três. Faça o que for necessário para ajudar seu entendimento do caso de uso.

Entretanto, você provavelmente deve pelo menos esboçar um dos diagramas. Você nunca sabe quais problemas ou incógnitas poderia descobrir.

Normalmente, sempre fazemos pelo menos o diagrama de seqüência, a não ser que faça mais sentido começar com um dos outros.

**P Como você sabe quando tem casos de uso suficientes?**

**R** Na verdade, você nunca sabe se encontrou todos os casos de uso. Saber quando param vem através da experiência. Entretanto, você provavelmente tem casos de uso suficientes quando percebe que possui um entendimento adequado do problema e que se sente confiante de que pode prosseguir.

Se você omitir um caso de uso, sempre pode voltar e acrescentá-lo em sua análise. Entretanto, você deve tomar cuidado com a análise em demasia de um problema. Não sucumba à paralisia da análise.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. O que é um processo de software?
2. O que é um processo iterativo?
3. No final da AOO, o que você deve ter feito?
4. O que os requisitos do sistema informam a você?
5. O que é caso de uso?
6. Quais passos você deve dar para definir seus casos de uso?

7. O que é um ator?
8. Quais são algumas perguntas que você pode fazer para descobrir os atores?
9. Como os casos de uso podem se relacionar entre si?
10. O que é uma variante de caso de uso?
11. O que é um cenário?
12. Quais são algumas maneiras pelas quais você pode modelar seus casos de uso?
13. Descreva as diferenças entre os vários modelos usados para visualizar casos de uso.
14. Para que serve um modelo de domínio?
15. Para que servem os casos de uso?

9

## Exercícios

1. Quais outros casos de uso você poderia acrescentar na lista de casos de uso da loja on-line?
2. Pegue um dos casos de uso da questão 1 e desenvolva-o.
3. Uma variante de caso de uso é um caso específico de um caso de uso mais geral. Quais variantes você consegue identificar nos casos de uso de usuário convidado e de usuário registrado?
4. Quais outros objetos de domínio você consegue encontrar?

PÁGINA EM BRANCO

# SEMANA 2

## DIA 10

### Introdução ao POO (Projeto Orientado a Objetos)

Ontem, você viu como a AOO (Análise Orientada a Objetos) o ajuda a entender um problema e seus requisitos. Através da análise de casos de uso e da construção de um modelo de domínio, você pode capturar o ‘mundo real’ ou detalhes em nível de domínio de seu problema. Entretanto, a AOO é apenas parte da história geral do desenvolvimento.

O POO (Projeto Orientado a Objetos) o ajuda a pegar o domínio que você encontrou na AOO e a projetar uma solução. Enquanto o processo da AOO o ajudou a descobrir muitos dos objetos de domínio do problema, o POO o ajuda a descobrir e projetar os objetos que aparecerão na solução específica do problema.

Hoje você aprenderá:

- Como transformar sua análise em uma solução
- Como identificar e projetar os objetos que aparecerão em sua solução
- Como os cartões CRC ( Classe Responsabilidade Colaboração) podem ajudá-lo a descobrir responsabilidades e relacionamentos de objetos
- Como você pode usar a UML para capturar seu projeto

## POO (Projeto Orientado a Objetos)

**Novo Termo**

POO é o processo de construir o modelo de objeto de uma solução. Dito de outra maneira, POO é o processo de dividir uma solução em vários objetos constituintes.

**Novo Termo**

O *modelo de objeto* é o projeto dos objetos que aparecem na solução de um problema. O modelo final de objeto pode conter muitos objetos não encontrados no domínio. O modelo de objeto descreverá as várias responsabilidades, relacionamentos e estrutura do objeto.

O processo de POO o ajuda a descobrir como você vai implementar a análise que completou durante a AOO. Principalmente, o modelo de objeto que conterá as classes principais do projeto, suas responsabilidades e uma definição de como elas vão interagir e obter suas informações.

Pense no POO em termos da construção de uma casa para uma família. Antes de construir a casa de seus sonhos, você decide quais tipos de ambientes deseja em sua casa. Através de sua análise, você pode achar que deseja uma casa que tenha uma cozinha, dois quartos, dois banheiros e um lavabo, uma sala de estar e uma sala de jantar. Você também pode precisar de um gabinete de leitura, uma garagem para dois carros e uma piscina. Todos os ambientes e a piscina compreendem a idéia e o projeto de sua casa.

O que você faz em seguida? Um construtor simplesmente começa a construir? Não. Primeiro, um arquiteto descobre como os ambientes melhor se encaixam, como a fiação deve ser passada e quais vigas são necessárias para manter a casa de pé. Então, o arquiteto prepara um conjunto de cópias heliográficas detalhadas, que capturam o projeto. O construtor usa essas cópias heliográficas como guia, enquanto constrói a casa.

Usando os termos do mundo da construção, você usa o POO para criar as cópias heliográficas de seu programa.

Um processo de projeto formal o ajuda a determinar quais objetos aparecerão em seu programa e como eles vão interagir ou se encaixar. O projeto indicará a estrutura de seus objetos e um processo de projeto o ajudará a descobrir muitos dos problemas de projeto que você encontrará ao codificar.

Ao se trabalhar como parte de uma equipe, é importante identificar e resolver o máximo de problemas de projeto possível, antes de iniciar a construção. Resolvendo-se os problemas antecipadamente, todo mundo trabalhará sob o mesmo conjunto de suposições. Uma estratégia consistente para o projeto tornará mais fácil reunir todos os componentes posteriormente. Sem um projeto claro, cada desenvolvedor fará seu próprio conjunto de suposições, freqüentemente incompatível com outros conjuntos de suposições. O trabalho será duplicado e a divisão de responsabilidades entre os objetos será desfeita. De modo geral, o projeto que emerge poderia se tornar facilmente uma confusão, se não estiver todo mundo na mesma página.

Além disso, erros são exponencialmente mais dispendiosos para corrigir, quanto mais tarde na seqüência de desenvolvimento eles forem descobertos. Os erros no projeto têm um custo muito baixo para corrigir.

Enquanto projetar sua solução, você verá que freqüentemente existe mais de uma solução para o problema. O POO permite que você explore cada solução interessante e decida antecipadamente qual caminho deve seguir. Através do projeto, você pode tomar decisões acertadas, e através da documentação, você pode documentar o motivo pelo qual optou pelas escolhas que fez.

O modelo de objeto identifica os objetos significativos que aparecem na solução; entretanto, o modelo de objeto é um superconjunto do domínio. Embora muitos dos objetos que aparecem no modelo de domínio encontrem seu lugar no projeto, muitos objetos não encontrados no modelo de domínio também aparecerão. Do mesmo modo, objetos não encontrados na análise podem achar seu lugar no projeto, exatamente como a fiação não aparece na análise inicial de uma nova casa. Uma vez tendo seu projeto, você pode começar a codificar.

Dito isso, não leve o projeto ao extremo. Exatamente como a AOO pode sofrer de *paralisia da análise*, o POO pode sofrer de *paralisia do projeto*.

Você deve evitar o projeto demasiado de sua solução. Simplesmente não é possível prever cada decisão de projeto que você precisará tomar, antes de tomá-la, e alguma parte do projeto pode ser deixada para o momento da construção. Você não quer ser pego tentando criar o projeto perfeito; você precisa começar a codificar em algum momento. O que você precisa fazer é projetar os aspectos arquitetonicamente significativos do sistema.

Como você sabe quais aspectos de seu sistema são *arquitetonicamente significativos*? As partes significativas são os aspectos do sistema onde uma decisão diferente alteraria completamente a estrutura ou o comportamento do sistema.

Lembre da loja on-line e do carrinho de compras discutidos ontem. Sabendo que tem um objeto carrinho de compras em sua análise, você precisa projetar quais abstrações específicas serão usadas para representar o carrinho de compras — como um mostrador de preço, uma loja persistente e um monitor de expiração. Mas você não precisa projetar uma tabela hashing ou vetor que será usado para representar o conteúdo do carrinho — esse é um assunto apropriado para uma fase de projeto detalhado (implementação), mas é detalhado demais para esta fase.

10

## Como você aplica POO (Projeto Orientado a Objeto)?

O POO é um processo iterativo que identifica os objetos e suas responsabilidades em seu sistema, e como esses objetos se relacionam. Você refina continuamente o modelo de objetos, quando faz a iteração pelo processo de projeto. Cada iteração deve dar uma idéia mais aprofundada do projeto e talvez até do próprio domínio.

**NOTA**

Quando você aprender mais sobre o problema que está tentando resolver durante o projeto, talvez precise aplicar mais análise. Lembre-se de que não há vergonha em voltar e refinar sua análise! A única vergonha está em criar software inútil.

Existem vários passos pouco definidos que você pode seguir para construir seu modelo de objetos. Normalmente, você:

1. Gerará uma lista inicial de objetos.
2. Refinará as responsabilidades de seus objetos.
3. Desenvolverá os pontos de interação.
4. Detalhará os relacionamentos entre objetos.
5. Construirá seu modelo.

Seu entendimento do projeto aumentará quando você completar esses passos e repetir o processo.

**NOTA**

Existem muitas maneiras de completar o POO. Os passos delineados anteriormente constituem um processo informal que combina aspectos de muitas metodologias diferentes.

A metodologia que você vai seguir dependerá da experiência, do domínio, da postura da empresa e do bom gosto. No final do POO, você deve ter decomposto a solução em vários objetos. Como você chega a esses objetos fica por sua conta e da sua equipe de projeto.

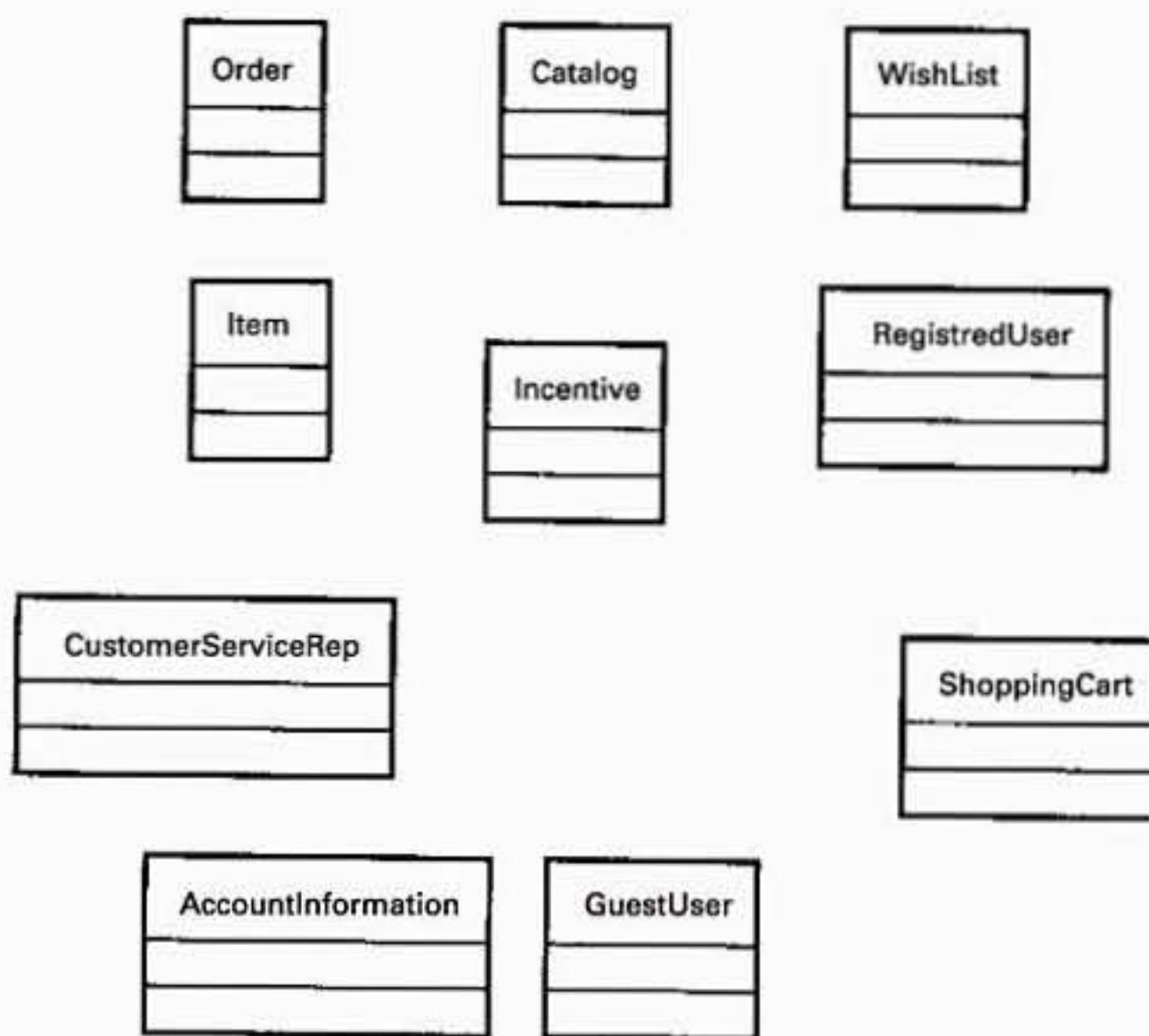
## **Passo 1: gere uma lista inicial de objetos**

Quando começa a projetar seu sistema, você precisa começar com o domínio que definiu durante a análise. Cada objeto do domínio e cada ator deve se tornar uma classe em seu novo modelo de objetos. Você verá que alguns dos objetos de domínio não terão lugar em seu modelo de objetos final; entretanto, neste ponto, você não pode ter certeza de qual terá; portanto, você precisa incluir todos eles.

Lembre da loja on-line do Dia 9, “Introdução à AOO (Análise Orientada a Objetos)”. A Figura 10.1 ilustra as classes básicas que aparecerão em seu modelo de objetos inicial.

Ao procurar a lista de classes inicial, você também desejará considerar todos os eventos que possam afetar seu sistema. Cada um desses eventos deve aparecer inicialmente como uma classe. O mesmo pode ser dito para todos os relatórios, telas e dispositivos. Todos esses elementos devem ser transformados em uma classe.

**FIGURA 10.1**  
As classes básicas da loja on-line.



10

**DICA**

Aqui estão algumas dicas para animar essa lista de classes inicial:

- Transforme cada ator em uma classe.
- Transforme cada objeto de domínio em uma classe.
- Transforme cada evento em uma classe.
- Considere como o sistema apresentará informações e transforme cada tela em um objeto.
- Represente todos os outros sistemas ou dispositivos com os quais o sistema interage como classes.

Neste ponto, você não pode dizer muito a respeito das classes que listou. Você pode ter uma idéia geral das responsabilidades e relacionamentos dos objetos; entretanto, você precisa se aprofundar um pouco mais, antes de poder finalizar seu entendimento das novas classes.

## Passo 2: refine as responsabilidades de seus objetos

Uma lista de objetos é um bom ponto de partida, mas é apenas uma pequena parte de seu projeto global. Um projeto completo capturará as responsabilidades de cada objeto, assim como a estrutura e os relacionamentos do objeto. Um projeto mostrará como tudo se encaixa.

Para ter esse entendimento, você precisa identificar o que cada objeto faz. Existem dois aspectos que você precisa explorar para que possa responder à pergunta, “o que o objeto faz?”

Primeiro, você precisa explorar a responsabilidade. Através do encapsulamento, você sabe que cada objeto deve ter um número pequeno de responsabilidades. Durante o projeto, você precisa identificar as responsabilidades de cada objeto e dividir o objeto, quando ele começar a

fazer coisas demais. Você também precisa certificar-se de que cada responsabilidade apareça apenas uma vez e esse conhecimento é espalhado igualmente entre todos os objetos.

Em seguida, você precisa explorar o modo como cada objeto faz seu trabalho. Os objetos frequentemente delegarão trabalho para outros objetos. Através de seu projeto, você precisa identificar essas colaborações.

**Novo TERMO** Um *objeto* delega trabalho para *colaboradores*.

**Novo TERMO** *Colaboração* é o relacionamento onde os objetos interagem para realizar o mesmo propósito.

Um entendimento profundo dos relacionamentos e responsabilidades de um objeto é importante. Em um nível prático, as responsabilidades serão transformadas em métodos. Os relacionamentos serão transformados em estruturas; entretanto, um entendimento global da responsabilidade o ajudará a dividir a responsabilidade eficientemente entre os objetos. Você precisa evitar ter um pequeno conjunto de objetos grandes. Através do projeto, você terá a certeza de dividir as responsabilidades.

## O que são cartões CRC?

Uma maneira de distribuir responsabilidades e colaborações é através do uso de cartões CRC (Classe Responsabilidade Colaboração). Conforme o nome sugere, um cartão CRC nada mais é do que uma ficha de arquivo 4x6 com linhas.

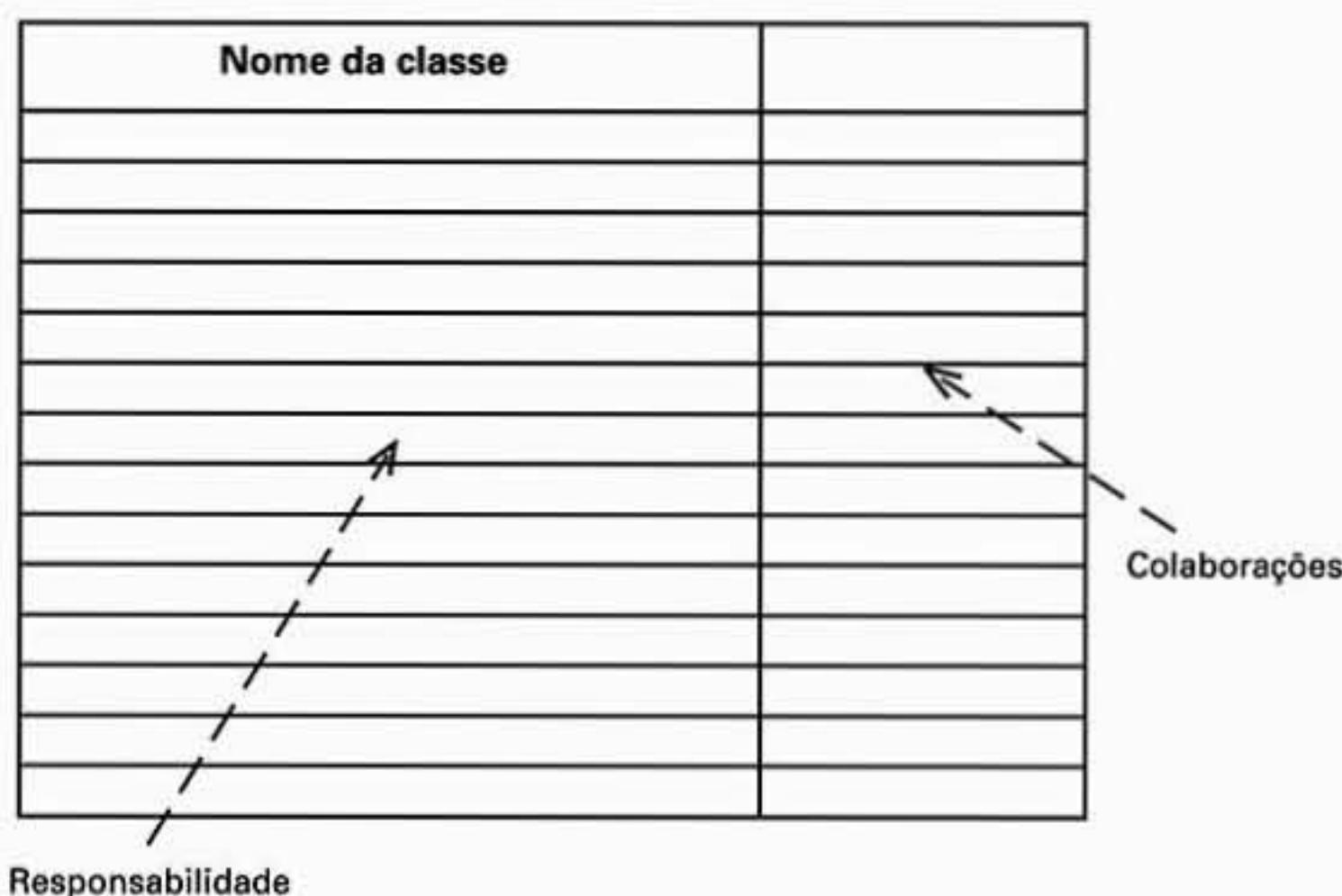
Quando você começa a projetar pela primeira vez, é difícil simplesmente começar a listar métodos e atributos. Em vez disso, você precisa começar a identificar o propósito de cada objeto.

Os cartões CRC ajudam a definir o propósito de um objeto, chamando a atenção para as responsabilidades do objeto. Quando usa cartões CRC, você simplesmente cria um cartão para cada classe. Você escreve o nome da classe no início do cartão e, em seguida, divide o cartão em duas seções. Liste as responsabilidades no lado esquerdo e, no lado direito, liste todos os outros objetos que o cartão precisa para executar suas responsabilidades.

A Figura 10.2 ilustra um modelo de cartão CRC.

Os cartões CRC são intencionalmente de baixa tecnologia. Você é intencionalmente limitado pelo tamanho do cartão. Se você achar que o cartão não é grande o suficiente, são boas as chances de que é preciso dividir a classe. Uma grande vantagem dos cartões CRC é que você não fica preso a um computador. Acredite se quiser, ter de projetar diante de um computador nem sempre é desejável. O projeto não é um exercício solitário e pode exigir conversas e discussões entre os projetistas. Os cartões CRC liberam você e seus colegas projetistas para projetar quando e onde quiserem. Projete no almoço; vá até uma sala de conferência ou para um banco de parque. Desde que possa usar seus cartões, você pode projetar.

**FIGURA 10.2**  
Um modelo de cartão CRC.



Os cartões CRC também o liberam de ter de manter um modelo eletrônico atualizado. Em breve seu modelo mudará freqüentemente e ter de manter o modelo atualizado pode ser uma grande inconveniência. Em vez disso, você simplesmente pega seus cartões e apaga, rabisca, dobra, escreve e os rasga, quando necessário. O custo de mudar um cartão é muito mais barato (e mais imediato) do que ter de atualizar algo em seu computador. Ter de parar a sessão de CRC e atualizar um computador pode interromper o fluxo da sessão.

Finalmente, os cartões CRC têm apelo em nossos instintos mais primitivos. Eles apenas ajudam a ter algo concreto que você possa pegar. Em vez de ter de modelar vários projetos alternativos, você pode apenas movimentar seus cartões. Para modelar interações, você pode sentar-se com outros projetistas, dividir os cartões e percorrer as interações. Tal processo interativo estimula a discussão.

## Como você aplica cartões CRC?

A criação de cartões CRC não é um exercício solitário. Em vez disso, você deve manter sessões CRC com outros projetistas/desenvolvedores, para que o processo de projeto seja o mais interativo possível.

Você inicia uma sessão escolhendo vários casos de uso. A quantidade que você escolhe depende do tempo de que dispõe e da dificuldade dos casos de uso. Se você tiver alguns casos de uso muito grandes, pode fazer sentido ter uma sessão por caso de uso. Se você tiver muitos casos pequenos, vai querer atacar vários casos de uso durante a sessão. O importante é que você manipule casos de uso relacionados e que torne notável o progresso em seu projeto. Não permita que o processo sofra empecilhos.

Quando você escolher os casos de uso, identifique as classes principais e crie um cartão para cada uma. Quando você tiver os cartões, divida-os entre os projetistas e, em seguida, inicie a ses-

são. Durante a sessão, você investigará o cenário de cada caso de uso. À medida que você percorrer o cenário, cada projetista deverá se concentrar nas responsabilidades e colaborações de sua classe. Quando sua classe for necessária no cenário, notará seu uso e dirá a todos os outros projetistas se precisa delegar para outro objeto.

A estratégia que você adota para a sessão de projeto é questão de preferência pessoal. Algumas metodologias exigem o desempenho do papel, onde cada projetista recebe um cartão e representa o papel da classe. Outras metodologias são um pouco mais formais. A estratégia escolhida fica por sua conta e de sua equipe de projeto. O importante é que todos fiquem engajados no processo e que ninguém fique excluído. É durante essas sessões que você discutirá alternativas de projeto, descobrirá novos objetos e construirá a camaradagem na equipe.

## Um exemplo de cartão CRC

Vamos considerar o caso de uso *Pedido* do Dia 9 e ver como você poderia usar cartões CRC para atribuir responsabilidades para ele.

- Pedido
  - O usuário registrado passa para a totalização e pagamento.
  - O usuário registrado fornece informações de entrega.
  - O sistema mostra o total do pedido.
  - O usuário registrado fornece informações de pagamento.
  - O sistema autoriza o pagamento.
  - O sistema confirma o pedido.
  - O sistema envia um e-mail de confirmação.
- Condições prévias
  - Um carrinho de compras não vazio.
- Condições posteriores
  - Um pedido no sistema.
- Alternativa: cancelar pedido
  - Durante os passos 1 a 4, o usuário opta por cancelar o pedido. O usuário volta para a home page.
- Alternativa: a autorização falhou
  - No passo 5, o sistema falha em autorizar as informações de pagamento. O usuário pode introduzir novamente as informações ou cancelar o pedido.

Comece identificando as classes. Imediatamente, você verá: Usuário Registrado, Pedido, Pagamento, Confirmação de Pedido, Carrinho de Compras, Informações de Entrega. Talvez você também queira incluir Sistema; entretanto, há um problema. Nos casos de uso, tudo que não foi feito por um ator foi feito pelo sistema ‘tudo incluído’. O objetivo da análise era entender o problema. Então, em vez de tentar dividir o sistema em suas partes, a análise o tratou como uma grande caixa preta. Através do projeto, você vai decompor o sistema em suas partes.

Se um cartão de sistema vai ser incluído ou não vai depender da complexidade do sistema. Pode ajudar listar todas as responsabilidades do sistema em um cartão e, em seguida, dividir esse cartão em várias classes, posteriormente. Neste caso, entretanto, você pode tentar decompor o sistema, antes de começar. Se você se esquecer de algo, sempre pode acrescentar posteriormente.

Comece lendo os passos do cenário. Aqui, o sistema autoriza o pagamento, exibe e confirma o pedido. O sistema também cria e introduz o pedido. Você pode começar dividindo o sistema em Funcionário, Mostra Pedido e Terminal de Pagamento.

Quando você se sentir confortável com sua lista de classes, pode criar os cartões CRC, como ilustrado na Figura 10.3.

**FIGURA 10.3**  
*Alguns dos cartões CRC para Pedido.*

O próximo passo é percorrer cada passo do cenário e identificar responsabilidades. O passo 1, “o usuário registrado passa para a totalização e pagamento”, é simplesmente um clique em um link na interface. Por simplicidade, vamos ignorar a interface. O passo 2, “o usuário registrado fornece informações de entrega”, é um pouco mais interessante. Aqui, você vê que o Usuário Registrado é responsável por fornecer suas informações de entrega para o Funcionário.

A Figura 10.4 ilustra o cartão resultante.

**FIGURA 10.4**  
*O cartão CRC de Usuário Registrado.*

Usuário Registrado	
fornece informações de entrega	Informações de Entrega



Quando você ver que um objeto ‘providencie’ ou ‘forneça’ algo, precisará certificar-se de que o objeto não esteja atuando como uma simples estrutura de dados.

Se o objeto não fizer nada além de fornecer acesso às informações, você desejará combinar esse objeto com o que manipula os dados.

O passo 3, “o sistema exibe o total do pedido”, é um pouco mais complicado. Antes que o sistema possa exibir algo, o Funcionário deve introduzir o pedido, ver o preço dele e totalizar o pedido.

O Funcionário usará o Carrinho de Compras para recuperar os itens e o Mostra Pedido para exibir o pedido resultante; entretanto, o Funcionário provavelmente não deve ser responsável também por ver o preço ou por totalizar o pedido. Essas tarefas são melhores delegadas para outro objeto. Lembre-se de que um objeto só deve ter um pequeno número de responsabilidades.

O objeto `Pedido` deve cuidar de ver o preço e totalizar o pedido. A Figura 10.5 resume as responsabilidades do Funcionário até aqui.

O objeto Pedido também deve conter os itens que estão sendo adquiridos. O objeto Item foi ignorado durante a criação da lista de classes; portanto, você deve adicioná-lo agora. O objeto Item contém informações de preço e produto, assim como uma quantidade. O objeto Item também conterá todos os incentivos que o usuário possa ter aplicado. Incentivo é outra classe que você deve adicionar na lista.

Os passos restantes são muito parecidos com os outros. O Usuário Registrado fornece informações de Pagamento, o Funcionário autoriza o pagamento e finaliza o pedido.

As figuras 10.6 e 10.7 resumem as responsabilidades do Funcionário e do Usuário Registrado.

**FIGURA 10.5**  
*O cartão CRC para Funcionário.*

**FIGURA 10.6**

*O cartão CRC para Funcionário completo.*

Funcionário	
recupera informações de entrega e pagamento	Informações de Entrega
introduz o pedido	Carrinho de Compras Pedido
exibe o pedido	Mostra Pedido
autoriza o pedido	Terminal de Pagamento
confirma o pedido	Pedido

## Quantas responsabilidades por classe?

Ao desenvolver seus cartões CRC, você precisa garantir que cada classe tenha apenas duas ou três responsabilidades principais. Se você tiver mais responsabilidades, deverá dividir a classe em duas ou mais classes separadas.

10

**ALERTA**

Você deve considerar a divisão da classe, mas em sistemas reais, isso não é necessariamente realista.

**FIGURA 10.7**

*O cartão CRC para Usuário Registrado.*

Usuário Registrado	
fornecer informações de entrega	Informações de Entrega
fornecer informações de pagamento	Pagamento

Pegue a classe Funcionário, por exemplo. O Funcionário é responsável por:

- Recuperar informações de pagamento e entrega.
- Introduzir o pedido.

- Exibir o pedido.
- Autorizar o pagamento.
- Confirmar o pedido.

Conforme a Figura 10.8 ilustra, entretanto, todas essas responsabilidades caem sob a responsabilidade de ‘Processa Pedido’.

**FIGURA 10.8**  
*Processa Pedido.*

Funcionário	
recupera informações de entrega e pagamento	Usuário Registrado
introduz o pedido	Carrinho de Compras Pedido
exibe o pedido	Mostra Pedido
autoriza o pedido	Terminal de Pagamento
confirma o pedido	Pedido

É importante listar subtarefas; entretanto, você não quer fazer isso em demasia. O importante é que todas as sub-responsabilidades trabalhem para um objetivo comum. Aqui, todas as tarefas trabalham para o processamento de um pedido.

Ao trabalhar com cartões CRC, você também precisa lembrar que eles atendem um propósito específico: definir responsabilidades e relacionamentos de colaboração simples. Não use cartões CRC para descrever relacionamentos complexos.

Você também deve manter os cartões CRC com baixa tecnologia — não tente automatizar o processo de cartão CRC.

## Limitações do cartão CRC

Assim como toda boa ferramenta, os cartões CRC têm seus usos, bem como suas limitações.

Os cartões CRC foram desenvolvidos originalmente como uma ferramenta de ensino. Como resultado, eles são uma maneira excelente de encarar o projeto inicial de um sistema — especialmente se você for iniciante em OO.

Entretanto, os cartões CRC são difíceis de usar quando o projeto se torna mais complicado. Pode ser difícil controlar interações complexas entre os objetos, simplesmente através do uso de cartões CRC. Um grande número de classes também torna o uso de cartões desajeitado. Também

pode se tornar difícil continuar a usar cartões CRC, quando você começar a modelar. Manter os dois em sincronismo pode ser um desafio.

O melhor conselho é usar o que funciona. Use cartões CRC até que eles não mais auxiliem o processo de projeto. Nesse momento, simplesmente pare de usá-los.



#### Quando você deve usar cartões CRC?

- Durante os estágios iniciais do projeto.
- Quando você ainda é iniciante em OO.
- Para descobrir responsabilidades e colaborações.
- Para percorrer um cenário. Não apenas os cartões encontrará responsabilidades e colaborações, como podem ajudá-lo a entender melhor o cenário.
- Em projetos menores ou para se concentrar em uma seção menor de um projeto grande. Não tente atacar um projeto inteiro com cartões CRC.

## Passo 3: desenvolva os pontos de interação

10

Quando tiver completado seus cartões CRC para um conjunto de casos de uso, você precisará desenvolver os pontos de interação. Um ponto de interação é qualquer lugar onde um objeto use outro.

**Novo Termo** Um *ponto de interação* é qualquer lugar onde um objeto use outro.

Existem vários problemas a considerar sobre o ponto de interação.

### Interfaces

Você precisa de uma interface bem definida, onde um objeto usa outro. Você quer ter certeza de que uma alteração em uma implementação não vá danificar o outro objeto.

### Agentes

Você também precisa ver as interações de forma crítica. Pegue uma biblioteca, por exemplo. Uma biblioteca tem prateleiras e livros. Quando é hora de tirar um livro da prateleira, o livro chama um método `removeBook()` na prateleira ou é a prateleira que chama um método `removeBook()` no livro? Na verdade, a responsabilidade não pertence ao livro ou à prateleira. Nenhuma das escolhas modela adequadamente o mundo real. No mundo real, uma bibliotecária ou um cliente retirará o livro. As bibliotecárias e os clientes também podem aparecer no mundo OO; entretanto, o mundo OO chama esses atores de *agentes* ou intermediários.

**Novo Termo** Um *agente* faz a mediação entre dois ou mais objetos para atingir algum objetivo.

Ao desenvolver pontos de interação, você desejará prestar atenção em lugares para usar agentes.

## Considerações futuras

Ao considerar as interações, você desejará procurar lugares onde o objeto que está sendo usado possa mudar. Se você puder localizar tais lugares com precisão, desejará projetar a interação de tal maneira, que ela não será desfeita caso novos objetos sejam introduzidos. É para planejamento de mudança que serve software à prova do futuro. Em tais lugares, você desejará estabelecer relacionamentos com capacidade de conexão e usar polimorfismo, para que possa introduzir novos objetos a qualquer momento.

Assim como em tudo, entretanto, não faça isso em demasia. Planeje a alteração apenas em lugares onde você saiba com certeza absoluta que ocorrerá uma mudança. Existem duas maneiras gerais de prever o futuro:

- A alteração é um requisito. Se os requisitos solicitarem alterações futuras, projete a alteração.
- Se você estiver usando uma biblioteca de terceiros e quiser migrar para uma nova versão de forma transparente no futuro, planeje a alteração.

O primeiro caso é melhor resolvido pelo estabelecimento de algum tipo de relacionamento com capacidade de substituição, através de herança. O segundo requer um pouco mais de trabalho. Você desejará empacotar a biblioteca em classes que criou. Você deve tomar o cuidado de projetar esses empacotamentos para tirar proveito da capacidade de conexão e do polimorfismo.

## Transformação de dados

Durante o projeto, você pode encontrar lugares onde precisa transformar dados, antes de passá-los para outro objeto. Normalmente, você delegaria tal transformação de dados para outro objeto; se precisar alterar a transformação, você só precisará atualizar a classe de transformação. Essa prática também ajuda a dividir responsabilidades.

Ao considerar esses pontos de interação e adicionar novas classes, talvez você precise rever e atualizar seus cartões CRC ou modelo de objetos (se você tiver iniciado a modelagem formal).

## Passo 4: detalhe os relacionamentos entre os objetos

Quando você tiver estabelecido os relacionamentos de responsabilidade e colaboração básicos, precisará detalhar os relacionamentos complexos entre as classes. É aí que você define as dependências, associações e generalizações. Detalhar esses relacionamentos é um passo importante, pois isso define como os objetos se encaixam. Isso também define a estrutura interna dos vários objetos.

Comece com os cartões CRC. Embora eles não capturem cada relacionamento, eles capturam a colaboração. Eles também vão sugerir alguns outros relacionamentos.

Procure classes que compartilhem responsabilidades semelhantes. Se duas ou mais classes compartilham o mesmo conjunto de responsabilidades, são boas as chances de que você possa contar

com as características comuns em uma classe base. Você também desejará rever o passo 3 e considerar todos os relacionamentos com capacidade de substituição que poderia precisar.

## Passo 5: construa seu modelo

Quando chegar ao passo 5, você já terá modelado o sistema formalmente. Um modelo completo consistirá em diagramas de classe e diagramas de interação. Esses diagramas descreverão a estrutura e o relacionamento das várias classes do sistema. A UML também define modelos para modelar interações, transição de estado e atividades. Para os propósitos deste livro, entretanto, você vai se concentrar nos diagramas de classe e interação. Você viu os dois tipos em dias anteriores.

A Figura 10.9 modela a estrutura *Pedido*.

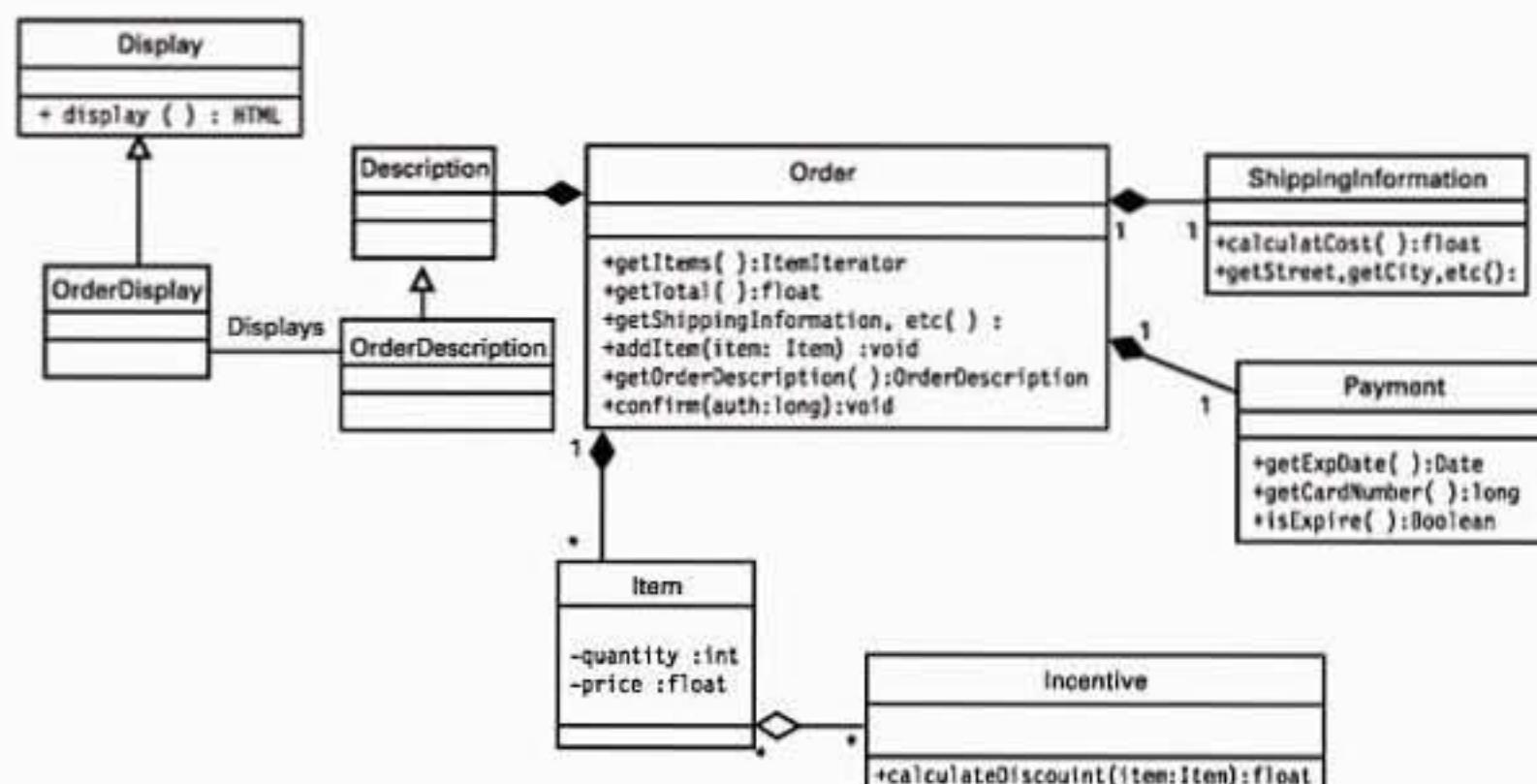
O modelo *Pedido* ilustra todos os relacionamentos importantes entre *Pedido* e as classes que exigem o pedido. Conceitualmente, você também terá modelos que ilustram os relacionamentos entre Funcionário, Pedidos e Usuário Registrado.

Novamente, você quer apenas modelar o que faz sentido — os componentes arquitetônicos interessantes. Lembre-se de que você está tentando transmitir informações específicas através de seus modelos e não simplesmente apresentar modelos por questões de documentação.

A Figura 10.10 ilustra o diagrama de seqüência atualizado para o caso de uso *Pedido*.

10

**FIGURA 10.9**  
*Modelo Pedido.*

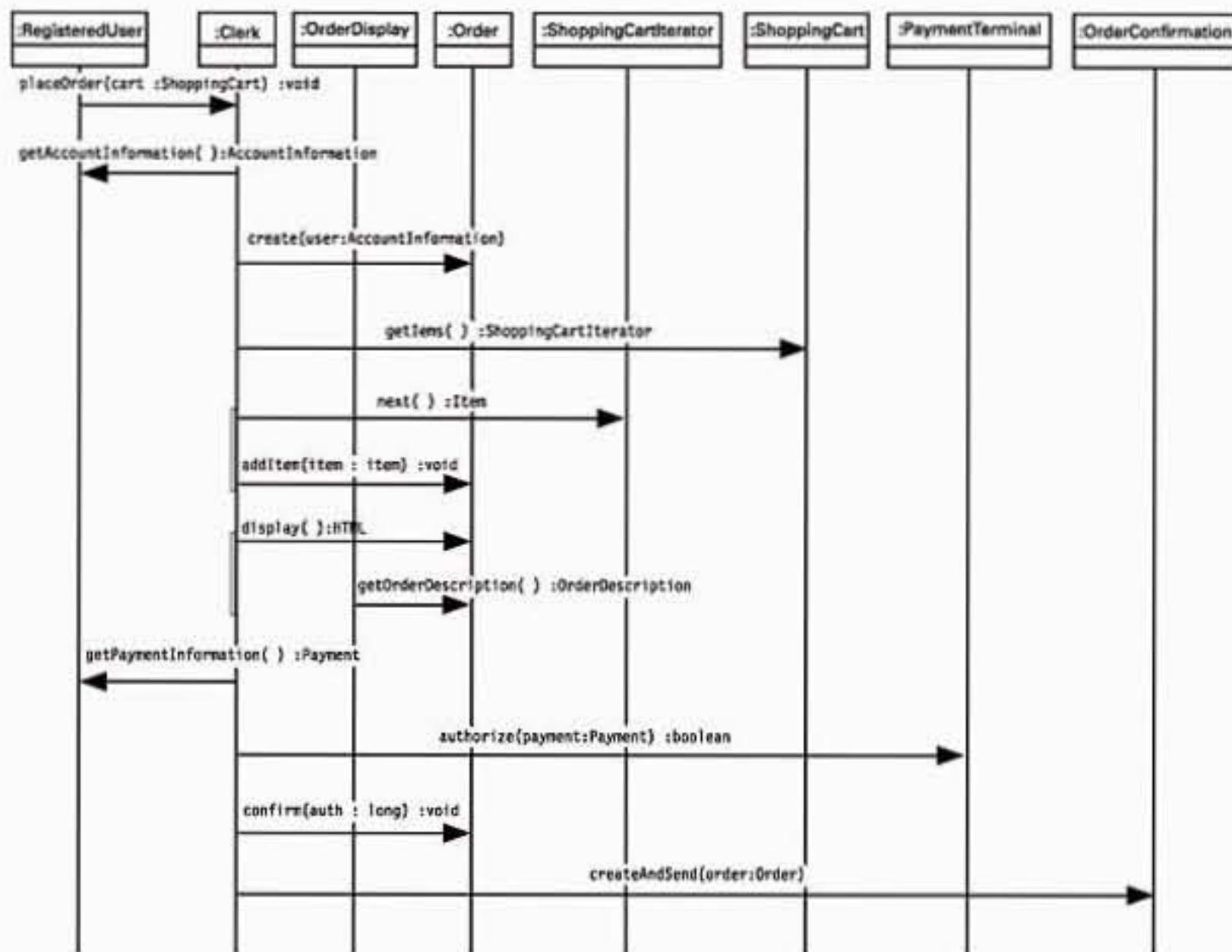


Você também pode criar diagramas de seqüência e colaboração para modelar as interações importantes do sistema.

Quando tiver terminado de criar os modelos, você deverá ter descrições de todas as principais estruturas e interações encontradas no sistema. Esses modelos dizem como os vários objetos estão estruturados, como eles se relacionam e como eles se encaixam para modelar a solução do problema elaborado durante a análise.

**FIGURA 10.10**

*Diagrama de seqüência de Pedido.*



## Resumo

O POO continua o trabalho da AOO, pegando o domínio e transformando-o em uma solução para seu problema. Através do processo de POO, você pega seu modelo de domínio e constrói o modelo de objetos de sua solução. O modelo de objetos descreve os aspectos arquitetonicamente significativos de seu sistema, como a estrutura e os relacionamentos dos objetos — como os objetos se encaixam. No final do POO, você deverá ter uma boa idéia do que implementará no código.

A grosso modo, existem cinco passos iterativos que você pode seguir, enquanto realiza o POO:

Passo 1: gerar a lista de objetos inicial.

Passo 2: refinar as responsabilidades de seus objetos através de cartões CRC.

Passo 3: desenvolver os pontos de interação.

Passo 4: detalhar os relacionamentos entre os objetos.

Passo 5: construir seu modelo.

Cada passo refina mais seu modelo de objetos e o deixa mais próximo da cópia heliográfica de seu sistema.

## Perguntas e respostas

**P Por que é importante projetar o sistema antes de começar a codificar?**

**R** Existem vários fatores que tornam o projeto importante. O projeto o ajuda a prever problemas de implementação, antes de começar a codificar. É muito mais fácil corrigir um problema em um modelo do que percorrer todo seu código para corrigi-lo.

O projeto também garante que todos os desenvolvedores de uma equipe de desenvolvimento estejam na mesma página. O projeto esclarecerá muitas das suposições arquitetônicas importantes que, de outro modo, teriam sido decididas durante o desenvolvimento. Deixando as decisões arquitetônicas para o momento da implementação, você corre o risco de ter cada desenvolvedor tomando suas próprias decisões incompatíveis.

**P Por que é importante modelar seu projeto antes de começar a codificar?**

**R** Modelar é importante pelo mesmo motivo que a maioria das pessoas redige uma palestra antes de proferi-la. Modelando o projeto, você pode ver como as partes se encaixam. Tendo uma idéia visual do projeto, você pode decidir se está confortável com ele. Você pode verificar se tudo se encaixa de forma tranquila e sensata.

Às vezes, a solução de um problema parecerá simples, quando você pensar a respeito dela. Pegar a solução e escrevê-la formalmente o obriga a ser intelectualmente honesto quanto a solução. Ter de transformar formalmente um problema o obriga a pensar de forma crítica e exata sobre a solução. O que parece simples em sua mente pode fazê-lo arrancar os cabelos, antes de você terminar!

10

**P Como você sabe quando seu projeto está concluído?**

**R** Não existem regras rígidas e rápidas que indiquem quando um projeto está terminado. Se você se preocupar em ter um projeto perfeito, poderá cair facilmente na paralisia do projeto.

Tudo se resume ao que você está tentando transmitir através de seus modelos, seu nível de experiência e o nível de experiência de sua equipe. Se você tiver uma equipe experiente, provavelmente poderá apenas modelar a arquitetura de alto nível do sistema. Se você ou os membros de sua equipe são iniciantes nos objetos, provavelmente desejarão passar mais tempo projetando. De qualquer modo, você termina quando se sente confiante o suficiente de que pode pegar o projeto e implementar a solução.

O processo de desenvolvimento OO é iterativo. Você deve usar isso para seu proveito e fazer as iterações para obter sua arquitetura final.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Quais são as três vantagens de um projeto formal?
2. O que é Projeto Orientado a Objetos — POO?
3. O que é modelo de objetos?
4. Quais são os inconvenientes do projeto em demasia?

5. Como você sabe quais aspectos de seu sistema são *arquitetonicamente significativos*?
6. Quais são os cinco passos básicos do POO (Projeto Orientado a Objetos)?
7. Como você gera a lista de objetos inicial?
8. O que um projeto completo captura?
9. O que os cartões CRC o ajudam a identificar?
10. O que é uma colaboração?
11. Por que um entendimento profundo dos relacionamentos e responsabilidades de um objeto é importante?
12. O que é um cartão CRC?
13. Descreva um motivo pelo qual os cartões CRC são intencionalmente de baixa tecnologia.
14. Quando você deve usar cartões CRC?
15. Qual é um problema importante dos cartões CRC?
16. O que é ponto de interação?
17. Quais são as quatro considerações que você pode fazer em relação a um ponto de interação?
18. O que é um agente?
19. O que você fará quando detalhar os relacionamentos complexos entre os objetos e por que isso é importante?
20. Qual tipo de modelos você poderia criar?

## Exercícios

1. No Dia 9, Exercício 2, você desenvolveu o seguinte caso de uso:

- Remove item
  - O usuário convidado seleciona um item do carrinho de compras.
  - O usuário convidado solicita ao carrinho para que remova o item.
- Condições prévias
  - O carrinho contém um item para remover.
- Condições posteriores
  - O item não aparece mais no carrinho.
- Alternativa: Operação cancelada
  - O usuário pode optar por cancelar a transação, após o Passo 1.

Para melhorar suas habilidades, use cartões CRC para descobrir as responsabilidades. Quais responsabilidades o Carrinho de Compras terá?

# SEMANA 2

DIA **11**

## Reutilizando projetos através de padrões de projeto

No capítulo de ontem, você viu como o projeto orientado a objetos o ajuda a projetar uma solução para um problema. Através do POO (Projeto Orientado a Objetos), você constrói uma cópia heliográfica que diagrama os objetos que compreendem seu sistema. Uma vez que tenha esse projeto, você pode começar a implementar sua solução.

Entretanto, você provavelmente tem algumas perguntas.

- Como você sabe que o seu projeto é um bom projeto?
- Seu projeto terá consequências imprevisíveis no futuro?
- Como outros projetistas resolveram esse problema ou um problema semelhante no passado?
- E quanto à reutilização? A POO (Programação Orientada a Objetos) fornece reutilização de código, mas o POO (Projeto Orientado a Objetos) permite reutilização?<sup>1</sup>

Este capítulo o ajudará a responder essas perguntas e muito mais, quando você explorar o assunto ‘padrões de projeto’.

Hoje você aprenderá:

- Como usar padrões de projeto
- Como aplicar quatro padrões comuns

---

1. N.R.T.: Usaremos o POO para Projeto Orientado a Objetos e a POO para Programação Orientada a Objetos

- Como um projeto pode tirar proveito do uso de padrões
- Como evitar uma armadilha de padrão comum

## Reutilização de projeto

Um objetivo importante da POO é a reutilização de código. Quando reutiliza código, você ganha tranquilidade, sabendo que seu software está construído em uma base de código confiável e testada. Além disso, você sabe que o código reutilizado resolverá seu problema. Tal tranquilidade enquanto você programa é ótima, mas e quanto à tranquilidade enquanto projeta? Como você sabe se seu projeto é bom?

Felizmente, os padrões de projeto podem ajudar a dirimir muitas das dúvidas que você encontrará ao projetar. À medida que o tempo passa, muitos projetistas e programadores têm notado que os mesmos elementos de projeto aparecem repetidamente em todos os seus projetos. A comunidade de OO resolveu identificar, nomear e descrever esses conceitos de projeto recorrentes. O resultado é uma lista sempre crescente de padrões de projeto.

**Novo Termo** Um *padrão de projeto* é um conceito de projeto reutilizável.

Verifica-se que você pode reutilizar padrões de projeto em todo o seu projeto, exatamente como se você reutilizasse uma classe dentro de seu programa. Essa reutilização traz as vantagens da reutilização da POO (Programação Orientada a Objetos) para o POO (Projeto Orientado a Objetos). Quando usa padrões de projeto, você sabe que baseou seu projeto em projetos confiáveis e comprovados pelo uso. Tal reutilização permite que você saiba se está na trilha certa para uma solução confiável. Quando você reutiliza um padrão de projeto, está usando um projeto que outros usaram com êxito, muitas vezes anteriormente.

## Padrões de projeto

O livro *Design Patterns — Elements of Reusable Object-Oriented Software*, de Gamma, Helm, Johnson e Vlissides, apresentou pela primeira vez o conceito de padrões de projeto para muitos na comunidade de OO. Esse trabalho de base não apenas definiu um conjunto de padrões de projeto reutilizáveis, mas também definiu formalmente o padrão de projeto.

De acordo com esse trabalho, um padrão de projeto consiste em quatro elementos:

- O nome do padrão
- O problema
- A solução
- As consequências

## O nome do padrão

Um nome identifica exclusivamente cada padrão de projeto. Assim como a UML fornece uma linguagem de projeto comum, os nomes dos padrões fornecem um vocabulário comum para descrever os elementos de seu projeto para outros. Outros desenvolvedores podem entender seu projeto rápida e facilmente, quando você usa um vocabulário comum.

Um simples nome reduz um problema inteiro, a solução e as consequências a um único termo. Assim como os objetos o ajudam a programar em um nível mais alto e abstrato, esses termos permitem que você projete a partir de um nível mais alto e abstrato e não fique preso aos detalhes que se repetem de um projeto para outro.

## O problema

Cada padrão de projeto existe para resolver algum conjunto distinto de problemas de projeto e cada padrão de projeto descreve o conjunto de problemas para o qual foi feito para resolver. Desse modo, você pode usar a descrição do problema para determinar se o padrão se aplica ao problema específico que está enfrentando.

## A solução

A solução descreve como o padrão de projeto resolve o problema e identifica os objetos arquitetonicamente significativos na solução, assim como as responsabilidades e relacionamentos que esses objetos compartilham.

11



**NOTA**  
É importante notar que você pode aplicar um padrão de projeto a uma classe inteira de problemas. A solução é uma solução geral e não apresenta uma resposta para um problema específico ou concreto.

Suponha que você quisesse encontrar a melhor maneira de percorrer os itens do carrinho de compras do capítulo 10, “Introdução ao POO (Projeto Orientado a Objetos)”. O padrão de projeto Iterator propõe um modo de fazer justamente isso; entretanto, a solução apresentada pelo padrão Iterator não é dada em termos de carrinhos de compras e itens. Em vez disso, a solução descreve o processo de varredura de qualquer lista de elementos.

Quando você começar a usar um padrão de projeto, deve mapear a solução geral para seu problema específico. Às vezes, pode ser difícil fazer esse mapeamento; entretanto, o próprio padrão de projeto precisa permanecer genérico para que permaneça aplicável a muitos problemas específicos diferentes.

## As consequências

Não existe um projeto perfeito. Todo bom projeto terá bons compromissos e todo compromisso assumido terá seu próprio conjunto especial de consequências. Um padrão de projeto enumerará as principais consequências inerentes ao projeto.

As consequências não são novidades. Você assume compromissos sempre que opta entre duas alternativas ao programar. Considere a diferença entre usar um array e usar uma lista encadeada. Um array proporciona pesquisa rápida pelo índice, mas funciona muito lentamente, quando você precisa reduzir ou expandir o array para inserir ou excluir elementos. Por outro lado, a lista encadeada proporciona rápidas adições e exclusões, mas tem uma sobrecarga de memória maior e pesquisas de índice mais lentas. Aqui, as consequências de uma lista encadeada são a sobrecarga de memória e as pesquisas mais lentas. A consequência de usar um array é o fato de que o redimensionamento do array é dispendioso, mas as pesquisas indexadas são muito rápidas. O que você escolhe depende da importância da memória e da velocidade em seu projeto, se você vai fazer muitas adições e exclusões de elementos e se vai fazer muitas pesquisas.

É sempre importante documentar suas decisões de projeto, assim como as consequências resultantes. Ter essas decisões documentadas ajuda os outros a entender as escolhas que você fez e a determinar se o projeto pode ajudar a resolver seus próprios problemas. Do mesmo modo, as consequências do padrão de projeto pesarão bastante em sua decisão de usar o padrão. Se um padrão de projeto tem consequências que não correspondem aos objetivos de seu projeto, você não deve usá-lo — mesmo que ele possa resolver seu problema.

## Realidades do padrão

Quando você começa a aprender sobre padrões, é importante saber o que eles podem e o que não podem fazer. As listas a seguir podem ajudar a manter corretas as intenções por trás dos padrões.

Os padrões são:

- Projetos reutilizáveis que provaram funcionar no passado.
- Soluções abstratas para um problema de projeto geral.
- Soluções para problemas recorrentes.
- Um modo de construir um vocabulário de projeto.
- Um registro público da experiência do projeto.
- Uma solução para um problema.

Os padrões não são:

- Uma solução para um problema específico.
- A resposta mágica para todos os seus problemas.
- Uma muleta, você mesmo ainda precisa fazer seu projeto funcionar.
- Classes concretas, bibliotecas, soluções prontas.

## Padrões por exemplo

Existem muitos livros que catalogam os padrões de projeto. Não há motivo para repetir formalmente esses catálogos aqui; entretanto, existe um conjunto de padrões de projeto que você en-

contrará quase que diariamente. Nenhum texto introdutório sobre OO será completo sem apresentar esses padrões para você.

Em vez de apresentar esses padrões formalmente, vamos adotar uma estratégia mais informal, através de exemplo.

Hoje, este capítulo apresentará três padrões importantes:

- Adapter
- Proxy
- Iterator

## O padrão Adapter

O Capítulo 4, “Herança: obtendo algo para nada”, apresentou o conceito de relacionamentos com capacidade de substituição. O Capítulo 6, “Polimorfismo: aprendendo a prever o futuro”, mostrou como você pode usar esses relacionamentos para adicionar novos objetos em seu sistema a qualquer momento. Quando você usa herança para definir capacidade de substituição, entretanto, seus objetos podem se tornar restritos pelas hierarquias resultantes. Para poder se conectar com seu programa, um objeto deve fazer parte da hierarquia com capacidade de substituição.

Então, o que você faria se quisesse conectar um objeto em seu programa, mas ele não pertencesse à hierarquia correta? Uma solução seria pegar a classe que você quisesse usar e, em seguida, editá-la para que ela herde da classe correta. Essa solução não é ótima por alguns motivos.

Nem sempre você terá o código-fonte das classes que deseja usar. Além disso, se essa classe já usar herança, você terá problemas se sua linguagem não suportar herança múltipla.

Mesmo que você tenha o código-fonte, simplesmente não é prático ou razoável reescrever um objeto cada vez que você quisesse que ele fizesse parte da hierarquia ‘correta’. A definição da hierarquia ‘correta’ mudará para cada programa. Se você tiver uma classe razoavelmente abstraiada, não desejará ficar editando sempre que quiser reutilizá-la. Em vez disso, você deve simplesmente utilizá-la inalterada. Reescrevê-la a cada vez anula os propósitos da reutilização e da abstração. Se você ficar criando edições especiais de uma classe, poderão restar muitas classes redundantes para manter.

O padrão Adapter apresenta uma solução alternativa que resolve o problema da incompatibilidade, transformando a interface incompatível naquela que você precisa. O padrão Adapter funciona empacotando o objeto incompatível dentro de um objeto adaptador compatível. O objeto adaptador contém uma instância do objeto e expõe o objeto através da interface que se encaixa em seu programa. Como a classe adaptadora empacota um objeto, às vezes esse padrão é referido como padrão Empacotador.

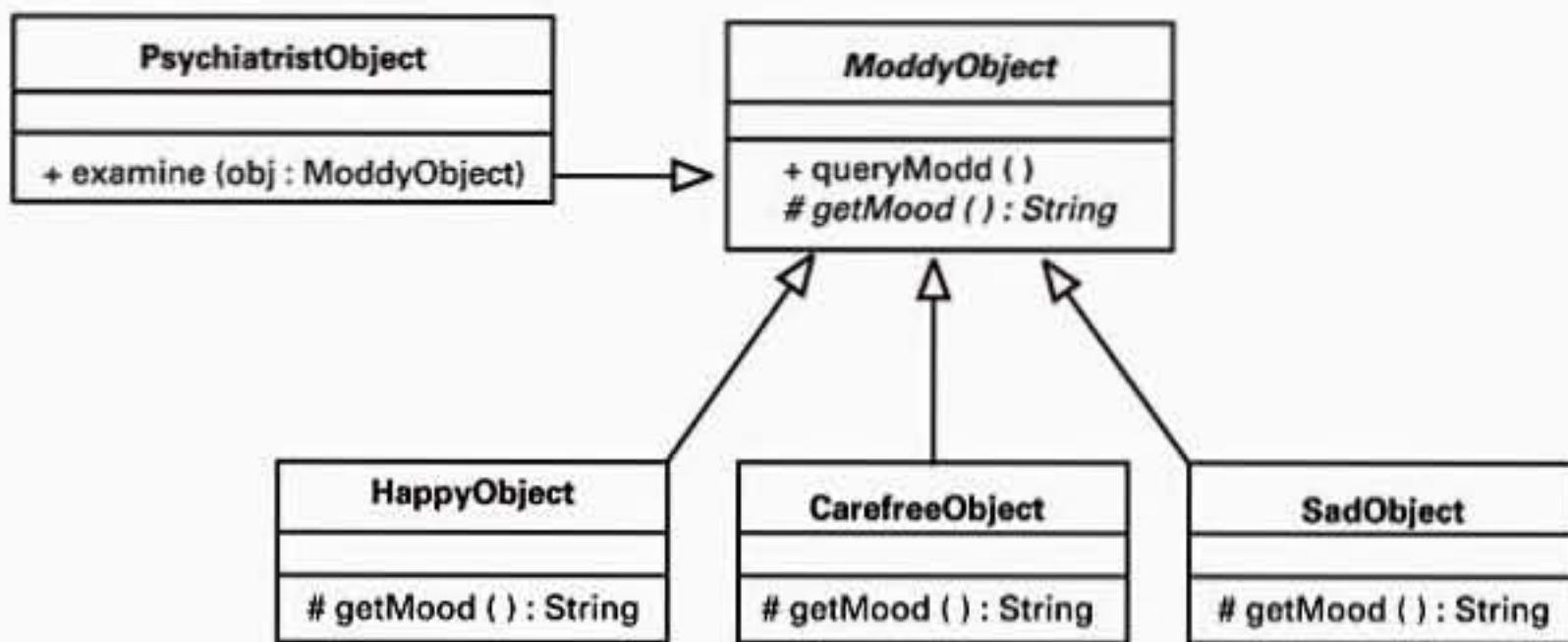
11

**Novo TERMO** Um *adaptador* é um objeto que transforma a interface de outro objeto.

## Implementando um adaptador

A Figura 11.1 resume a hierarquia MoodyObject apresentada no Capítulo 7, “Polimorfismo: hora de escrever código”.

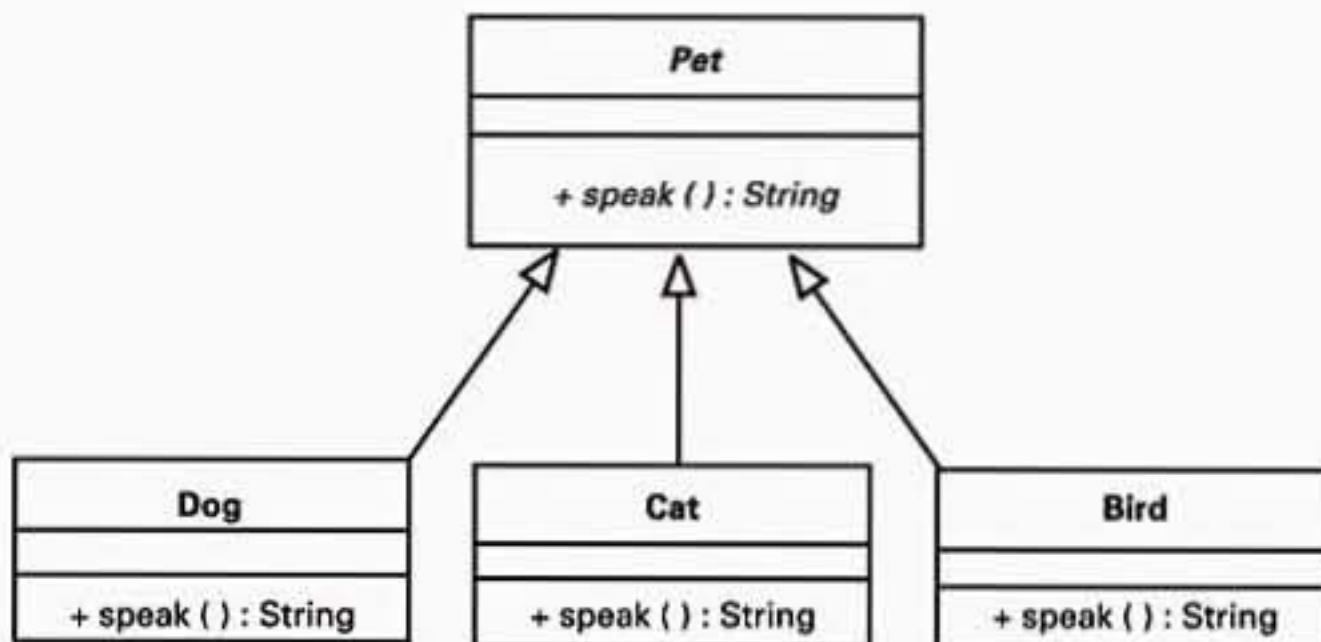
**FIGURA 11.1**  
A hierarquia MoodyObject.



PsychiatristObject só pode examinar um objeto se for um MoodyObject. O método examine() de PsychiatristObject depende do método queryMood() de MoodyObject. Qualquer outro tipo de objeto precisará encontrar um psiquiatra diferente.

A Figura 11.2 apresenta a nova hierarquia Pet.

**FIGURA 11.2**  
A hierarquia Pet.



Cada Pet fala de sua própria maneira especializada e fornece sua própria implementação de speak().

Então, qual é o problema?

Hoje em dia, até animais de estimação vão ao psiquiatra, mas o objeto PsychiatristObject não pode examinar um Pet porque um Pet não é um objeto MoodyObject. Para que o objeto PsychiatristObject possa examinar um Pet, você precisará criar um adaptador Pet.

A Listagem 11.1 apresenta um possível adaptador Pet.

**LISTAGEM 11.1 PetAdapter.java**

```
public class PetAdapter extends MoodyObject {  
  
    private Pet pet;  
  
    public PetAdapter( Pet pet ) {  
        this.pet = pet;  
    }  
    protected String getMood() {  
        // implementando apenas porque é exigido por  
        // MoodyObject, como também sobrepõe queryMood,  
        // não precisamos disso  
        return pet.speak();  
    }  
  
    public void queryMood() {  
        System.out.println( getMood() );  
    }  
}
```

O PetAdapter empacota uma instância de Pet. Em vez de expor a interface Pet, o adaptador oculta essa interface atrás da interface MoodyObject, basicamente transformando a interface de Pet. Quando chega um pedido no PetAdapter, o adaptador delega para a instância de Pet, conforme for necessário.

A Listagem 11.2 mostra o adaptador em ação.

11

**LISTAGEM 11.2 Usando o adaptador**

```
PetAdapter dog = new PetAdapter( new Dog() );  
PetAdapter cat = new PetAdapter( new Cat() );  
PetAdapter bird = new PetAdapter( new Bird() );  
PsychiatristObject psychiatrist = new PsychiatristObject();  
  
psychiatrist.examine( dog );  
psychiatrist.examine( cat );  
psychiatrist.examine( bird );
```

Uma vez empacotada, a instância de Pet se parece com qualquer outro MoodyObject para o objeto PsychiatristObject. Essa solução é mais eficiente do que as alternativas, pois ela exige que você escreva apenas uma classe adaptadora. Essa classe adaptadora pode manipular qualquer objeto Pet que apareça. A parte complicada é garantir que o PetAdapter empacote todas as instâncias de Pet, antes que você passe o objeto Pet para o objeto PsychiatristObject.

A implementação fornecida anteriormente é chamada de adaptador de objeto, pois o adaptador usa composição para transformar a interface de uma instância. Você também pode implementar um adaptador através de herança. Tal adaptador é conhecido como adaptador de classe, pois ele adapta a própria definição da classe.

Se sua linguagem suporta heranças múltiplas, então você sempre pode herdar da classe que deseja usar, assim como de uma classe da hierarquia.

Se sua linguagem não tem herança múltipla, como Java, suas escolhas podem ser limitadas, dependendo de como a hierarquia foi construída. Na verdade, talvez você tenha de se privar da herança totalmente e usar composição, no ponto em que também poderia criar um adaptador de objeto.

Embora a herança múltipla funcione, essa solução é limitada. Criar uma nova subclasse para cada classe que você quiser usar pode levar a uma proliferação inaceitável em classes empacotadoras.

Cada estratégia tem sua própria limitação. Um adaptador de classe só funcionará para a classe que herda. Você precisará de um adaptador separado para cada subclasse.

Do mesmo modo, embora um adaptador de objeto possa funcionar com cada subclasse, você precisará de subclasses do adaptador, se quiser mudar a maneira como ele empacota várias subclasses.

## Quando usar o padrão Adapter

O padrão Adapter é útil quando você quer usar um objeto que tem uma interface incompatível. O padrão Adapter permite que você reutilize diretamente objetos que, de outro modo, precisaria alterar ou jogar fora.

Os adaptadores também são úteis no sentido de uma ação preventiva. De tempos em tempos, você precisará雇用 bibliotecas de terceiros em seus programas. Infelizmente, as APIs das ferramentas de terceiros podem variar substancialmente entre as versões, especialmente para novos produtos ou para tecnologias que estão amadurecendo. As APIs também podem variar bastante em relação às bibliotecas de um produto concorrente.

O padrão Adapter pode ajudar a evitar que seu programa tenha que trocar de APIs e que fique preso ao fornecedor. Criando uma interface adaptadora que você controla, é possível trocar para novas versões de uma biblioteca a qualquer momento. Basta criar uma subclasse do adaptador para cada biblioteca que você quiser usar.

Também é importante notar que um adaptador pode ser simples ou complicado e o nível de complexidade depende do objeto que está sendo empacotado. Alguns adaptadores se resumem simplesmente a mapear um pedido para o método correto. Outros precisarão realizar mais processamento.

Use o padrão Adapter, quando:

- você quiser usar objetos incompatíveis em seu programa;
- você quiser que seu programa permaneça independente de bibliotecas de terceiros.

A Tabela 11.1 destaca o usuário do padrão Adapter.

**TABELA 11.1** O padrão Adapter

<i>Nome do padrão</i>	Adapter, Wrapper
<i>Problema</i>	Como reutilizar objetos incompatíveis
<i>Solução</i>	Fornecer um objeto que converta a interface incompatível em uma compatível
<i>Consequências</i>	Tornar incompatíveis objetos compatíveis; resulta em classes extras — talvez muitas —, se você usar herança ou precisar manipular cada subclasse de uma forma diferente

## O padrão Proxy

Normalmente, quando um objeto quer interagir com outro, ele faz isso atuando diretamente sobre o outro objeto. Na maioria dos casos, essa estratégia direta é a melhor estratégia, mas existem ocasiões em que você desejará controlar o acesso entre seus objetos de forma transparente. O padrão Proxy trata desses casos.

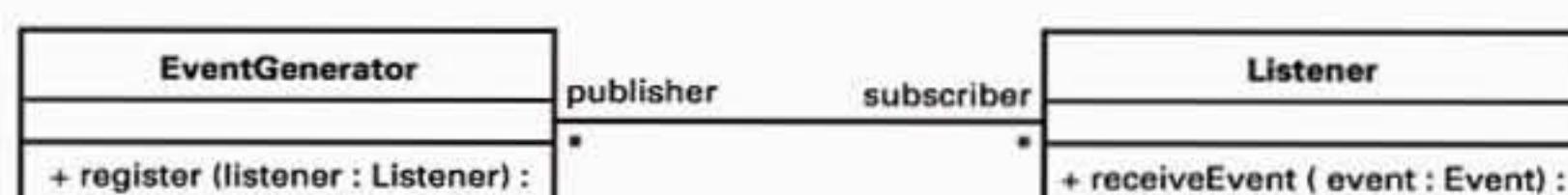
### Publicação/assinatura e o padrão Proxy

Considere o problema de um serviço de eventos publicação/assinatura onde um objeto vai registrar seu interesse em um evento. Quando o evento ocorrer, o publicador notificará os objetos assinantes do evento.

A Figura 11.3 ilustra um possível relacionamento publicação/assinatura.

11

**FIGURA 11.3**  
Um relacionamento  
publicação/assinatura.



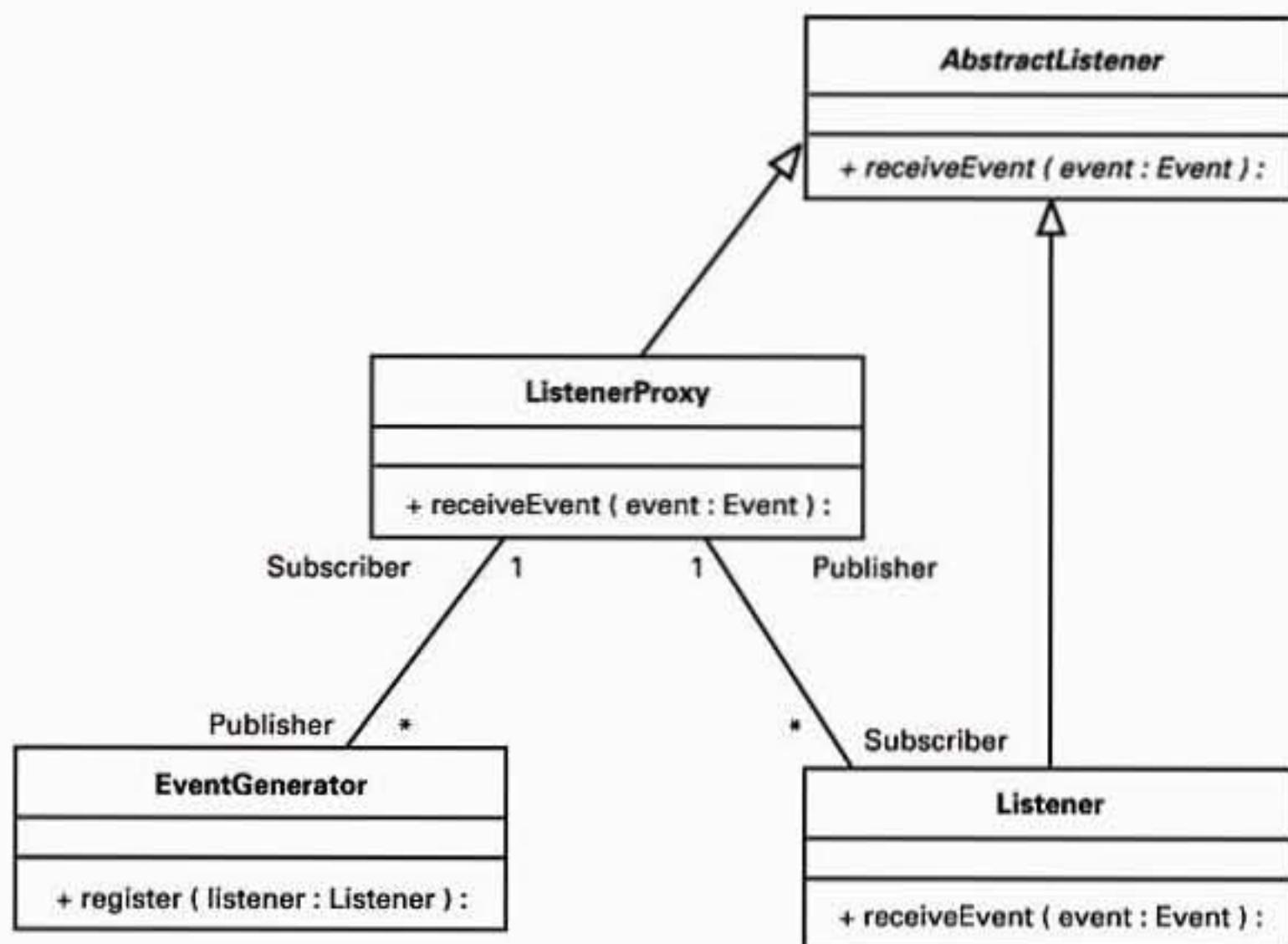
Na Figura 11.3, muitos *Listeners* registram seu interesse nos eventos gerados por um *EventGenerator*. Quando o *EventGenerator* gerar um evento, ele colocará o evento em cada um de seus objetos *Listeners*.

Embora essa solução funcione, ela coloca uma grande carga sobre o *EventGenerator*. O *EventGenerator* não tem apenas a responsabilidade de gerar eventos, como também é responsável por controlar todos os *Listeners* e colocar os eventos neles.

O padrão Proxy apresenta uma solução elegante para esse problema. Considere a Figura 11.4.

Em vez de conter seus *Listeners* diretamente, o *EventGenerator* pode conter um *ListenerProxy*. Quando o gerador precisa disparar um evento, ele o dispara uma vez para o proxy. Então, fica por conta do proxy controlar e atualizar todos os *Listeners*.

**FIGURA 11.4**  
Uma solução proxy.



## O padrão Proxy geral

O cenário publicação/assinatura descreve um possível uso para um proxy; entretanto, você pode usar proxies em muitos lugares.

Primeiro, o que é um proxy?

Um *proxy* é um substituto ou lugar reservado que intermedia o acesso ao objeto de interesse real. Você pode usar um proxy em qualquer lugar onde precise de um substituto ou lugar reservado para outro objeto. Em vez de usar um objeto diretamente, você usa o proxy. O proxy cuida de todos os detalhes da comunicação com o objeto (ou objetos) real.

### Novo TERMO

Um *proxy* é um substituto ou lugar reservado que intermedia o acesso ao objeto de interesse real. Para todos os efeitos, o *substituto* é indistinguível do objeto real que intermedia.

Um substituto intermedia o acesso a um objeto (ou objetos) subjacente de forma transparente. Você pode considerar um substituto como um modo de enganar seus objetos. Por exemplo, enquanto o EventGenerator pensa que está se comunicando apenas com um Listener, o ListenerProxy pode, na verdade, divulgar a mensagem para muitos objetos Listener diferentes.

Poder enganar seus objetos é uma capacidade importante, pois ela permite que você realize todos os tipos de mágica nos bastidores. Um substituto permite que você coloque responsabilidades nele, sem ter de incorporar essas responsabilidades no usuário do substituto. Mais importante, seu substituto pode executar todas essas responsabilidades sem que os outros objetos saibam o que você está fazendo.

As responsabilidades que você pode colocar no substituto são infinitas; entretanto, o uso comum inclui adicionar otimizações, realizar tarefas de limpeza, fazer recursos remotos parecerem locais e adiar operações dispendiosas.

## Quando usar o padrão Proxy

Use substitutos quando:

- Você quiser adiar uma operação dispendiosa. Considere um objeto que extrai informações de um banco de dados. Embora seu programa talvez precise saber a respeito dos objetos que pode extrair, ele não precisa extraír todas as informações até que realmente acesse o objeto. Um substituto pode representar o objeto real e carregar as informações extras quando elas forem necessárias.

Outro exemplo é um substituto de arquivo, que permite gravar em um arquivo, mas que apenas realiza a operação de Entrada e Saída no arquivo real, quando você já tiver terminado. Em vez de fazer várias gravações lentas, o substituto fará uma única gravação grande.

- Você quer proteger de forma transparente o modo como um objeto é usado. A maioria dos objetos é mutante, como as classes da linguagem Java. Um substituto protetor poderia tornar uma coleção de classes imutável, interceptando pedidos que de outro modo alterariam uma coleção de classes. Filtrando todas as chamadas de método através de um substituto, você pode governar de forma transparente as operações permitidas sobre um objeto.
- O objeto real existe remotamente, através de uma rede ou processo. Os substitutos são importantes para a computação distribuída. Um substituto pode fazer com que um recurso distribuído pareça como se fosse um recurso local, encaminhando os pedidos através de uma rede ou através de um espaço de processo.
- Quando você quer executar ações adicionais de forma transparente, ao usar um objeto. Por exemplo, talvez você queira contar o número de vezes que um método é chamado sem que o chamador saiba. Um substituto de contagem poderia controlar o acesso a um objeto.

A Tabela 11.2 destaca o usuário do padrão Proxy.

**TABELA 11.2** O padrão Proxy

<i>Nome do padrão</i>	Proxy, Surrogate
<i>Problema</i>	Precisa controlar o acesso a um objeto
<i>Solução</i>	Fornecer um objeto que intermedia o acesso a outro objeto de forma transparente
<i>Conseqüências</i>	Introduz um nível de procedimento indireto no uso do objeto

11

## O padrão Iterator

Os padrões de projeto freqüentemente descrevem soluções para problemas comuns. Quando você programa, são boas as chances de que vá gastar muito tempo escrevendo códigos que fa-

zem laços (loops) sobre objetos de uma hierarquia. Conforme você aprendeu no Capítulo 8 “Introdução à UML”, o relacionamento entre objetos é uma *agregação*. O todo contém partes.

A Listagem 11.3 apresenta um método conhecido que faz um laço sobre o conteúdo de um maço de cartas (uma coleção) e constrói uma representação de caracteres do maço.

---

**LISTAGEM 11.3** Fazendo um laço sobre o conteúdo de um monte de cartas

```
public String deckToString( Deck deck ) {  
    String cards = "";  
    for( int i = 0; i < deck.size(); i ++ ) {  
        Card card = deck.get( i );  
        cards = cards + card.display();  
    }  
    return cards;  
}
```

---

Suponha que você quisesse fazer um laço sobre um array de cartas. A Listagem 11.4 mostra o método que pode manipular justamente esse caso.

---

**LISTAGEM 11.4** Laço sobre o conteúdo de um array

```
public String deckToString( Card [] deck ) {  
    String cards = "";  
    for( int i = 0; i < deck.length; i ++ ) {  
        cards = cards + deck [i].display();  
    }  
    return cards;  
}
```

---

Finalmente, e se você quiser fazer um laço sobre o maço de cartas na ordem inversa? Você precisará adicionar um outro método. A Listagem 11.5 mostra um método que faz um laço sobre o maço na ordem inversa.

---

**LISTAGEM 11.5** Laço sobre o conteúdo de um maço de cartas na ordem inversa

```
public String reverseDeckToString( Deck deck ) {  
    String cards = "";  
    for( int i = deck.size() - 1; i > -1; i - ) {  
        Card card = deck.get( i );  
        cards = cards + card.display();  
    }  
    return cards;  
}
```

---

Se você lembrar das lições sobre acoplamento e responsabilidades, deverá ouvir uma voz em sua mente, gritando, “Código ruim!” Sempre que você quiser fazer um laço sobre uma coleção de cartas, precisará adicionar um método que saiba como fazer um laço sobre esse tipo de coleção específico. Na verdade, a lógica não é tão diferente assim de um caso para outro. Os únicos elementos que mudam são os métodos e atributos que você acessa na coleção. Agora você vê um alerta: código repetido.

O problema é que cada um dos métodos listados anteriormente é dependente da implementação da coleção, como um Deck ou um Array. Quando programa, você sempre quer garantir que não fique acoplado a uma implementação específica. O acoplamento torna seu programa resistente à mudança.

Pense a respeito. O que acontecerá se você quiser mudar a coleção que contém suas cartas? Você precisará atualizar ou adicionar um novo método que faça o laço. Se você não tiver sorte, simplesmente mudar a implementação da coleção poderia necessitar de alterações por todo o seu programa.



**Embora um único objeto possa conter apenas um laço, seu programa inteiro provavelmente vai repetir os laços muitas vezes, em muitos lugares diferentes.**

Outro problema com esses exemplos provém do fato de que os mecanismos de navegação estão codificados no método. É por isso que você precisa de um método para o laço para frente e outro para o laço inverso. Se você quiser fazer um laço aleatoriamente pelas cartas, então precisará de um terceiro método (e um para cada tipo de coleção). Você não precisará apenas de vários métodos, como também precisará implementar novamente a lógica de navegação, sempre que definir um laço. Infelizmente, tal duplicação de lógica é um sintoma de responsabilidade confusa. A lógica de navegação deve aparecer em um e apenas um lugar.

11

Felizmente, o padrão Iterator resolve muitos dos problemas de forte acoplamento e confusão de responsabilidade, colocando a lógica do laço ou iteração em seu próprio objeto.

A Figura 11.5 ilustra a interface Iterator.

**FIGURA 11.5**  
*A interface Iterator.*

<b>Iterator</b>
<code>+ first () : void</code>
<code>+ next () : void</code>
<code>+ isDone () : boolean</code>
<code>+ currentItem () : Object</code>

A interface Iterator fornece uma interface genérica para iterar sobre uma coleção. Em vez de escrever seus laços e métodos para usar uma coleção específica, você pode simplesmente programar a interface genérica do Iterator. A interface Iterator oculta completamente a implementação da coleção subjacente.

**NOTA**

A linguagem Java define uma interface iteradora ligeiramente diferente: `java.util.Iterator`. O Iterator Java define apenas três métodos: `public boolean hasNext()`, `public void remove()` e `public Object next()`.

A interface Java Iterator permite que você faça uma iteração apenas em uma direção. Ao chegar ao fim, você não pode voltar para o início. Fora essa deficiência, a interface Java Iterator é semelhante àquela apresentada anteriormente.

Ao escrever programas Java, você deve usar `java.util.Iterator` para que outros programas Java possam usar suas implementações do Iterador. Para os propósitos deste livro, entretanto, esta lição continuará sendo verdade para a definição do Iterador clássica fornecida na Figura 11.5.

A Listagem 11.6 apresenta um método alternativo `deckToString()`, que faz laço sobre uma instância de `Iterator`.

**LISTAGEM 11.6** Laço sobre o conteúdo de uma instância de `Iterator`

```
public String deckToString( Iterator i ) {
    String cards = "";
    for ( i.first(); !i.isDone(); i.next() ) {
        Card card = (Card) i.currentItem();
        cards = cards + card.display();
    }
    return cards;
}
```

Apenas porque um objeto passa de volta um iterador, não significa que o objeto realmente armazena seus itens dentro do iterador. Em vez disso, um iterador dará acesso ao conteúdo do objeto.

Existem três vantagens de usar um iterador para percorrer uma coleção.

Primeiro, um iterador não o vinculará a uma coleção específica. Todos os métodos originais fariam laço sobre implementações específicas da coleção. Como resultado, cada método diferiria apenas nos métodos que chama na coleção. Se esses métodos fizessem um laço sobre um iterador, como na Listagem 11.4, você só precisaria escrever um método `deckToString()`.

Segundo, o iterador pode retornar seus elementos em qualquer ordem que achar conveniente. Isso significa que uma implementação do iterador poderia retornar os elementos em ordem. Outro iterador poderia retornar os elementos na ordem inversa. Usando um iterador, você pode escrever a lógica de navegação uma vez e fazer com que ela apareça em apenas um lugar: no próprio iterador.

Finalmente, um iterador torna simples mudar a coleção subjacente, quando isso for necessário. Como você não programou para uma implementação específica, pode trocar para uma nova coleção a qualquer momento, desde que a coleção saiba como retornar uma instância de `Iterator`.

## Implementando um Iterador

**NOTA**

A maioria das coleções Java já dá acesso a um iterador. Por exemplo, a `LinkedList` usada internamente pelo objeto `Deck` já tem um método `iterator()` que retorna um `java.util.Iterator`; entretanto, isso ajuda a ver uma implementação de iterador real e esse padrão não é específico da linguagem Java. Assim, ignore o método `iterator()` de `LinkedList` da linguagem Java por uns instantes.

A Listagem 11.7 mostra uma implementação de `Iterator` que permite ao usuário percorrer os elementos de uma coleção `LinkedList`.

### **LISTAGEM 11.7** Uma implementação de `Iterator`

```
public class ForwardIterator implements Iterator {  
    private Object [] items;  
    private int index;  
  
    public ForwardIterator( java.util.LinkedList items ) {  
        this.items = items.toArray();  
    }  
  
    public boolean isDone() {  
        if( index == items.length ) {  
            return true;  
        }  
        return false;  
    }  
  
    public Object currentItem() {  
        if( !isDone() ) {  
            return items [index];  
        }  
        return null;  
    }  
  
    public void next() {  
        if( !isDone() ) {  
            index++;  
        }  
    }  
  
    public void first() {  
        index = 0;  
    }  
}
```

Outra implementação poderia fornecer uma iteração inversa (veja o código-fonte completo, para um exemplo disso).

A Listagem 11.8 ilustra as alterações que você precisará fazer em Deck para que possa retornar um Iterator.

---

**LISTAGEM 11.8** Uma classe Deck atualizada

```
public class Deck {  
    private java.util.LinkedList deck;  
  
    public Deck() {  
        buildCards();  
    }  
  
    public Iterator iterator() {  
        return new ForwardIterator( deck );  
    }  
    // reduzido por brevidade  
}
```

---

Alternativamente, é totalmente válido, do ponto de vista da OO, considerar um iterador como uma extensão da coleção a que ele dá acesso. Como resultado, você tem algumas outras opções de implementação.

A linguagem Java possibilita uma construção conhecida como classe interna. Uma classe interna é uma classe definida dentro de outra classe. Como a classe interna é definida dentro de outra classe, ela tem total acesso a todos os métodos públicos dessa classe, assim como aos métodos protegidos e privados, e às variáveis internas. A analogia mais próxima na linguagem C++ é uma classe friend. friend fornece acesso especial a outro objeto confiável.

É fácil abusar da classe friend e da classe interna, pois elas podem destruir o encapsulamento facilmente; entretanto, um iterador realmente faz parte da coleção. A Listagem 11.9 mostra uma implementação de classe interna do Iterator da classe Deck.

---

**LISTAGEM 11.9** Uma implementação de Iterator de classe interna

```
public class Deck {  
  
    private java.util.LinkedList deck;  
    public Deck() {  
        buildCards();  
    }  
  
    public Iterator iterator() {
```

```
        return new ForwardIterator();
    }

//reduzido por brevidade

private class ForwardIterator implements Iterator {

    int index;

    public boolean isDone() {
        // note que a classe interna tem acesso
        // liberado à variável interna deck de Deck
        if(index == deck.size() ) {
            return true;
        }
        return false;
    }

    public Object currentItem() {
        if( !isDone() ) {
            return deck.get( index );
        }
        return null;
    }

    public void next() {
        if( !isDone()) {
            index++;
        }
    }

    public void first() {
        index = 0;
    }
}
```

## Quando usar o padrão Iterator

Existem várias razões para se usar o padrão Iterator:

- Você pode usar um iterador quando quiser ocultar a implementação de uma coleção.
- Você pode usar um iterador quando quiser fornecer diferentes tipos de laços sobre uma coleção (como laço para frente, laço inverso, laço filtrado etc.).
- Você pode usar um iterador para manter a interface de uma coleção simples. Você não precisará adicionar métodos para ajudar a fazer o laço sobre o conteúdo. Basta deixar os usuários do objeto utilizarem um iterador.
- Você pode definir uma classe de coleção base que retorne um iterador. Se todas as suas coleções herdarem dessa base, o iterador permitirá que você trate todas as suas coleções genericamente. Na verdade, `java.util.Collection` faz justamente isso. Essa utilização também é a forma geral do padrão Iterator. O exemplo Deck é uma versão abreviada do padrão Iterator. A classe Deck não herda de uma classe de coleção base abstrata; assim, você não pode tratá-la genericamente.
- Os iteradores também são úteis para fornecer acesso otimizado às coleções. Algumas estruturas de dados, como a tabela hashing, não fornecem um modo otimizado de fazer a iteração sobre os elementos. Um iterador pode fornecer tal ordenação, ao custo de um pouco de memória extra. Entretanto, dependendo de seu aplicativo, a economia de tempo pode mais do que compensar a sobrecarga de memória.

A Tabela 11.3 destaca o usuário do padrão Iterator.

**TABELA 11.3** O padrão Iterator

---

<i>Nome do padrão</i>	Iterator, Cursor
<i>Problema</i>	Fazer laço sobre uma coleção sem se tornar dependente da implementação da coleção
<i>Solução</i>	Fornecer um objeto que manipule os detalhes da iteração, ocultando assim os detalhes do usuário
<i>Consequências</i>	Navegação desacoplada, interface da coleção mais simples, lógica de laço encapsulada

---

## Apossando-se de um padrão

Alguns padrões são mais difíceis de entender do que outros; entretanto, a complexidade pode ser enganosa. Antes de poder aplicar ou mesmo pensar sobre a alteração de um padrão, você precisa entender o padrão: em outras palavras “apossar-se dele”. Existem alguns passos que você deve dar para poder dominar um padrão:

1. Leia o padrão.
2. Leia o padrão novamente, prestando bastante atenção nos principais participantes e no exemplo de código.

3. Pratique a implementação do padrão.
4. Aplique o padrão em um problema real.

Quando você tiver concluído esses quatro passos, provavelmente terá um entendimento total do padrão. Neste ponto, você pode começar a alterar o padrão de acordo com as necessidades específicas de seu problema; entretanto, você não deve tentar aumentar um padrão com o qual ainda não esteja familiarizado.

## Resumo

Os padrões de projeto são uma ajuda útil ao projetar suas soluções. De sua própria maneira, os padrões são a consciência coletiva da comunidade de OO, que tem anos de experiência de projeto. Os padrões de projeto podem fornecer diretrizes valiosas durante o projeto.

Você precisa ter em mente os limites dos padrões de projeto, quando os utilizar. Um padrão de projeto trata de um problema abstrato e de apenas um problema. Um padrão de projeto não fornece a solução para um problema específico. Em vez disso, o padrão fornece uma solução abstrata para um problema geral. Fica por sua conta fornecer o mapeamento entre o problema abstrato e seu problema específico.

Mapear um padrão de projeto provavelmente é o maior desafio que você enfrentará enquanto usar padrões. É uma habilidade que vem apenas com o tempo, estudo e prática.

11

## Perguntas e respostas

**P A linguagem Java usa padrões?**

**R** Sim, muitas das APIs Java empregam padrões. Na verdade, cada um dos padrões que você leu a respeito hoje está representado na linguagem Java. Aqui está uma lista breve dos padrões que você viu:

Iterator: as classes Java Collection

Proxy: RMI da linguagem Java

Adapter: usado amplamente por receptores de evento

**P Todos os padrões são traduzidos para todas as linguagens?**

**R** Não. Cada linguagem é diferente. Alguns padrões são impossíveis de implementar em certas linguagens, enquanto outros são desnecessários, devido aos recursos internos da linguagem.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

### Teste

1. O que é uma classe adaptadora?
2. Qual problema o padrão Iterator resolve?
3. Por que você usaria o padrão Iterator?
4. Qual problema o padrão Adapter resolve?
5. Por que você usaria o padrão Adapter?
6. Qual problema o padrão Proxy resolve?
7. Por que você usaria o padrão Proxy?
8. Considere a situação a seguir; qual padrão você usaria e por que?

A Sun Microsystems, IBM e Apache fornecem bibliotecas para analisar documentos XML. Você optou por usar a biblioteca Apache em seu aplicativo XML. No futuro, contudo, você poderia optar por usar um fornecedor diferente.

9. Considere a situação a seguir; qual padrão você usaria e por que?

Você deve escrever um aplicativo que recupere dados de um armazém de dados baseado em arquivo. Às vezes, seu aplicativo será executado de forma local e poderá acessar o armazém de dados através de meios diretos. Outras vezes, o cliente será executado de forma remota e precisará se comunicar com um servidor para ler o armazém de dados.

10. O padrão Adapter transforma uma interface. O padrão Proxy altera a interface de um objeto?

### Exercícios

1. As listagens 11.10 e 11.11 apresentam uma classe de carrinho de compras e uma classe de item. A Listagem 11.12 apresenta uma interface iterator. Use essas definições para criar um iterator que permita fazer a iteração pelo conteúdo do carrinho de compras.

---

#### **LISTAGEM 11.10 Item.java**

```
public class Item {  
  
    private int id;  
    private int quantity;  
    private float unit_price;
```

**LISTAGEM 11.10** Item.java (*continuação*)

```
private String description;
private float discount;

/** 
 * Cria um novo item com a quantidade, preço,
 * descrição e desconto por unidade dados.
 * @param id a id do produto
 * @param quantity o número de itens selecionados
 * @param unit_price o preço com desconto anterior
 * @param description a descrição do produto
 * @param discount o valor em dólar a ser subtraído por item
 */
public Item( int id, int quantity, float unit_price, float discount,
String desc) {
    this.id = id;
    this.quantity = quantity;
    this.unit_price = unit_price;
    this.discount = discount;
    this.description = desc;
}

/** 
 * @return int a quantidade do item
 */
public int getQuantity() {
    return quantity;
}

/** 
 * @param quantity a nova quantidade
 */
public void setQuantity( int quantity ) {
    this.quantity = quantity;
}

/** 
 * @return o preço unitário do item
 */
public float getUnitPrice() {
    return unit_price;
}

/** 
 * @return float o preço total do item menos todos os descontos
*/
```

**LISTAGEM 11.10** Item.java (*continuação*)

```
public float getTotalPrice() {
    return ( unit_price * quantity ) - ( discount * quantity );
}

/**
 * @return String a descrição do produto
 */
public String getDescription() {
    return description;
}

/**
 * @return int a id do produto
 */
public int getID() {
    return id;
}
}
```

---

**LISTAGEM 11.11** ShoppingCart.java

```
public class ShoppingCart {

    java.util.LinkedList items = new java.util.LinkedList();
    /**
     * adiciona um item no carrinho
     * @param item o item a adicionar
     */
    public void addItem( Item item ) {
        items.add( item );
    }

    /**
     * remove o item dado do carrinho
     * @param item o item a remover
     */
    public void removeItem( Item item ) {
        items.remove( item );
    }

    /**
     * @return int o número de itens no carrinho
     */

    public int getNumberItems() {
        return items.size();
    }
}
```

**LISTAGEM 11.11 ShoppingCart.java (continuação)**

```
/**  
 * recupera o item indexado  
 * @param index o índice do item  
 * @return Item o item no índice  
 */  
public Item getItem( int index ) {  
    return (Item) items.get( index );  
}  
}
```

**LISTAGEM 11.12 Iterator.java**

```
public interface Iterator {  
    public void first();  
    public void next();  
    public boolean isDone();  
    public Object currentItem();  
}
```

2. O PetAdapter, apresentado anteriormente no capítulo, está limitado a apenas empacotar uma instância de Pet. Altere o adaptador de modo que você possa mudar o objeto que ele empacota, a qualquer momento.

Por que você desejará um adaptador mutante?

11

## Respostas do teste

1. Uma classe adaptadora transforma a interface de um objeto em uma interface esperada por seu programa. Um adaptador contém um objeto e delega mensagens da nova interface para a interface do objeto contido.
2. O padrão Iterator descreve um mecanismo para fazer laços pelos elementos de uma coleção.
3. Você usaria o padrão Iterator para conter lógica de navegação em um lugar, para fornecer um modo padronizado para percorrer coleções e para ocultar do usuário a implementação da coleção.
4. O padrão Adapter descreve um mecanismo que permite a você transformar uma interface de objetos.
5. Você usaria o padrão Adapter quando precisasse utilizar um objeto que tivesse uma interface incompatível. Você também pode usar empacotadores preventivamente, para isolar seu código das alterações de APIs.

6. O padrão Proxy intermedia de forma transparente o acesso a um objeto. Os proxies adicionam um procedimento indireto para uso de objetos.
7. Você usaria o padrão Proxy sempre que quisesse intermediar o acesso a um objeto de uma maneira que uma referência simples não permitisse. Exemplos comuns incluem recursos remotos, otimizações e limpeza de objeto geral, como contagem de referência ou reunião de estatísticas de utilização.
8. Nessa situação, você pode usar o padrão Adapter para criar uma interface independente daquela fornecida pela Sun, IBM ou Apache. Criando sua própria interface, você pode permanecer independente da API ligeiramente diferente de cada fornecedor. Empacotando a biblioteca, você está livre para trocar de biblioteca a qualquer momento, seja para migrar para uma nova versão ou para trocar de fornecedor, pois você controla a interface do adaptador.
9. Nessa situação, você pode usar o padrão Proxy para ocultar a identidade do objeto armazém de dados com que seus objetos se comunicam. Dependendo da localização do cliente, você pode instanciar um proxy interligado em rede ou um proxy local. De qualquer modo, o restante do programa não saberá a diferença; portanto, todos os seus objetos podem usar uma interface proxy sem ter de se preocupar com a implementação subjacente.
10. O padrão Proxy não altera uma interface, no sentido de que ele não retira nada dela. Entretanto, um proxy está livre para adicionar mais métodos e atributos na interface.

## Respostas dos exercícios

1.

---

### LISTAGEM 11.13 ShoppingCart.java

---

```
public class ShoppingCart {  
    java.util.LinkedList items = new java.util.LinkedList();  
  
    /**  
     * adiciona um item no carrinho  
     * @param item o item a adicionar  
     */  
    public void addItem( Item item ) {  
        items.add( item );  
    }  
  
    /**  
     * remove o item dado do carrinho  
     * @param item o item a remover  
     */  
    public void removeItem( Item item ) {  
        items.remove( item );  
    }  
}
```

**LISTAGEM 11.13 ShoppingCart.java (continuação)**

```
}

/**
 * @return int o número de itens no carrinho
 */
public int getNumberItems() {
    return items.size();
}

/**
 * recupera o item indexado
 * @param index o índice do item
 * @return Item o item no índice
 */
public Item getItem( int index ) {
    return (Item) items.get( index );
}

public Iterator iterator() {
    // ArrayList tem um método iterator() que retorna um iterador
    // entretanto, para propósitos de demonstração, ajuda ver um iterador
    // simples
    return new CartIterator( items );
}
}
```

11

**LISTAGEM 11.14 CartIterator.java**

```
public class CartIterator implements Iterator {

    private Object [] items;
    private int index;

    public CartIterator( java.util.LinkedList items ) {
        this.items = items.toArray();
    }

    public boolean isDone() {
        if(index >= items.length )
            return true;
    }
        return false;
    }

    public Object currentItem() {
        if( !isDone() ) {
```

**LISTAGEM 11.14** CartIterator.java (*continuação*)

```
        return items [index];
    }
    return null;
}

public void next() {
    index++;
}

public void first() {
    index = 0;
}
}
```

---

2. Tornando o adaptador mutante, você pode usar o mesmo empacotador para empacotar muitos objetos diferentes e não precisa instanciar um empacotador para cada objeto que precise ser empacotado. A reutilização de empacotadores faz melhor uso da memória e libera seu programa de ter de pagar o preço da instanciação de muitos empacotadores.

**LISTAGEM 11.15** MutableAdapter.java

```
public class MutableAdapter extends MoodyObject {

    private Pet pet;

    public MutableAdapter( Pet pet ) {
        setPet( pet );
    }

    protected String getMood() {
        // implementando apenas porque é exigido por
        // MoodyObject, como também sobrepuja queryMood
        // não precisamos dele
        return pet.speak();
    }

    public void queryMood() {
        System.out.println( getMood() );
    }

    public void setPet( Pet pet ) {
        this.pet = pet;
    }
}
```

---

# SEMANA 2

DIA **12**

## Padrões avançados de projeto

No capítulo de ontem, você viu como os padrões de projeto permitem reutilizar projetos testados. Hoje, você continuará seu estudo de padrões de projeto, examinando mais três padrões.

Hoje você aprenderá:

- Sobre três importantes padrões de projeto
- Como garantir que seus objetos permaneçam únicos
- Como aprimorar um exemplo anterior
- A respeito de algumas armadilhas comuns de padrão que você deve evitar

## Mais padrões por exemplo

Vamos continuar nosso estudo sobre padrões de projeto, considerando três padrões importantes:

- Abstract Factory
- Singleton
- Typesafe Enum

Cada um desses padrões aparece em quase todos os projetos. Na verdade, você pode usar o padrão Typesafe Enum para corrigir um exemplo do Capítulo 3, “Encapsulamento: hora de escrever algum código!”

## O padrão Abstract Factory

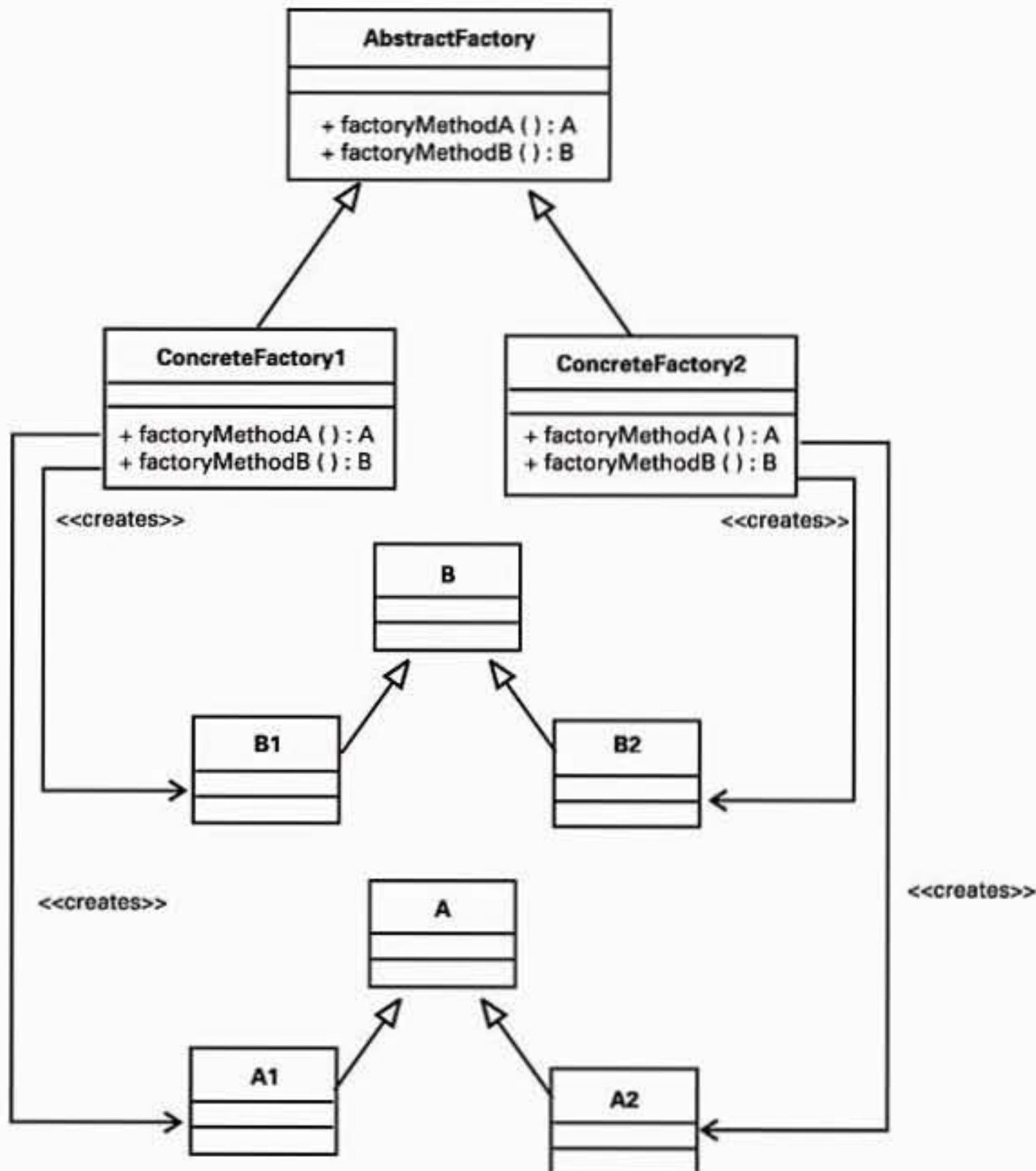
O Capítulo 7, “Polimorfismo: hora de escrever algum código”, mostrou como você pode combinar herança e polimorfismo para escrever software ‘à prova de futuro’. Os relacionamentos com capacidade de conexão da herança, combinados com o polimorfismo, permitem que você conecte novos objetos em seu programa, a qualquer momento; entretanto, há um inconveniente.

Para que seu programa possa instanciar esses novos objetos, você deve entrar no código e alterá-lo para que ele instancie os novos objetos, em vez dos antigos. (E você precisará fazer isso em todos os lugares onde os objetos antigos são instanciados!). Não seria ótimo se houvesse um modo mais fácil de conectar seus novos objetos?

O padrão Abstract Factory resolve esse problema através da delegação. Em vez de instanciar objetos através de seu programa, você pode delegar essa responsabilidade para um objeto chamado *factory*. Quando um objeto precisar criar outro, ele solicitará que o factory faça isso. Usando um factory, você pode isolar toda a criação de objeto em um único local. Quando você precisar introduzir novos objetos em seu sistema, só precisará atualizar o factory, para que ele crie uma instância de suas novas classes. Os objetos que usam o factory nunca saberão a diferença.

A Figura 12.1 ilustra o projeto geral do padrão Abstract Factory.

**FIGURA 12.1**  
*O padrão Abstract Factory.*



O padrão Abstract Factory usa herança e capacidade de conexão. A classe base de `factory` define todos os métodos de criação de objeto e cada subclasse `factory` define quais objetos cria, sobrepondo os métodos.

## Implementando um Abstract Factory

Existem ocasiões em que você precisará usar bibliotecas de terceiros dentro de seu programa. Infelizmente, quando é hora de migrar para uma nova versão, você pode verificar que as APIs públicas das bibliotecas mudaram ligeiramente. Felizmente, o padrão Abstract Factory oferece uma solução que torna tranquila a migração da biblioteca.

Imagine que você esteja trabalhando em um projeto que usa um analisador XML. Um analisador XML receberá um documento XML como String e retornará uma representação de objeto do documento, conhecida como `Document`. Se você sabe que vai atualizar a biblioteca no futuro e que a API vai mudar, existem alguns passos que pode dar para se proteger.

### XML

XML, ou Extensible Markup Language, é uma linguagem para escrever tags que descrevem dados. Assim como a HTML oferece tags para formatar e exibir seus dados, a XML permite que você defina suas próprias tags personalizadas para descrever o significado conceitual de seus dados.

A HTML é uma maneira excelente de marcar dados para exibição. Entretanto, a HTML não tem a capacidade de transmitir o significado dos dados. A HTML pode dizer para que você coloque negrito em uma palavra, mas não pode dizer que a palavra negritada é o título do documento. Em vez disso, você precisa aplicar esse significado externamente ao documento.

Por outro lado, a XML fornece um mecanismo que permite dizer que alguma palavra em um documento é um título. Se você usar essas tags para marcar seus dados, diferentes programas poderão ler seus documentos e saber o que significa cada dado. Assim, por exemplo, você pode escrever um programa que sabe formatar títulos com negrito. Ou então, você pode escrever outro programa que lê uma lista de documentos e formula uma lista de títulos para sua seleção. Tentar escrever os mesmos programas para ler um documento HTML seria muito mais difícil.

Nos últimos anos, a XML se tornou um padrão importante para a troca de dados. Duas entidades não relacionadas podem se comunicar, desde que ambas entendam as tags XML. Como resultado, a XML tem aparecido como o padrão para comunicação entre empresas. Para comprar e vender, duas empresas podem concordar com um conjunto de tags comuns. Uma vez que tenham essas tags, elas podem trocar documentos XML livremente.

Considere o seguinte documento XML de uma receita:

```
<Recipe>
    <Name>Chicken Tacos</Name>
    <Ingredients>
        <Ingredient>
            <Name>Chicken</Name>
```

```

<Quantity UOM="1b ">1</Quantity>
</Ingredient>

</Ingredients>
</Recipe>

```

Esse documento XML tem tags que descrevem os dados de uma receita. Se você entende a marcação de tag, sabe que os dados que estão entre as tags <Name> representam o nome da receita. Todos os dados que estão entre as tags <Ingredients> contêm informações de ingredientes.

Você pode escrever programas que reconhecem a marcação Recipe. Um programa poderia permitir que você selecionasse receitas que gostaria de fazer durante uma semana. Esse programa poderia usar os documentos Recipe que você escolhesse para formular e imprimir uma lista de compras. Outro programa poderia imprimir o menu da semana.

Para que um programa leia e entenda um documento XML, ele deve analisar o documento. Um analisador de XML pegará um documento e o converterá em uma árvore de objetos. Cada objeto conterá parte do documento. Seu programa pode simplesmente percorrer esses objetos para extrair informações do documento. Assim, se você escrever um programa que leia receitas, ele poderá percorrer todos os objetos Ingredient para construir uma lista de ingredientes.

Atualmente, muitos analisadores de XML estão disponíveis para escolha. Cada analisador tem uma API ligeiramente diferente, de modo que, uma vez que você escreva um programa para usar um analisador específico, poderá ser difícil trocar de analisador posteriormente. O padrão Abstract Factory apresenta uma maneira de evitar que você fique preso ao uso de um analisador em particular.

**Novo TERMO**

Um *analisador de XML* pega um documento XML e o transforma em uma representação de objeto.

Inicialmente, você precisa empacotar o analisador em um objeto que controle.



Lembre-se de que um *empacotador* é um adaptor. Um *empacotador* converte a interface de um objeto em uma interface alternativa. Normalmente, você usa um *empacotador* para converter a interface em uma interface esperada por seu programa.

Como você controla a API do empacotador, garante uma API estável que seu programa pode usar. A Listagem 12.1 ilustra um possível empacotador analisador.

**LISTAGEM 12.1** Parser.java

```

public interface Parser {
    public org.w3c.dom.Document parse( String document );
}

```

**NOTA**

`org.w3c.dom.Document` é uma interface definida pelo W3C para representar um documento XML como uma estrutura de objetos. Você pode encontrar mais informações sobre a interface Document no endereço <http://www.w3.org/TR/2000/CR-DOM-Level-2-20000510/java-binding.html>.

As listagens 12.2 e 12.3 mostram duas possíveis implementações: `VersionOneParser`, que usa a versão 1.0 da biblioteca, e `VersionTwoParser`, que usa a versão 2.0 da biblioteca.

**LISTAGEM 12.2** VersionOneParser.java

```
public class VersionOneParser implements Parser {  
  
    public org.w3c.dom.Document parse( String document ) {  
        // instancia a versão 1 do analisador  
        // XMLParser p = new XMLParser();  
        // passa o documento para o analisador e retorna o resultado  
        // return p.parseXML( document );  
    }  
}
```

**LISTAGEM 12.3** VersionTwoParser.java

```
public class VersionTwoParser implements Parser {  
  
    public org.w3c.dom.Document parse( String document ) {  
        // instancia a versão 2 do analisador  
        // DOMParser parser = new DOMParser();  
        // passa o documento para o analisador e retorna o resultado  
        // return parser.parse( document );  
    }  
}
```

12

Seu programa pode usar o padrão Abstract Factory para criar a versão correta do analisador, sempre que ele precisar analisar um documento. A Listagem 12.4 apresenta a interface base de factory.

**LISTAGEM 12.4** ParserFactory.java

```
public interface ParserFactory {  
    public Parser createParser();  
}
```

Você precisará de duas implementações concretas de factory, pois existem duas implementações do analisador. As listagens 12.5 e 12.6 apresentam essas implementações.

**LISTAGEM 12.5** VersionOneParserFactory.java

```
public class VersionOneParserFactory implements ParserFactory {  
    public Parser createParser() {  
        return new VersionOneParser();  
    }  
}
```

---

**LISTAGEM 12.6** VersionTwoParserFactory.java

```
public class VersionTwoParserFactory implements ParserFactory {  
    public Parser createParser() {  
        return new VersionTwoParser();  
    }  
}
```

---

Agora, em vez de instanciar analisadores diretamente, seu programa pode usar uma das classes factory para recuperar objetos analisadores, quando precisar analisar um documento. Você precisará simplesmente instanciar a factory correta no início do programa e torná-la disponível para os objetos de seu programa.

O padrão Factory Method está intimamente relacionado ao padrão Abstract Factory. Na verdade, um Abstract Factory pode usar Factory Method para criar os objetos que retorna.

Um método Factory nada mais é do que um método que cria objetos. `createParser()` é um exemplo de método Factory. Você também pode encontrar exemplos de métodos Factory em toda a API Java.

`Class.newInstance()` é um exemplo de método Factory.

Conforme você pode ver, um método Factory pode aparecer em uma classe normal ou em uma Factory abstrata. Em qualquer caso, ele cria objetos, ocultando assim a classe real do objeto criado.

## Quando usar o padrão Abstract Factory

Use o padrão Abstract Factory, quando:

- Você quiser ocultar o modo como um objeto é criado.
- Você quiser ocultar a classe atual do objeto criado.
- Você quiser um conjunto de objetos usados juntos. Isso evita que você use objetos incompatíveis juntos.
- Você quiser usar diferentes versões de uma implementação de classe. Um Abstract Factory permite que você troque essas diferentes versões em seu sistema.

A Tabela 12.1 destaca o usuário do padrão Abstract Factory.

**TABELA 12.1** O padrão Abstract Factory

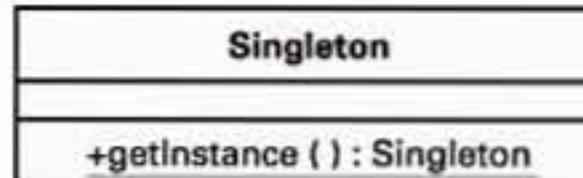
<i>Nome do padrão</i>	Abstract Factory
<i>Problema</i>	Precisa de uma maneira de trocar objetos plugáveis de forma transparente
<i>Solução</i>	Fornecer uma interface abstrata que providencie métodos para instanciar os objetos
<i>Conseqüências</i>	Permite que você troque facilmente novos tipos de classe em seu sistema; entretanto, é dispendioso adicionar tipos não relacionados

## O padrão Singleton

Quando projetar seus sistemas, você verá que algumas classes deveriam ter logicamente apenas uma instância, como um factory ou um objeto que acesse algum recurso não compartilhado (conexão de banco de dados, região de memória etc.). Nada, entretanto, impedirá que um objeto instancie outro. Como você impõe seu projeto?

O padrão Singleton fornece a resposta dessa pergunta. O padrão Singleton impõe seu projeto colocando a responsabilidade da criação e da intermediação do acesso à instância no próprio objeto. Fazer isso garante que apenas uma instância seja criada, além de fornecer um único ponto de acesso para essa instância. A Figura 12.2 ilustra a assinatura de uma classe singleton (carta única de determinado naipe).

**FIGURA 12.2**  
*O padrão Singleton*



A Listagem 12.7 ilustra uma possível classe singleton.

**LISTAGEM 12.7** Uma implementação do padrão Singleton

```

public class Singleton {

    // uma referência de classe para a instância singleton
    private static Singleton instance;

    // o construtor deve ser oculto para que os objetos não possam instanciar
    // protected permite que outras classes herdem de Singleton
    protected Singleton() {}

    // um método de classe usado para recuperar a instância singleton
    public static Singleton getInstance() {
        if( instance == null ) {
            instance = new Singleton();
        }
    }
}
  
```

**LISTAGEM 12.7** Uma implementação do padrão Singleton (*continuação*)

```
    return instance;
}
}
```

A classe Singleton contém uma instância estática de Singleton e dá acesso à instância de Singleton através do método da classe getInstance().

## Implementando um Singleton

O Capítulo 7, Laboratório 1, apresentou uma classe Payroll. Uma classe de folha de pagamento ‘real’ provavelmente acessaria um banco de dados de funcionários. Poderia ser uma boa idéia ter apenas uma instância de Payroll, para evitar conflitos de recurso. A classe Payroll é uma boa candidata para o padrão Singleton.

A Listagem 12.8 apresenta um singleton Payroll.

**LISTAGEM 12.8** Payroll Singleton

```
public class Payroll {

    // uma referência de classe para a instância singleton única
    private static Payroll instance;

    private int      total_hours;
    private int      total_sales;
    private double   total_pay;

    // oculta o construtor para que outros objetos não possam instanciar
    protected Payroll() {}

    // observe o uso de static; você não tem uma instância, quando recupera
    // uma instância; portanto, o método deve ser um método de classe; daí,    //
    static
    public static Payroll getInstance() {
        if( instance == null ) {
            instance = new Payroll();
        }
        return instance;
    }

    public void payEmployees( Employee [] emps ) {
        for( int i = 0; i < emps.length; i ++ ) {
            Employee emp = emps [i];
            total_pay += emp.calculatePay();
            emp.printPaycheck();
        }
    }
}
```

**LISTAGEM 12.8 Payroll Singleton (*continuação*)**

```
}

public void calculateBonus( Employee [] emps ) {
    for( int i = 0; i < emps.length; i ++ ) {
        Employee emp = emps [i];
        System.out.println("Pay bonus to " + emp.getLastName() + ", " +
            emp.getFirstName() + " $" + emp.calculateBonus() );
    }
}

public void recordEmployeeInfo( CommissionedEmployee emp ) {
    total_sales += emp.getSales();
}

public void recordEmployeeInfo( HourlyEmployee emp ) {
    total_hours += emp.getHours();
}

public void printReport() {
    System.out.println( "Payroll Report:" );
    System.out.println( "Total Hours: " + total_hours );
    System.out.println( "Total Sales: " + total_sales );
    System.out.println( "Total Paid: $" + total_pay );
}
}
```

O singleton Payroll adiciona um método `getInstance()`. Esse método é responsável por criar e dar acesso à instância singleton. Preste bem atenção ao construtor; aqui, o construtor é protegido para que outros objetos não possam inadvertidamente instanciar mais objetos Payroll. Como ele é protegido, outros objetos podem herdar de Payroll.

A Listagem 12.9 dá um exemplo de como você pode usar o singleton.

**LISTAGEM 12.9 Usando o singleton Payroll**

```
// recupera o singleton de folha de pagamento
Payroll payroll = Payroll.getInstance();

// cria e atualiza alguns funcionários
CommissionedEmployee emp1 = new CommissionedEmployee( "Mr.", "Sales", 25000.00,
1000.00);
CommissionedEmployee emp2 = new CommissionedEmployee( "Ms.", "Sales", 25000.00,
1000.00);
emp1.addSales( 7 );
```

**LISTAGEM 12.9** Usando o singleton Payroll (*continuação*)

```
emp2.addSales( 5 );

HourlyEmployee emp3 = new HourlyEmployee( "Mr.", "Minimum Wage", 6.50 );
HourlyEmployee emp4 = new HourlyEmployee( "Ms.", "Minimum Wage", 6.50 );
emp3.addHours( 40 );
emp4.addHours( 46 );

// usa os métodos sobrecarregados
payroll.recordEmployeeInfo( emp2 );
payroll.recordEmployeeInfo( emp1 );
payroll.recordEmployeeInfo( emp3 );
payroll.recordEmployeeInfo( emp4 );
```

---

Uma vez que você tenha uma instância singleton, ela funcionará como qualquer outra instância. O que é interessante a respeito da Listagem 12.9 é:

```
Payroll payroll = Payroll.getInstance();
```

Note que você não escreve mais:

```
Payroll payroll = new Payroll();
```

## Herança e o padrão Singleton

O padrão Singleton apresenta algumas dificuldades de herança. Especificamente, quem gerencia a instância singleton, a progenitora ou a filha? Você tem algumas escolhas.

A primeira escolha é simplesmente criar a subclasse singleton e atualizar a progenitora para instanciar a filha. As listagens 12.10 e 12.11 destacam essa estratégia.

**LISTAGEM 12.10** ChildSingleton

```
public class ChildSingleton extends Singleton {

    protected ChildSingleton() {}

    public String toString() {
        return "I am the child singleton ";
    }
}
```

---

**LISTAGEM 12.11** Updated Singleton

```
public class Singleton {

    // uma referência de classe para a instância singleton
```

**LISTAGEM 12.11 Updated Singleton (continuação)**

```
private static Singleton instance;

// o constructor deve ser oculto para que os objetos não possam instanciar
// protected permite que outras classes herdem de Singleton
protected ParentSingleton() {}

// um método de classe usado para recuperar a instância singleton
public static Singleton getInstance() {
    if( instance == null ) {
        instance = new ChildSingleton();
    }
    return instance;
}

public String toString() {
    return "I am the singleton";
}
}
```

Essa solução tem o inconveniente de exigir alterações na classe progenitora. Uma solução alternativa inclui fazer com que o singleton leia uma variável de configuração na primeira vez que `getInstance()` for chamado. O singleton pode instanciar qualquer objeto que seja especificado no valor da configuração.

Você também pode permitir que cada filha forneça sua própria implementação de `getInstance()`. Essa estratégia não exigirá alterações na progenitora.

**12**

Este quadro conta com um truque da linguagem Java, embora exista um análogo na linguagem C++. Trata-se de uma solução interessante que tira proveito do fato de que os blocos estáticos são executados quando uma classe é carregada na linguagem Java.

Através de outra estratégia, você pode adicionar um método protegido `register( Singletons )` na progenitora. Você pode colocar um bloco estático na filha, que a instancie. Dentro do construtor, a filha pode se registrar na singleton progenitora.

Aqui está o código da classe Singleton atualizada:

```
public class Singleton {

    // uma referência de classe para a instância de singleton
    private static Singleton instance;

    // o constructor deve ser oculto para que os objetos não possam instanciar
    // protected permite que outras classes herdem de Singleton
    protected Singleton() {}
```

```
// um método de classe usado para recuperar a instância de singleton
public static Singleton getInstance() {
    if( instance == null ) {
        //valor padrão
        instance = new Singleton();
    }
    return instance;
}

protected static void register( Singleton s ) {
    if( instance == null ) {
        instance = s;
    }
}
}
```

E a classe ChildSingleton atualizada:

```
public class ChildSingleton extends Singleton {

    static {
        new ChildSingleton();
    }

    protected ChildSingleton() {
        Singleton.register( this );
    }
}
```

Para fazer tudo isso funcionar, você precisará chamar `Class.forName( "ChildSingleton" )`

Antes de chamar o método `getInstance()` do singleton. Aqui está um exemplo:

```
Class.forName( "ChildSingleton" );
Singleton s = Singleton.getInstance();
System.out.println( s.toString() );
```

## Quando usar o padrão Singleton

Use o padrão Singleton quando você quiser restringir uma classe a ter apenas uma instância.

A Tabela 12.2 destaca o usuário do padrão Singleton.

**TABELA 12.2** O padrão Singleton

<i>Nome do padrão</i>	Singleton
<i>Problema</i>	Deve existir apenas uma instância de um objeto no sistema em determinado momento.
<i>Solução</i>	Permitir que o objeto gerencie sua própria criação e acesso através de um método de classe.

**TABELA 12.2** O padrão Singleton (*continuação*)

<i>Consequências</i>	Acesso controlado à instância do objeto. Também pode dar acesso a um número definido de instâncias (como apenas seis instâncias), com uma leve alteração no padrão. É um pouco mais difícil herdar um singleton.
----------------------	--

## O padrão Typesafe Enum

A Listagem 12.12 lista uma seleção da classe Card, apresentada pela primeira vez no Capítulo 3, Laboratório 3.

**LISTAGEM 12.12** Uma seleção de Card.java

```
public class Card {  
  
    private int rank;  
    private int suit;  
    private boolean face_up;  
  
    // constantes usadas para instanciar  
    // naipes  
    public static final int DIAMONDS = 4;  
    public static final int HEARTS = 3;  
    public static final int SPADES = 6;  
    public static final int CLUBS = 5;  
    // valores  
    public static final int TWO = 2;  
    public static final int THREE = 3;  
    public static final int FOUR = 4;  
    public static final int FIVE = 5;  
    public static final int SIX = 6;  
    public static final int SEVEN = 7;  
    public static final int EIGHT = 8;  
    public static final int NINE = 9;  
    public static final int TEN = 10;  
    public static final int JACK = 74;  
    public static final int QUEEN = 81;  
    public static final int KING = 75;  
    public static final int ACE = 65;  
  
    // cria uma nova carta - usa as constantes apenas para inicializar  
    public Card( int suit, int rank ) {  
        // Em um programa real, você precisaria fazer a validação nos argumentos.  
  
        this.suit = suit;  
        this.rank = rank;  
    }  
}
```

Ao instanciar objetos Card, você precisa passar uma constante de número e de naipe válida. A utilização de constantes dessa maneira pode levar a muitos problemas. Nada o impede de passar qualquer int que você quiser. E, embora você deva referenciar apenas o nome da constante, isso abre a representação interna de Card. Por exemplo, para conhecer o naipe da carta (Card), você deve recuperar o valor int e depois compará-lo com as constantes. Embora isso funcione, essa não é uma solução limpa.

O problema reside no fato de que naipe e número são objetos propriamente ditos. int não resolve isso, pois você precisa aplicar um significado a int. Novamente, isso confunde a responsabilidade, pois você precisará reaplicar esse significado sempre que encontrar um int que represente um número ou um naipe.

Linguagens como C++ têm uma construção, conhecida como *enumeração*; entretanto, as enumerações se reduzem simplesmente a um atalho para declarar uma lista de constantes inteiras. Essas constantes são limitadas. Por exemplo, elas não podem fornecer comportamento. Também é difícil adicionar mais constantes.

Em vez disso, o padrão Typesafe Enum fornece uma maneira OO de declarar suas constantes. Em vez de declarar simples constantes inteiras, você cria classes para cada tipo de constante. Para o exemplo Card, você criaria uma classe Rank (número) e uma classe Suit (naipe). Então, você criaria uma instância para cada valor de constante que quisesse representar e a tornaria publicamente disponível a partir da classe (através de public final, exatamente como as outras constantes).

## Implementando o padrão Typesafe Enum

Vamos ver a implementação das classes Rank e Suit, nas listagens 12.13 e 12.14.

---

### LISTAGEM 12.13 Suit.java

```
public final class Suit {  
  
    //define estaticamente todos os valores válidos de Suit  
    public static final Suit DIAMONDS = new Suit( (char)4 );  
    public static final Suit HEARTS   = new Suit( (char)3 );  
    public static final Suit SPADES   = new Suit( (char)6 );  
    public static final Suit CLUBS   = new Suit( (char)5 );  
  
    // ajuda a fazer a iteração pelos valores da enumeração  
    public static final Suit [] SUIT  = { DIAMONDS, HEARTS, SPADES, CLUBS };  
  
    // variável de instância para conter o valor de exibição  
    private final char display;  
  
    // não permite instanciação por objetos externos  
    private Suit( char display ) {
```

**LISTAGEM 12.13** Suit.java (*continuação*)

```
    this.display = display;  
}  
  
// retorna o valor de Suit  
public String toString() {  
    return String.valueOf( display );  
}  
}
```

Suit é simples. O construtor recebe um char que representa o Suit. Como Suit é um objeto completo, ele também pode ter métodos. Aqui, o Suit fornece um método `toString()`. Uma enumeração de typesafe que pode adicionar quaisquer métodos que se mostrem úteis.

Você também notará que a constante é privada. Isso impede que os objetos instanciem objetos Suit diretamente. Em vez disso, você está restrito a usar apenas as instâncias de constante declaradas pela classe. A classe também é declarada como final para que outras classes não possam ser subclasses dela. Existem ocasiões em que você permitirá a herança. Nessas ocasiões, torne o construtor protegido e remova a declaração final.



**NOTA** Devido ao modo como a linguagem Java funciona, certifique-se de fornecer versões finais de `equals()` e `hashCode()` que chamem super, caso você abra sua enumeração para a herança. Se não, você estará aberto a problemas estranhos, caso suas subclasses redefinam esses métodos incorretamente.

Você notará que a classe Suit define várias instâncias de constante, uma para cada um dos naipes válidos. Quando você precisa de um valor constante de Suit, pode escrever `Suit.DIAMONDS`.

A classe Rank, na Listagem 12.14, funciona de modo semelhante a Suit; entretanto, ela acrescenta mais alguns métodos. O método `getRank()` retorna o valor de Rank. Esse valor pode ser importante para calcular o valor de uma jogada. Ao contrário das constantes originais, você não precisa mais aplicar significado às constantes Rank ou Suit. Em vez disso, elas contêm seus próprios significados, pois são objetos.

**LISTAGEM 12.14** Rank.java

```
public final class Rank {  
  
    public static final Rank TWO   = new Rank( 2, "2" );  
    public static final Rank THREE = new Rank( 3, "3" );  
    public static final Rank FOUR  = new Rank( 4, "4" );  
    public static final Rank FIVE  = new Rank( 5, "5" );  
    public static final Rank SIX   = new Rank( 6, "6" );  
    public static final Rank SEVEN = new Rank( 7, "7" );
```

**LISTAGEM 12.14** Rank.java (*continuação*)

```
public static final Rank EIGHT = new Rank( 8, "8" );
public static final Rank NINE  = new Rank( 9, "9" );
public static final Rank TEN   = new Rank( 10, "10" );
public static final Rank JACK  = new Rank( 11, "J" );
public static final Rank QUEEN = new Rank( 12, "Q" );
public static final Rank KING  = new Rank( 13, "K" );
public static final Rank ACE   = new Rank( 14, "A" );

public static final Rank [] RANK =
{ TWO, THREE, FOUR, FIVE, SIX, SEVEN,
  EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE };

private final int    rank;
private final String display;

private Rank( int rank, String display ) {
    this.rank = rank;
    this.display = display;
}

public int getRank() {
    return rank;
}

public String toString() {
    return display;
}
}
```

---

Por exemplo, você não precisa mais aplicar significado a `int 4` e sabe que 4 significa DIAMONDS. Quando você precisar determinar o valor da constante, pode fazer comparações de objeto usando `equals()`.

A Listagem 12.15 mostra as alterações que você precisaria fazer em `Card` para poder usar as novas constantes. (As classes `Deck` e `Dealer` atualizadas estão disponíveis no código-fonte deste capítulo).

**LISTAGEM 12.15** A classe Card.java atualizada

```
public class Card {

    private Rank rank;
    private Suit suit;
    private boolean face_up;
```

**LISTAGEM 12.15** A classe Card.java atualizada (*continuação*)

```
// cria uma nova carta - usa as constantes apenas para inicializar
public Card( Suit suit, Rank rank ) {
    // Em um programa real, você precisaria fazer a validação nos argumentos.
    this.suit = suit;
    this.rank = rank;
}

public Suit getSuit() {
    return suit;
}

public Rank getRank() {
    return rank;
}

public void faceUp() {
    face_up = true;
}

public void faceDown() {
    face_up = false;
}

public boolean isFaceUp() {
    return face_up;
}

public String display() {
    return rank.toString() + suit.toString();
}
```

12

Você também pode ter notado que Rank e Suit declaram arrays de constantes. Isso torna fácil fazer um laço pelos valores das constantes disponíveis. A Listagem 12.16 mostra como esse fato simplifica bastante o método buildCards() de Deck.

**LISTAGEM 12.16** O método buildCards() atualizado

```
private void buildCards() {

    deck = new java.util.LinkedList();

    for( int i = 0; i < Suit.SUIT.length; i ++ ) {
        for( int j = 0; j < Rank.RANK.length; j ++ ) {
```

**LISTAGEM 12.16** O método buildCards() atualizado (*continuação*)

```
        deck.add( new Card( Suit.SUIT [i], Rank.RANK [j] ) );  
    }  
}  
}
```

---

## Quando usar o padrão Typesafe Enum

Use o padrão Typesafe Enum, quando:

- Você se achar escrevendo numerosas primitivas públicas ou constantes de String.
- Você se achar impondo identidade em um valor, em vez de derivar a identidade do próprio valor. A Tabela 12.3 destaca o usuário do padrão Typesafe Enum.

**TABELA 12.3** O padrão Typesafe Enum

<i>Nome do padrão</i>	Typesafe Enum.
<i>Problema</i>	As constantes inteiras são limitadas.
<i>Solução</i>	Criar uma classe para cada tipo de constante e depois fornecer instâncias de constante para cada valor de constante.
<i>Conseqüências</i>	Constantes OO extensíveis. Constantes úteis que têm comportamento. Você ainda precisa atualizar código para usar as novas constantes, quando elas forem adicionadas. Exige mais memória do que uma constante simples.

---

## Armadilhas do padrão

Você pode abusar dos padrões de projeto, assim como acontece com qualquer outra ferramenta, usando-os incorretamente. Os padrões de projeto não garantem um bom projeto; na verdade, incluir um padrão em um lugar onde ele realmente não pertence arruinará seu projeto. Você precisa ser criterioso em sua decisão de incluir um padrão em seu projeto.

Recentemente, os padrões de projeto vêm sendo cada vez mais criticados. Infelizmente, existe uma tendência, especialmente entre os iniciantes, de ser pego na tentativa de aplicar o máximo de padrões possível a um projeto. O entusiasmo de usar padrões tem feito muitos desenvolvedores se esquecerem do objetivo dos padrões e até do próprio processo de projeto. Não caia nessa armadilha! Não fique cego com o prazer de aplicar padrões e deixe o projeto se reduzir ao máximo de padrões possível. Você não ganhará pontos de seus colegas por usar o máximo de padrões, mas ganhará pontos por produzir um projeto limpo e coerente. Tal projeto poderia nem mesmo usar padrões!

**DICA**

Existem algumas diretrizes que o ajudarão a evitar a armadilha do padrão:

Dica 1: colocando parafusos redondos em buracos quadrados. Se você se achar pensando, "como posso usar <insira seu padrão predileto aqui> em meu projeto?", estará com problemas. Em vez disso, você deve pensar, "já vi esse projeto antes; acho que existe um padrão que o resolve". Agora, vá e veja um livro sobre padrões. Sempre comece do ponto de vista do problema e não da solução (o padrão).

Dica 2: ataques de amnésia. Você estará com problemas, caso não possa explicar, em duas frases ou menos, porquê escolheu um padrão e quais as vantagens que ele oferece. Você deve poder explicar facilmente porque incluiu um padrão e para que ele contribui em um projeto. A situação é sem esperança, caso você não consiga pensar em uma explicação.

Existe uma segunda armadilha, mais sutil, nos padrões: *tagarelice do padrão*. Não use padrões para tentar parecer inteligente e não tente sugerir o uso de um padrão que você não tenha estudado. Você poderia mencionar o padrão, mas seja claro, caso não esteja familiarizado com ele. Você não deve apenas usar padrões apropriadamente em seu projeto, mas também em suas conversas. Não é uma boa idéia contribuir para os fatores que prejudicam a prática de usar padrões.

## Resumo

Os padrões de projeto são uma ajuda útil ao se projetar suas soluções. De sua própria maneira, os padrões são a consciência coletiva da comunidade de OO, que tem anos de experiência em projeto.

Lembre-se dos limites dos padrões de projeto, quando utilizá-los. Um padrão de projeto trata de um e apenas um problema abstrato. Um padrão de projeto não fornece a solução para um problema específico. Em vez disso, o padrão fornece uma solução abstrata para um problema geral. Fica por sua conta fornecer o mapeamento entre o problema abstrato e seu problema específico.

Mapear um padrão de projeto provavelmente é o maior desafio que você enfrentará ao usar padrões. Trata-se de uma habilidade que só vem com o tempo, estudo e prática.

12

## Perguntas e respostas

**P Como você escolhe um padrão de projeto?**

**R** Cada padrão de projeto tem um problema e padrões relacionados. Estude o padrão, se a descrição do problema parece corresponder ao seu caso. Também ajudará, se você examinar todos os padrões relacionados. Se, após estudar o padrão, ele parecer resolver seu problema, tente aplicar o padrão ao seu caso. Certifique-se de examinar as consequências. Se qualquer uma das consequências entrar em conflito com seus requisitos, você provavelmente deverá ignorar o padrão.

**P Como você sabe quando deve usar um padrão de projeto?**

**R** Não há uma resposta fácil para essa pergunta.

Você não pode usar um padrão, caso não o conheça, e ninguém pode conhecer todos os padrões de projeto disponíveis. Quando você projetar, tente obter o máximo de entradas possível. Pergunte às pessoas se elas conhecem padrões que possam ajudá-lo a tomar suas decisões de projeto.

Estude os padrões. Quanto mais padrões você conhecer, mais oportunidades verá para usá-los.

**P A linguagem Java usa quaisquer dos padrões abordados hoje?**

**R** Sim. A linguagem Java usa muitos dos padrões abordados hoje.

Padrão Factory Method: muitas classes Java têm métodos factory (um padrão intimamente relacionado ao padrão Abstract Factory).

Padrão Singleton: `java.lang.System` é um exemplo de singleton na linguagem Java.

Padrão Typesafe Enum: o padrão Typesafe Enum ainda não estava definido quando muitas das APIs Java foram criadas. As adições futuras na API Java usarão o padrão Typesafe Enum.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. O que é uma classe empacotadora?
2. Qual problema o padrão Abstract Factory resolve?
3. Por que você usaria o padrão Abstract Factory?
4. Qual problema o padrão Singleton resolve?
5. Por que você usaria o padrão Singleton?
6. Qual problema o padrão Typesafe Enum resolve?
7. Por que você usaria o padrão Typesafe Enum?
8. Os padrões garantem um projeto perfeito? Por que sim ou por que não?

## Exercícios

A Listagem 12.17 apresenta a classe Bank do Capítulo 7. Transforme Bank em um singleton.

**LISTAGEM 12.17** Bank.java

```
public class Bank {  
  
    private java.util.Hashtable accounts = new java.util.Hashtable();  
  
    public void addAccount( String name, BankAccount account ) {  
        accounts.put( name, account );  
    }  
  
    public double totalHoldings() {  
        double total = 0.0;  
  
        java.util.Enumeration enum = accounts.elements();  
        while( enum.hasMoreElements() ) {  
            BankAccount account = (BankAccount) enum.nextElement();  
            total += account.getBalance();  
        }  
        return total;  
    }  
  
    public int totalAccounts() {  
        return accounts.size();  
    }  
  
    public void deposit( String name, double amount ) {  
        BankAccount account = retrieveAccount( name );  
        if( account != null ) {  
            account.depositFunds( amount );  
        }  
    }  
  
    public double balance( String name ) {  
        BankAccount account = retrieveAccount( name );  
        if( account != null ) {  
            return account.getBalance();  
        }  
        return 0.0;  
    }  
  
    private BankAccount retrieveAccount( String name ) {  
        return (BankAccount) accounts.get( name );  
    }  
}
```

12

2. Considere a classe Error apresentada na Listagem 12.18. Essa classe define várias constantes. Aplique o padrão Typesafe Enum no projeto dessa classe.

**LISTAGEM 12.18 Error.java**

```
public class Error {  
  
    // níveis de erro  
    public final static int NOISE    = 0;  
    public final static int INFO     = 1;  
    public final static int WARNING  = 2;  
    public final static int ERROR    = 3;  
    private int level;  
  
    public Error( int level ) {  
        this.level = level;  
    }  
  
    public int getLevel() {  
        return level;  
    }  
  
    public String toString() {  
        switch (level){  
            case 0: return "NOISE ";  
            case 1: return "INFO ";  
            case 2: return "WARNING ";  
            default: return "ERROR ";  
        }  
    }  
}
```

---

3. Projete e crie um Abstract Factory para a hierarquia BankAccount apresentada como uma solução no Capítulo 7, Laboratório 3.

## Respostas do teste

1. Uma classe empacotadora transforma a interface de um objeto naquela esperada por seu programa. Uma classe empacotadora contém um objeto e delega mensagens da nova interface para a interface do objeto contido.
2. O padrão Abstract Factory fornece um mecanismo que instancia instâncias de classe descendentes específicas, sem revelar qual descendente é realmente criado. Isso permite que você conecte de forma transparente diferentes descendentes em seu sistema.
3. Você usa o padrão Abstract Factory para ocultar os detalhes da instanciação, para ocultar qual classe de objeto é instanciada e quando quer que um conjunto de objetos sejam usados juntos.
4. O padrão Singleton garante que um objeto seja instaciado apenas uma vez.

5. Você usa o padrão Singleton quando quer que um objeto seja instanciado apenas uma vez.
6. Usar constantes primitivas não é uma estratégia de OO para programação, pois você precisa aplicar um significado externo à constante. Você viu quantos problemas a decomposição da responsabilidade poderia causar!

O padrão Typesafe Enum resolve esse problema, transformando a constante em um objeto de nível mais alto. Usando um objeto de nível mais alto, você pode encapsular melhor a responsabilidade dentro do objeto constante.

7. Você deve usar o padrão Typesafe Enum quando se achar declarando constantes públicas que devem ser objetos propriamente ditos.
8. Não, os padrões não garantem um projeto perfeito, pois você poderia acabar usando um padrão incorretamente. Além disso, usar corretamente um padrão não significa que o restante de seu projeto seja válido. Muitos projetos válidos nem mesmo contêm um padrão.

## Respostas dos exercícios

1.

### LISTAGEM 12.19 Bank.java

```
public class Bank {  
  
    private java.util.Hashtable accounts = new java.util.Hashtable();  
  
    private static Bank instance;  
  
    protected Bank() {}  
  
    public static Bank getInstance() {  
        if( instance == null ) {  
            instance = new Bank();  
        }  
        return instance;  
    }  
  
    public void addAccount( String name, BankAccount account ) {  
        accounts.put( name, account );  
    }  
  
    public double totalHoldings() {  
        double total = 0.0;  
  
        java.util.Enumeration enum = accounts.elements();  
        while( enum.hasMoreElements() ) {  
            BankAccount account = (BankAccount)enum.nextElement();  
            total += account.getBalance();  
        }  
    }  
}
```

**LISTAGEM 12.19** Bank.java (*continuação*)

```
    }
    return total;
}

public int totalAccounts() {
    return accounts.size();
}

public void deposit( String name, double amount ) {
    BankAccount account = retrieveAccount( name );
    if( account != null ) {
        account.depositFunds( amount );
    }
}

public double balance( String name ) {
    BankAccount account = retrieveAccount( name );
    if( account != null ) {
        return account.getBalance();
    }
    return 0.0;
}

private BankAccount retrieveAccount( String name ) {
    return (BankAccount) accounts.get( name );
}
}
```

---

2.

**LISTAGEM 12.20** Level.java

```
public final class Level {

    public final static Level NOISE    = new Level( 0, "NOISE" );
    public final static Level INFO     = new Level( 1, "INFO" );
    public final static Level WARNING  = new Level( 2, "WARNING" );
    public final static Level ERROR    = new Level( 3, "ERROR" );

    private int level;
    private String name;

    private Level( int level, String name ) {
        this.level = level;
        this.name  = name;
    }
}
```

**LISTAGEM 12.20** Level.java (*continuação*)

```
}

public int getLevel() {
    return level;
}

public String getName() {
    return name;
}
}
```

**LISTAGEM 12.21** Error.java

```
public class Error {

    private Level level;

    public Error( Level level ) {
        this.level =level;
    }

    public Level getLevel() {
        return level;
    }

    public String toString() {
        return level.getName();
    }
}
```

12

3. A solução consiste em uma abstrata Factory de conta bancária (escrita como uma interface; entretanto, ela também pode ser uma classe abstrata) e uma Concreta Factory de conta bancária. A Factory tem um método para criar cada tipo de conta bancária.

Essa Factory oculta os detalhes da instanciação e não necessariamente o subtipo do objeto.

**LISTAGEM 12.22** AbstractAccountFactory.java

```
public interface AbstractAccountFactory {

    public CheckingAccount createCheckingAccount( double initDeposit, int trans,
double fee );
```

**LISTAGEM 12.22 AbstractAccountFactory.java (continuação)**

---

```
    public OverdraftAccount createOverdraftAccount( double initDeposit, double
rate );

    public RewardsAccount createRewardsAccount( double initDeposit, double
interest, double min );

    public SavingsAccount createSavingsAccount( double initBalance, double
interestRate );

    public TimedMaturityAccount createTimedMaturityAccount( double initBalance,
double interestRate, double feeRate );

}
```

---

**LISTAGEM 12.23 ConcreteAccountFactory.java**

---

```
public class ConcreteAccountFactory implements AbstractAccountFactory {

    public CheckingAccount createCheckingAccount( double initDeposit, int trans,
double fee ) {
        return new CheckingAccount( initDeposit, trans, fee );
    }

    public OverdraftAccount createOverdraftAccount( double initDeposit, double
rate ) {
        return new OverdraftAccount( initDeposit, rate );
    }

    public RewardsAccount createRewardsAccount( double initDeposit, double
interest, double min ) {
        return new RewardsAccount( initDeposit, interest, min );
    }

    public SavingsAccount createSavingsAccount( double initBalance, double
interestRate ) {
        return new SavingsAccount( initBalance, interestRate );
    }

    public TimedMaturityAccount createTimedMaturityAccount( double initBalance,
double interestRate, double feeRate ){
        return new TimedMaturityAccount( initBalance, interestRate, feeRate );
    }
}
```

---

# SEMANA 2

DIA **13**

## OO e programação da interface com o usuário

A interface com o usuário (UI) fornece a interface entre o usuário e seu sistema. Quase todo sistema moderno terá alguma forma de UI, seja gráfica, dirigida pela linha de comando ou mesmo baseada em telefone ou fala. (Alguns sistemas podem combinar todos os quatro tipos!). Em qualquer caso, você precisa tomar um cuidado especial no projeto e implementação de suas interfaces com o usuário. Felizmente, a POO pode trazer as mesmas vantagens para sua UI que apresenta para outros aspectos do sistema.

Hoje você aprenderá:

- Como a POO e a construção da UI se relacionam
- Sobre a importância de uma UI desacoplada
- Quais padrões o ajudam a desacoplar a UI

## POO e a interface com o usuário

Fundamentalmente, o processo de projetar e programar interfaces com o usuário não é diferente do processo de projetar e programar qualquer outro aspecto de seu sistema. Talvez você precise aprender algumas novas APIs para que possa construir suas UIs, mas no final, você precisa aplicar na UI os mesmos princípios orientados a objetos que aplicaria nas outras partes do seu sistema.



Você aprenderá como encarar o desenvolvimento de UI do ponto de vista de um desenvolvedor. Como desenvolvedor, você projetará e implementará as classes que constituem e suportam a interface com o usuário.

A lição de hoje não abordará o assunto geral do projeto de UI. O projeto de UI abrange todos os aspectos de como os recursos de um programa se tornam disponíveis para um usuário. O assunto geral do projeto de UI está completamente removido da programação e está mais enraizado nas artes gráficas e na psicologia. O ACM Special Interest Group on Computer-Human Interaction (SIGCHI) é um recurso excelente de informações sobre projeto e usabilidade da UI.

A questão merece ênfase: ao projetar e programar suas interfaces com o usuário, você *deve* aplicar em suas UIs os mesmos princípios de OO que aplicaria no restante do seu sistema! Freqüentemente, as interfaces com o usuário são simplesmente reunidas e jogadas no sistema como uma cogitação posterior.

Em vez disso, o código de sua UI precisa ser tão orientado a objetos quanto o código do restante do sistema. A implementação da UI precisa usar encapsulamento, herança e polimorfismo corretamente.

Você também precisa considerar a UI enquanto realiza AOO e POO (Projeto Orientado a Objetos). Sem uma análise e projeto corretos, você pode perder certos requisitos e verificar que escreveu uma UI que não é suficientemente flexível para fornecer o nível desejado de funcionalidade ou uma UI que não pode se adaptar às alterações futuras.

## A importância das UIs desacopladas

Você verá que o mesmo sistema freqüentemente exige diversas interfaces diferentes, muitas vezes não relacionadas. Por exemplo, um sistema de abastecimento pode permitir que as pessoas façam pedidos pela Web, pelo telefone, através de PDA ou de um aplicativo local personalizado. Cada uma dessas interfaces se ligará ao mesmo sistema; entretanto, cada estratégia se ligará ao sistema e apresentará as informações de sua própria maneira.

Você também verá que os requisitos da interface com o usuário podem se tornar um alvo móvel. Os sistemas amadurecem com o tempo, quando novos recursos são adicionados e quando os usuários expõem áreas de debilidade. Em resposta, você precisará atualizar continuamente a interface com o usuário para poder expor cada novo recurso e corrigir todos os defeitos. Essa realidade pede uma interface com o usuário cujo projeto seja flexível e possa aceitar alterações prontamente.

A melhor maneira de obter flexibilidade é projetando um sistema que seja completamente desacoplado de sua UI. Um projeto desacoplado permite que você adicione qualquer UI no sistema e faça alterações nas UIs existentes, sem ter de fazer alterações correspondentes no sistema em si. Um projeto desacoplado também permite testar os recursos do sistema, mesmo que você não te-

nha terminado de desenvolver a UI. Além disso, um projeto desacoplado permite que você aponte precisamente erros que são da UI ou que são do sistema.

Felizmente, a POO é a solução perfeita para esses problemas. Isolando as responsabilidades corretamente, você pode diminuir o impacto das alterações em partes não relacionadas do sistema. Isolando funcionalidade, você deve conseguir adicionar qualquer interface em seu sistema, a qualquer momento, sem fazer alterações no sistema subjacente. O segredo é não incorporar o código da UI dentro do próprio sistema. Os dois *devem* ser separados.

Vamos ver um exemplo que desacopla incorretamente a UI. A Listagem 13.1 mostra como *não* se deve escrever uma UI.

### LISTAGEM 13.1 VisualBankAccount.java

```
import javax.swing.JPanel;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import javax.swing.JTextField;
import javax.swing.JButton;

public class VisualBankAccount extends JPanel implements ActionListener {

    // dados privados
    private double balance;

    // elementos da UI
    private JLabel    balanceLabel = new JLabel();
    private JTextField amountField = new JTextField( 10 );
    private JButton   depositButton = new JButton( "Deposit" );
    private JButton   withdrawButton = new JButton( "Withdraw" );

    public VisualBankAccount( double initDeposit ){
        setBalance( initDeposit );
        buildUI();
    }

    // manipula os eventos dos botões
    public void actionPerformed( ActionEvent e ) {
        if( e.getSource() == depositButton ) {
            double amount = Double.parseDouble( amountField.getText() );
            depositFunds( amount );
        }else if( e.getSource() == withdrawButton ) {
            double amount = Double.parseDouble( amountField.getText() );
            if( amount > getBalance() ) {
                amount = getBalance();
            }
            withdrawFunds( amount );
        }
    }
}
```

**LISTAGEM 13.1** VisualBankAccount.java (*continuação*)

```
        }
        withdrawFunds( amount );
    }
}

private void buildUI() {

    setLayout( new BorderLayout() );

    // constrói a tela
    JPanel buttons = new JPanel( new BorderLayout() );
    JPanel balance = new JPanel( new BorderLayout() );
    buttons.add( depositButton, BorderLayout.WEST );
    buttons.add( withdrawButton, BorderLayout.EAST );
    balance.add( balanceLabel, BorderLayout.NORTH );
    balance.add( amountField, BorderLayout.SOUTH );
    add( balance, BorderLayout.NORTH );
    add( buttons, BorderLayout.SOUTH );

    // configura os callbacks para que os botões façam algo
    // o botão de depósito deve chamar depositFunds()
    depositButton.addActionListener( this );
    // o botão de saque deve chamar withdrawFunds
    withdrawButton.addActionListener( this );
}

public void depositFunds( double amount ) {
    setBalance( getBalance() + amount );
}

public double getBalance() {
    return balance;
}

protected void setBalance( double newBalance ) {
    balance = newBalance;
    balanceLabel.setText( "Balance: " + balance );
}

public double withdrawFunds( double amount ) {
    setBalance( getBalance() - amount );
    return amount;
}
}
```

VisualBankAccount usa a biblioteca Swing da linguagem Java para se apresentar. Toda linguagem OO tem bibliotecas para criar e exibir interfaces gráficas com o usuário (GUI). Não se preocupe se você não entender tudo que há neste exemplo. É importante apenas que você entenda o significado geral do exemplo.

Um pouco de experiência pode ajudar. Swing fornece uma classe para cada elemento importante da GUI, como botões (JButton), rótulos (JLabel) e campos de entrada (JTextField). Você coloca esses elementos juntos, dentro de painéis ( JPanel ), para construir sua UI.

Cada elemento da GUI tem um método `addActionListener()`. Esse método permite que você registre um objeto que implementa a interface ActionListener como um callback. Quando o elemento da GUI gerar um evento (normalmente como resultado de um clique de mouse), ele informará cada um de seus receptores de ação. Aqui, VisualBankAccount atua como receptor. Quando recebe um evento de um dos botões, ele executa a ação correta e, em seguida, deposita ou saca dinheiro.

VisualBankAccount fornece toda a funcionalidade da classe BankAccount, apresentada em lições anteriores. VisualBankAccount também sabe como se apresentar, como se vê na Figura 13.1.

**FIGURA 13.1**

VisualBankAccount *dentro* de um frame.



Quando você digitar um valor e clicar no botão Deposit ou Withdraw, a conta bancária extrairá o valor do campo de entrada e chamará seu método `withdrawFunds()` ou `depositFunds()`, com o valor.

Como VisualBankAccount é um JPanel, você pode incorporá-lo em qualquer GUI Java. Infelizmente, a UI não está desacoplada da classe de conta bancária. Tal acoplamento forte torna impossível usar a conta bancária em outras formas de interfaces com o usuário ou para fornecer uma UI diferente, sem ter de alterar a própria classe VisualBankAccount. Na verdade, você precisará criar uma versão separada da classe para cada tipo de UI dentro da qual queira que ela participe.

## Como desacoplar a UI usando o padrão Model View Controller

O padrão de projeto MVC (Model View Controller) fornece uma estratégia para o projeto de interfaces com o usuário que desacoplam completamente o sistema subjacente da interface com o usuário.

**NOTA**

MVC é apenas uma estratégia para o projeto de interfaces com o usuário orientadas a objetos. Existem outras estratégias válidas para o projeto de interface com o usuário; entretanto, a MVC é uma estratégia testada que é popular no setor do software. Se você acabar realizando o trabalho da interface com o usuário, especialmente em relação à Web e ao J2EE da Sun, encontrará o MVC.

O Document/View Model, popularizado pelas Microsoft Foundation Classes e o padrão de projeto PAC (*Presentation Abstraction Control*), fornecem alternativas à MVC. Veja o livro *Pattern-Oriented Software Architecture A System of Patterns*, de Frank Buschmann et al, para uma apresentação completa dessas alternativas.

O padrão MVC desacopla a UI do sistema, dividindo o projeto da UI em três partes separadas:

- O modelo, que representa o sistema
- O modo de visualização, que exibe o modelo
- O controlador, que processa as entradas do usuário

Cada parte da triade MVC tem seu conjunto próprio de responsabilidades exclusivas.

## O modelo

O modelo é responsável por fornecer:

- Acesso à funcionalidade básica do sistema
- Acesso às informações de estado do sistema
- Um sistema de notificação de mudança de estado

O modelo é a camada da triade MVC que gerencia o comportamento básico e o estado do sistema. O modelo responde às consultas sobre seu estado a partir do modo de visualização e do controlador e aos pedidos de mudança de estado do controlador.

**NOTA**

Um sistema pode ter muitos modelos diferentes. Por exemplo, um sistema de banco pode ser constituído de um modelo de conta e um modelo de caixa. Vários modelos pequenos repartem melhor a responsabilidade do que um único modelo grande.

Não deixe o termo *modelo* confundi-lo. Um modelo é apenas um objeto que representa o sistema.

O controlador é a camada da triade MVC que interpreta a entrada do usuário. Em resposta à entrada do usuário, o controlador pode comandar o modelo ou o modo de visualização para que mude ou execute alguma ação.

O modo de visualização é a camada da triade MVC que exibe a representação gráfica ou textual do modelo. O modo de visualização recupera todas as informações de estado a respeito do modelo, a partir do modelo.

Em qualquer caso, o modelo não sabe absolutamente que um modo de visualização ou controlador está fazendo uma chamada de método. O modelo só sabe que um objeto está chamando um de seus métodos. A única conexão que um modelo mantém com a UI é através do sistema de notificação de mudança de estado.

Se um modo de visualização ou controlador estiver interessado na notificação de mudança de estado, ele se registrará no modelo. Quando o modelo mudar de estado, percorrerá sua lista de objetos registrados (freqüentemente chamados de receptores ou observadores) e informará cada objeto da mudança de estado. Para construir esse sistema de notificação, os modelos normalmente empregarão o padrão Observer.

### O padrão Observer

O padrão Observer fornece um projeto para um mecanismo de publicação/assinatura entre objetos. O padrão Observer permite que um objeto (o observador) registre seu interesse em outro objeto (o observável). Quando o observável quiser notificar os seus observadores de uma alteração, ele chamará um método `update()` em cada observador.

A Listagem 13.2 define a interface `Observer`. Todos os observadores que quiserem se registrar com o objeto observável devem implementar a interface `Observer`.

#### **LISTAGEM 13.2** *Observer.java*

```
public interface Observer {  
    public void update();  
}
```

Um observável fornecerá um método, através do qual os observadores podem registrar e anular o registro de seu interesse em atualizações. A Listagem 13.3 apresenta uma classe que implementa o padrão Observer.

## Implementando o modelo

Aplicar o padrão MVC no `VisualBankAccount` pode torná-lo muito mais flexível. Vamos começar retirando a funcionalidade básica do ‘sistema’ do código de exibição, para podermos criar o modelo.

A Listagem 13.3 apresenta a funcionalidade básica da conta bancária — o modelo.

#### **LISTAGEM 13.3** *BankAccountModel.java*

```
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class BankAccountModel {  
  
    // dados privados  
    private double balance;
```

**LISTAGEM 13.3 BankAccountModel.java**

```
private ArrayList listeners = new ArrayList();

public BankAccountModel( double initDeposit ) {
    setBalance( initDeposit );
}

public void depositFunds( double amount ) {
    setBalance( getBalance() + amount );
}

public double getBalance() {
    return balance;
}

protected void setBalance( double newBalance ) {
    balance = newBalance;
    updateObservers();
}

public double withdrawFunds( double amount ) {
    if( amount > getBalance() ) {
        amount = getBalance();
    }
    setBalance( getBalance() - amount );
    return amount;
}

public void register( Observer o ) {
    receptores.add( o );
    o.update();
}

public void deregister( Observer o ) {
    receptores.remove( o );
}

private void updateObservers() {
    Iterator i = receptores.iterator();
    while( i.hasNext() ) {
        Observer o = ( Observer ) i.next();
        o.update();
    }
}
```

---

A BankAccountModel é semelhante à classe BankAccount original, apresentada em lições anteriores; entretanto, o modelo também usa o padrão Observer para adicionar suporte ao registro e atualização de objetos que estejam interessados na notificação de mudança de estado.

Você também notará que essa classe contém toda a lógica do sistema. Agora, `withdrawFunds()` verifica o valor do saque para garantir que ele não seja maior do que o saldo. É fundamental manter tais regras de domínio dentro do modelo. Se essas regras fossem obedecidas no modo de visualização ou no controlador, cada modo de visualização e controlador desse modelo precisaria manter a regra. Conforme você já viu, ter cada modo de visualização ou controlador impondo essa regra é uma mistura de responsabilidades e é algo propenso a erros. A mistura de responsabilidades também torna difícil mudar a regra, pois você precisa mudá-la em cada lugar.

Os relacionamentos com capacidade de substituição também tornam a colocação de regras no modo de visualização uma prática perigosa. Devido aos relacionamentos com capacidade de substituição, um modo de visualização funcionará para qualquer subclasse; entretanto, se você colocar as regras de saque no modo de visualização, este não funcionará mais para um objeto OverdraftAccount. Isso porque os objetos OverDraftAccount permitem que você saque valores maiores do que o saldo corrente.

## O modo de visualização

O modo de visualização é responsável por:

- Apresentar o modelo para o usuário
- Registrar no modelo notificação de mudança de estado
- Recuperar informações de estado do modelo

O modo de visualização é a camada da tríade MVC que exibe informações para o usuário. O modo de visualização obtém informações de exibição do modelo, usando a interface pública deste, e também se registrará no modelo para que ele possa ser informado da mudança de estado e se atualize de acordo.

**NOTA**

Um único modelo pode ter muitos modos de visualização diferentes.

**13**

## Implementando o modo de visualização

Agora que existe um modelo, é hora de implementar o modo de visualização da conta bancária. A Listagem 13.4 apresenta o modo de visualização de BankAccountModel.

### LISTAGEM 13.4 BankAccountView.java

```
import javax.swing.JPanel;
import javax.swing.JLabel;
```

**LISTAGEM 13.4** BankAccountView.java (*continuação*)

```
import java.awt.BorderLayout;
import javax.swing.JTextField;
import javax.swing.JButton;

public class BankAccountView extends JPanel implements Observer {

    public final static String DEPOSIT = "Deposit";
    public final static String WITHDRAW = "Withdraw";

    private BankAccountModel model;
    private BankAccountController controller;

    // Elementos da GUI, aloca tudo previamente para evitar valores nulos
    private JButton depositButton = new JButton( DEPOSIT );
    private JButton withdrawButton = new JButton( WITHDRAW );
    private JTextField amountField = new JTextField();
    private JLabel balanceLabel = new JLabel();

    public BankAccountView( BankAccountModel model ) {
        this.model = model;
        this.model.register( this );
        attachController( makeController() );
        buildUI();
    }

    // chamado pelo modelo quando este muda
    public void update() {
        balanceLabel.setText( "Balance: " + model.getBalance() );
    }

    // dá acesso ao valor introduzido no campo
    public double getAmount() {
        // supõe que o usuário introduziu um número válido
        return Double.parseDouble( amountField.getText() );
    }

    // insere o controlador dado no modo de visualização, permite que um objeto
    // externo configure o controlador
    public void attachController( BankAccountController controller ) {
        // cada modo de visualização só pode ter um controlador; portanto, remove
        // o antigo primeiro
        if( this.controller != null ) { //remove o controlador antigo
            depositButton.removeActionListener( controller );
            withdrawButton.removeActionListener( controller );
        }
    }
}
```

**LISTAGEM 13.4** BankAccountView.java (*continuação*)

```
this.controller = controller;
depositButton.addActionListener( controller );
withdrawButton.addActionListener( controller );
}

protected BankAccountController makeController() {
    return new BankAccountController( this, model );
}

private void buildUI() {
    setLayout( new BorderLayout() );

    // associa cada botão a uma string encomendada
    // o controlador usará essa string para interpretar eventos
    depositButton.setActionCommand( DEPOSIT );
    withdrawButton.setActionCommand( WITHDRAW );

    // constrói a tela
    JPanel buttons = new JPanel( new BorderLayout() );
    JPanel balance = new JPanel( new BorderLayout() );
    buttons.add( depositButton, BorderLayout.WEST );
    buttons.add( withdrawButton, BorderLayout.EAST );
    balance.add( balanceLabel, BorderLayout.NORTH );
    balance.add( amountField, BorderLayout.SOUTH );
    add( balance, BorderLayout.NORTH );
    add( buttons, BorderLayout.SOUTH );
}
}
```

O construtor de BankAccountView aceita uma referência para um BankAccountModel. Na criação, BankAccountView se registra no modelo, cria e se anexa ao seu controlador e constrói sua UI. O modo de visualização usa o modelo para recuperar todas as informações de que necessita para a exibição. Quando o saldo mudar, o modelo chamará o método update() do modo de visualização. Quando esse método for chamado, o modo de visualização atualizará sua tela de saldo.

Normalmente, um modo de visualização criará seu próprio controlador, como BankAccountView faz dentro do método factory makeController(). As subclasses podem sobrepor esse método factory para criar um controlador diferente. Dentro do método attachController(), o modo de visualização registra o controlador nos botões de depósito e saque, para que o controlador possa receber eventos do usuário.

Você notará, entretanto, que o modo de visualização primeiro remove qualquer controlador previamente existente. Um modo de visualização normalmente terá apenas um controlador.

Você também notará que `attachController()` é um método público. Usando esse método, você pode trocar de controlador sem ter que fazer uma subclasse do modo de visualização. Esse método permite que você crie diferentes controladores e passe-os para o modo de visualização. O controlador que você verá na próxima seção interpretará os eventos do usuário exatamente como `VisualBankAccount` os interpretava (apenas com uma ligeira modificação). Nada o impede de escrever controladores que bloqueiem o usuário ou limitem o que ele pode fazer.

Ao contrário de um modo de visualização, que só pode ter um controlador por vez, um modelo pode ter muitos modos de visualização diferentes. A Listagem 13.5 apresenta um segundo modo de visualização para a conta bancária.

---

**LISTAGEM 13.5** BankAccountCLV.java

```
public class BankAccountCLV implements Observer {  
  
    private BankAccountModel model;  
  
    public BankAccountCLV( BankAccountModel model ) {  
        this.model = model;  
        this.model.register( this );  
    }  
  
    public void update() {  
        System.out.println( "Current Balance: $" + model.getBalance() );  
    }  
}
```

---

`BankAccountCLV` simplesmente imprime o saldo na linha de comando. Embora esse comportamento seja simples, `BankAccountCLV` é um modo de visualização alternativo para `BankAccountModel`. Você notará que esse modo de visualização não exige um controlador, pois ele não aceita eventos do usuário. Nem sempre você precisa fornecer um controlador.

 **DICA**

Um modo de visualização pode nem sempre aparecer na tela.

Tome como exemplo um processador de textos. O modelo do processador de textos controlará o texto introduzido, a formatação, notas de pé de página etc. Um modo de visualização exibirá o texto no editor principal; entretanto, outro modo de visualização poderá converter os dados do modelo para o formato PDF, HTML ou Postscript e, em seguida, gravá-lo em um arquivo. O modo de visualização que grava em um arquivo não aparece na tela; em vez disso, o modo de visualização exibe em um arquivo. Outros programas podem então abrir, ler e exibir os dados do arquivo.

## O controlador

O controlador é responsável por:

- Interceptar os eventos do usuário do modo de visualização
- Interpretar o evento e chamar os métodos corretos do modelo ou modo de visualização
- Registrar-se no modelo para notificação de mudança de estado, se estiver interessado

O controlador atua como a cola entre o modo de visualização e o modelo. O controlador intercepta eventos do modo de visualização e depois os transforma em pedidos do modelo ou do modo de visualização.



### NOTA

Um modo de visualização tem apenas um controlador e um controlado tem apenas um modo de visualização. Alguns modos de visualização permitem que você configure seu controlador diretamente.

Cada modo de visualização tem um controlador e toda a interação com o usuário passa por esse controlador. Se o controlador for dependente das informações de estado, ele também será registrado no modelo para notificação de mudança de estado.

## Implementando o controlador

Com um modelo e um modo de visualização já criados, resta apenas construir o controlador de BankAccountView. A Listagem 13.6 apresenta o controlador do modo de visualização.

### LISTAGEM 13.6 BankAccountController.java

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class BankAccountController implements ActionListener {

    private BankAccountView view;
    private BankAccountModel model;

    public BankAccountController( BankAccountView view, BankAccountModel model )
    {
        this.view = view;
        this.model = model;
    }

    public void actionPerformed( ActionEvent e ) {

        String command = e.getActionCommand();
        double amount = view.getAmount();
        if( command.equals( view.WITHDRAW ) ) {
```

**LISTAGEM 13.6** BankAccountController.java (*continuação*)

```
        model.withdrawFunds( amount );
    }else if( command.equals( view.DEPPOSIT ) ) {
        model.depositFunds( amount );
    }
}
```

---

Na construção, o controlador aceita uma referência para o modo de visualização e para o modelo. O controlador usará o modo de visualização para obter os valores introduzidos no campo de entrada. O controlador usará o modelo para realmente sacar e depositar dinheiro na conta.

BankAccountController em si é muito simples. O controlador implementa a interface ActionListener de modo que possa receber eventos do modo de visualização. O modo de visualização cuida do registro do controlador para eventos, de modo que o controlador não precisa fazer nada a não ser interpretar eventos, quando os receber.

Quando o controlador recebe um evento, ele verifica o comando do evento para determinar se o evento é um saque ou um depósito. Em qualquer caso, ele faz a chamada correspondente no modelo. Ao contrário do VisualBankAccount original, o controlador só precisa chamar depositFunds() ou withdrawFunds(). Ele não precisa mais se certificar de que o valor do saque não seja maior que o saldo, pois agora o modelo cuida desse detalhe do domínio.

## Reunindo o modo de visualização e o controlador

A Listagem 13.7 apresenta um pequeno método main que reúne o modelo e dois modos de visualização. O método main não precisa fazer nada no controlador, pois o modo de visualização cuida desse detalhe.

**LISTAGEM 13.7** Reunindo o modelo, modos de visualização e o controlador

```
import java.awt.event.WindowListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;

public class MVCDriver {

    public static void main( String [] args ) {

        BankAccountModel model = new BankAccountModel( 10000.00 );
        BankAccountView view   = new BankAccountView( model );
        BankAccountCLV clv    = new BankAccountCLV( model );

        JFrame frame = new JFrame();
```

**LISTAGEM 13.7** Reunindo o modelo, modos de visualização e o controlador (*cont.*)

```

WindowAdapter wa = new WindowAdapter() {
    public void windowClosing( WindowEvent e ) {
        System.exit( 0 );
    }
};

frame.addWindowListener( wa );

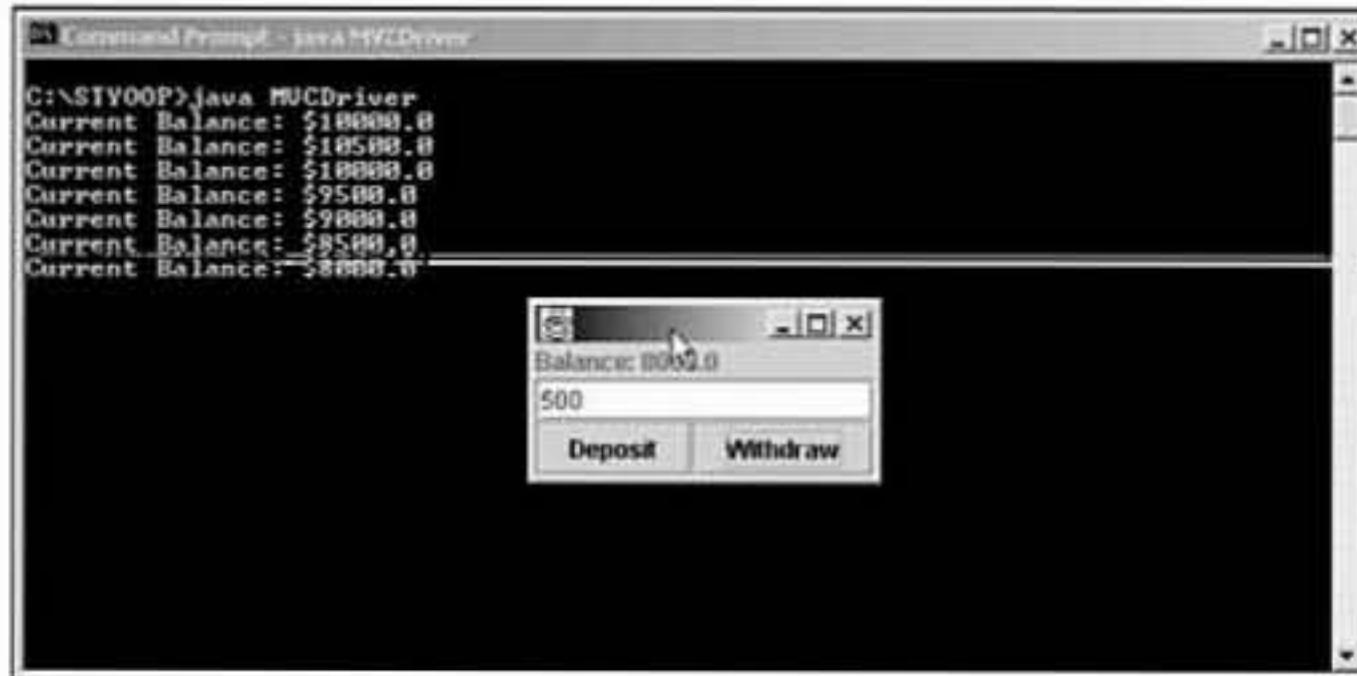
frame.getContentPane().add( view );
frame.pack();
frame.show();
}
}

```

O primeiro método `main` cria uma instância do modelo. Quando o método `main` tem o modelo, pode então criar vários modos de visualização. No caso de `BankAccountView`, o método `main` também precisa incorporar o modo de visualização em um quadro, para que o modo de visualização possa ser exibido. A Figura 13.2 ilustra a saída resultante.

**FIGURA 13.2**

*Um modelo de conta bancária com vários modos de visualização.*



Se você executar `MVCDriver`, verá dois modos de visualização separados no mesmo modelo. Usando o padrão MVC, você pode criar quantos modos de visualização de seus modelos subjacentes precisar.

13

## Problemas com o MVC

Assim como acontece com qualquer projeto, o MVC tem suas deficiências e também seus pontos críticos. Os problemas incluem:

- Uma ênfase nos dados
- Um forte acoplamento entre o modo de visualização/controlador e o modelo
- Uma oportunidade de ineficiência

A gravidade dessas deficiências depende do problema que se está resolvendo e seus requisitos.

## Uma ênfase nos dados

Em uma escala OO de pureza, o padrão MVC não se classifica próximo ao topo, devido a sua ênfase nos dados. Em vez de pedir a um objeto para que faça algo com seus dados, o modo de visualização pede seus dados ao modelo e depois os exibe.

Você pode diminuir a gravidade desse problema, exibindo apenas os dados que retira do modelo. Não realize processamento adicional nos dados. Se você se achar realizando processamento extra nos dados, após recuperá-los ou antes de chamar um método no modelo, são boas as chances de que o modelo deve fazer esse trabalho para você. Existe uma linha tênue entre fazer muito e fazer o que é necessário nos dados. Com o passar do tempo, você aprenderá a diferenciar entre fazer muito com os dados e fazer apenas o que é necessário.



**DICA**  
Se você verificar que repete o mesmo código em cada modo de visualização, considere a colocação dessa lógica no modelo.

Evitar o padrão MVC unicamente por motivos de pureza pode insultar algumas realidades da programação. Tome como exemplo um site Web. Algumas empresas impõem uma separação clara entre apresentação (o modo de visualização) e a lógica corporativa (o modelo). A imposição de tal separação tem uma base corporativa válida: os programadores podem programar e o pessoal ligado ao conteúdo pode escrever conteúdo. Retirar o conteúdo da camada de programação significa que quem não for programador pode criar conteúdo. Impor o conteúdo na camada de programação significa que a pessoa que está escrevendo conteúdo deve ser um programador ou que um programador precisa pegar o conteúdo e incorporá-lo dentro do código. É muito mais difícil atualizar um site, se você incorporar conteúdo no código.

A realidade também diz que os requisitos não são definitivos, muito menos conhecidos. Novamente, o site Web se constitui em um exemplo excelente. Um site Web precisa gerar código HTML para exibir em um navegador da Web. E quanto aos PDAs, telefones celulares e outros dispositivos de exibição? Nenhum deles usa HTML básica. E quanto daqui a seis meses? São boas as chances de que vão existir outras formas de exibição. Para satisfazer requisitos desconhecidos, você precisa de um projeto que seja flexível. Se você tiver um sistema muito estático, com requisitos bem definidos, poderá usar uma alternativa, como um PAC. Se você não tiver sorte suficiente para ter tais requisitos claros, precisará considerar o MVC.

## Acoplamento forte

O modo de visualização e o controlador são fortemente acoplados à interface pública do modelo. As alterações na interface do modelo exigirão alterações no modo de visualização e no controlador. Quando usa o padrão MVC, você supõe implicitamente que o modelo é estável, e é provável

que o modo de visualização mude. Se esse não for o caso, você precisará escolher um projeto diferente ou estar preparado para fazer alterações no modo de visualização e no controlador.

O modo de visualização e o controlador também são intimamente relacionados entre si. Um controlador é quase sempre usado exclusivamente com um modo de visualização específico. Você pode tentar encontrar reutilização através de um projeto cuidadoso; mas mesmo que não encontre, o padrão MVC ainda fornece uma boa divisão de responsabilidades entre os objetos. A OO não é simplesmente um meio de reutilização.

## Ineficiência

Você deve tomar o cuidado de evitar ineficiências ao projetar e implementar uma UI baseada em MVC. As ineficiências podem aparecer no sistema em qualquer parte da tríade MVC.

O modelo deve evitar a propagação de notificações de mudança de estado desnecessárias para seus observadores. Um modelo pode enfileirar notificações de mudança relacionadas para que uma notificação possa significar muitas mudanças de estado. O modelo de evento AWT (Abstract Window Toolkit) da linguagem Java usa essa estratégia para redesenhar a tela. Em vez de redesenhar após cada evento, AWT enfileira os eventos e realiza uma única operação para redesenhar.

Ao projetar o controlador e o modo de visualização, talvez você queira considerar a colocação dos dados na memória cache, caso a recuperação de dados do modelo seja lenta. Após uma notificação de mudança de estado, recupere apenas o estado que mudou. Você pode aumentar o padrão do observador para que o modelo passe um identificador para o método `update()`. O modo de visualização pode usar esse identificador para decidir se precisa ou não se atualizar.

## Resumo

A interface com o usuário é uma parte importante de qualquer sistema. Para alguns, ela pode ser a única parte do sistema com a qual eles interagem; para esses, a UI é o sistema. Você sempre deve encarar a análise, o projeto e a implementação da UI exatamente como encara qualquer outra parte do sistema. Uma UI nunca deve ser uma cogitação posterior ou algo colocado no sistema no último momento.

Embora existam muitas estratégias para o projeto da UI, o padrão MVC fornece um projeto que oferece flexibilidade, desacoplando a UI do sistema subjacente. Mas, assim como acontece com qualquer outra decisão de projeto, você ainda precisa ponderar os prós e contras do MVC, antes de decidir utilizá-lo. O MVC não o exime das realidades de seu sistema.

13

## Perguntas e respostas

- P Sua classe `BankAccountModel` contém toda a lógica do sistema. O modelo sempre precisa conter a lógica ou ele pode atuar como um gateway para o sistema real?**

**R** Depende. Às vezes, o modelo agirá como um gateway para o sistema; outras vezes, o modelo será incorporado dentro do sistema real. Tudo se resume a uma decisão de projeto. Em qualquer caso, a UI não tem meios de saber se o modelo atua como um gateway ou não.

**P** Na Listagem 13.6, você escreveu:

```
if( command.equals( view.WITHDRAW ) ) {  
    model.withdrawFunds( amount );  
} else if( command.equals( view.DEPOSIT ) ) {  
    model.depositFunds( amount );  
}
```

**Isso não é lógica com estruturas condicionais? Achei que você tinha dito que lógica com estruturas condicionais é considerada ‘má’ OO.**

**R** Sim. Esse é um exemplo de lógica com estruturas condicionais.

Para manter esse exemplo simples, decidimos tornar o controlador um `ActionListener` que manipulasse os dois eventos. Em uma implementação real, você pode evitar a lógica com estruturas condicionais capturando o evento original dentro do próprio modo de visualização e fazendo com que o modo de visualização gere seus próprios eventos personalizados. Por exemplo, o modo de visualização poderia gerar eventos de depósito e saque. O controlador poderia receber cada um desses eventos separadamente. Talvez o controlador implementasse os métodos `depositPerformed()` e `withdrawPerformed()`. O modo de visualização chamaria o método correto no controlador, dependendo do evento; assim, não haveria mais nenhuma estrutura condicional, mas um exemplo muito mais difícil de entender.

**P** Certo. Sua resposta anterior me fez sentir um pouco melhor. Mas se você implementar o controlador conforme explicado anteriormente, o modo de visualização não terá de executar lógica com estrutura condicional para descobrir qual botão chamou o evento?

**R** Não. O modo de visualização pode evitar a lógica com estruturas condicionais fornecendo um receptor separado para cada botão. Quando existe uma correspondência de um para um entre um elemento e seu receptor, você não precisa usar lógica com estruturas condicionais para descobrir onde o evento se origina. (Veja no Exercício 2 a implementação alternativa.)

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Como a análise, o projeto e a implementação da UI são diferentes do restante do sistema?
2. Por que você deve desacoplar a UI do sistema subjacente?
3. Quais são os três componentes da tríade MVC?
4. Quais são as duas alternativas para o padrão MVC?
5. Descreva as responsabilidades do modelo.
6. Descreva as responsabilidades do modo de visualização.
7. Descreva as responsabilidades do controlador.
8. Quantos modelos um sistema pode ter? Quantos modos de visualização um modelo pode ter? Quantos controladores um modo de visualização pode ter?
9. Quais ineficiências você deve evitar ao usar o padrão MVC?
10. Quais suposições o padrão MVC faz?
11. Qual é a história do padrão MVC? (Note que esta pergunta exige que você realize uma rápida pesquisa na Web.)

## Exercícios

1. A Listagem 13.8 apresenta uma classe Employee. Altere a classe Employee de modo que ela possa registrar e retirar o registro de receptores, assim como informá-los de alterações de estado. A Listagem 13.2 apresenta uma interface Observer que você pode usar para este exercício.

### LISTAGEM 13.8 EmployeeModel.java

```
public abstract class Employee {  
  
    private String first_name;  
    private String last_name;  
    private double wage;  
  
    public Employee(String first_name, String last_name, double wage) {  
        this.first_name = first_name;  
        this.last_name = last_name;  
        this.wage = wage;  
    }  
  
    public double getWage() {  
        return wage;  
    }
```

**LISTAGEM 13.8 EmployeeModel.java (continuação)**

---

```
public void setWage( double wage ) {
    this.wage = wage;
}

public String getFirstName() {
    return first_name;
}

public String getLastName(){
    return last_name;
}

public abstract double calculatePay();

public abstract double calculateBonus();

public void printPaycheck() {
    String full_name = last_name + ", " + first_name;
    System.out.println( "Pay: " + full_name + " $" + calculatePay() );
}
}
```

---

2. Usando as listagens 13.9 e 13.10 como ponto de partida, escreva um novo BankAccountController que implemente a nova interface BankActivityListener e manipule os eventos sem lógica com estruturas condicionais.

A Listagem 13.9 apresenta um BankActivityEvent e seu BankActivityListener correspondente.

**LISTAGEM 13.9 BankActivityListener.java e BankActivityEvent.java**

---

```
public interface BankActivityListener {

    public void withdrawPerformed( BankActivityEvent e );

    public void depositPerformed( BankActivityEvent e );
}

public class BankActivityEvent {

    private double amount;

    public BankActivityEvent( double amount ) {
        this.amount = amount;
    }
}
```

**LISTAGEM 13.9** BankActivityListener.java e BankActivityEvent.java (*continuação*)

```
}

public double getAmount() {
    return amount;
}
}
```

A Listagem 13.10 apresenta uma BankAccountView atualizada. Esse BankAccountView intercepta os eventos ActionEvent do botão e encaminha o novo evento BankActivityEvent para o controlador.

**LISTAGEM 13.10** BankAccountView.java

```
import javax.swing.JPanel;
import javax.swing.JLabel;
import java.awt.BorderLayout;
import javax.swing.JTextField;
import javax.swing.JButton;
import java.util.ArrayList;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class BankAccountView extends JPanel implements Observer {

    public final static String DEPOSIT = "Deposit";
    public final static String WITHDRAW = "Withdraw";

    private BankAccountModel model;
    private BankAccountController controller;

    // Elementos da GUI, aloca tudo previamente para evitar valores nulos
    private JButton depositButton = new JButton( DEPOSIT );
    private JButton withdrawButton = new JButton( WITHDRAW );
    private JTextField amountField = new JTextField();
    private JLabel balanceLabel = new JLabel();

    public BankAccountView( BankAccountModel model ) {
        this.model = model;
        this.model.register( this );
        attachController( makeController() );
        buildUI();
    }

    // chamado pelo modelo, quando este muda
```

**LISTAGEM 13.10** BankAccountView.java (*continuação*)

```
public void update() {
    balanceLabel.setText( "Balance: " + model.getBalance() );
}

// codifica o controlador dados no modo de visualização, permite que objeto
externo configure o controlador
public void attachController( BankAccountController controller ) {
    this.controller = controller;
}

protected BankAccountController makeController() {
    return new BankAccountController( this, model );
}

// dá acesso ao valor introduzido no campo
private double getAmount() {
    // pressupõe que o usuário introduziu um número válido
    return Double.parseDouble( amountField.getText() );
}

private void fireDepositEvent() {
    BankActivityEvent e = new BankActivityEvent( getAmount() );
    controller.depositPerformed( e );
}

private void fireWithdrawEvent() {
    BankActivityEvent e = new BankActivityEvent( getAmount() );
    controller.withdrawPerformed( e );
}

private void buildUI() {

    setLayout( new BorderLayout() );

    // associa cada botão a uma string de encomenda
    depositButton.setActionCommand( DEPOSIT );
    withdrawButton.setActionCommand( WITHDRAW );
    // constrói a tela
    JPanel buttons = new JPanel( new BorderLayout() );
    JPanel balance = new JPanel( new BorderLayout() );
    buttons.add( depositButton, BorderLayout.WEST );
    buttons.add( withdrawButton, BorderLayout.EAST );
    balance.add( balanceLabel, BorderLayout.NORTH );
    balance.add( amountField, BorderLayout.SOUTH );
    add( balance, BorderLayout.NORTH );
```

**LISTAGEM 13.10** BankAccountView.java (*continuação*)

```
add( buttons, BorderLayout.SOUTH );

depositButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent e ) {
            fireDepositEvent();
        }
    }
);

withdrawButton.addActionListener(
    new ActionListener() {
        public void actionPerformed( ActionEvent e ) {
            fireWithdrawEvent();
        }
    }
);
}
```

**PÁGINA EM BRANCO**

# SEMANA 2

DIA **14**

## Construindo software confiável através de testes

Quando você usa programação orientada a objetos, se esforça para escrever software natural, confiável, reutilizável, manutenível, extensível e oportuno. Para atingir esses objetivos, você deve entender que a ‘boa’ OO não acontece por acidente. Você deve atacar seus problemas através de uma análise e um projeto cuidadosos, e ao mesmo tempo nunca perder de vista os princípios básicos da POO. Somente então a OO pode começar a cumprir suas promessas. Mesmo com uma análise e um projeto cuidadosos, entretanto, a POO não é uma fórmula mágica. Ela não o protegerá de seus próprios erros ou dos erros dos outros. E erros acontecerão! Para produzir software confiável, você precisa testá-lo.

Hoje você aprenderá:

- Onde os testes entram no processo iterativo
- Sobre os diferentes tipos de testes
- Como testar suas classes
- Como testar software incompleto
- O que você pode fazer para escrever código mais confiável
- Como tornar seus testes mais eficazes

## Testando software OO

A OO não evitara que erros aconteçam em seu software. Mesmo os melhores programadores cometem erros. Os erros são normalmente considerados como um defeito de software que surge de um erro de digitação, de um erro na lógica ou apenas por um engano bobo cometido durante a codificação. Embora a implementação de um objeto seja uma fonte de erros comum, eles aparecem em outras formas.

Erros também podem resultar quando um objeto usa outro incorretamente. Os erros podem até ser provenientes de falhas básicas na análise ou no próprio projeto. Por sua própria natureza, um sistema OO é repleto de objetos interagindo. Essas interações podem ser a fonte de todos os tipos de erros.

Felizmente, você pode proteger seu software de erros, através de testes de software, onde é possível validar a análise, o projeto e a implementação de seu software.



**ALERTA**  
Assim como a OO, o teste não é uma solução mágica; é extremamente difícil testar seu software completamente. O número total de caminhos possíveis através de um programa não trivial torna difícil e demorado obter cobertura total do código. Assim, mesmo um código testado pode abrigar erros ocultos.

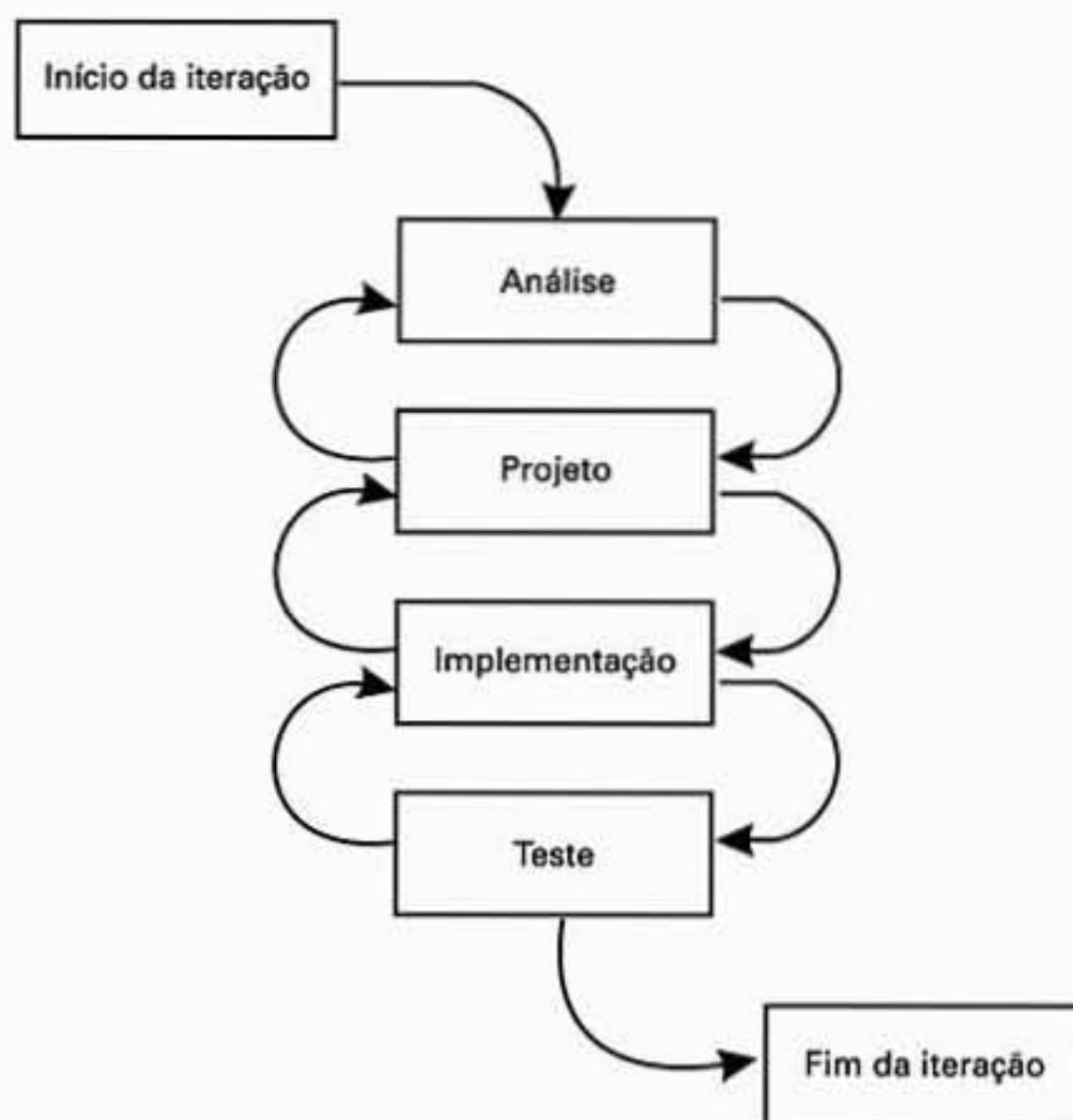
O melhor que você pode fazer é realizar uma quantidade de testes que garantam a qualidade de seu código, enquanto também permitam cumprir seus prazos finais e permanecer dentro do orçamento. A ‘quantidade’ real de testes que você realizará dependerá da abrangência do projeto e de seus próprios níveis de bem-estar.

## Testes e o processo de desenvolvimento de software iterativo

A Figura 14.1 ilustra a iteração apresentada pela primeira vez no Capítulo 9, “Introdução à análise orientada a objetos”. O teste é a última etapa de uma iteração.

Antes de você sair de uma iteração, o teste é uma etapa importante. O estágio de testes verifica se todas as alterações feitas por você durante essa iteração não danificaram qualquer funcionalidade existente. O estágio de testes também verifica se toda nova funcionalidade adicionada agora funciona corretamente. Por esses motivos, os testes realizados antes de se sair de uma iteração são freqüentemente referidos como testes *funcionais* ou de *aceitação*.

**FIGURA 14.1**  
*Uma iteração.*



**NOTA**

O teste no final de uma iteração é um marco importante. Para sair da iteração, seu sistema deve passar pelos testes; entretanto, também devem ocorrer testes durante outros estágios de uma iteração. As lições de hoje mostrarão a você como usar testes eficientemente, durante a implementação e estágios de testes da iteração.

Torne os testes um objetivo e algo que você faz por todo o desenvolvimento. Se você não pensar nos testes até o final do desenvolvimento, poderá descobrir que não é possível testar seu software. Em vez disso, você precisa desenvolver pensando nos testes. Você deve tornar os testes uma parte integrante do processo de desenvolvimento.

Se erros forem encontrados, você deverá voltar e corrigi-los. Normalmente, você voltará para a implementação e tentará corrigir o problema no código. Às vezes, isso é tudo que você precisará fazer: bastará corrigir a implementação, testar tudo novamente e prosseguir. Entretanto, os erros podem ser provenientes de uma falha de projeto ou mesmo de um requisito ignorado ou mal-entendido. Talvez você precise voltar ao projeto ou à análise, antes de poder corrigir um erro na implementação.

**ALERTA**

Após corrigir um erro, não é suficiente apenas testar o erro corrigido. Em vez disso, você precisa realizar todos os testes. Ao corrigir um erro, você pode introduzir facilmente um ou mais erros novos!

Um mal-entendido na análise significa que o sistema não funcionará conforme o cliente espera. Um sistema deve funcionar conforme o esperado e o cliente que conduz o sistema deve concordar com o que é esperado do comportamento. Não apenas você precisa testar o código quanto a

falhas de implementação, como também precisa testar o código para ver se ele funciona conforme o esperado.

Para testar um sistema, você precisa escrever e executar casos de teste. Cada caso de teste testará um aspecto específico do sistema.

**Novo Termo** Um *caso de teste* é o bloco de construção básico do processo de teste. O processo de teste executa vários *casos de teste* para poder validar completamente um sistema.

Cada *caso de teste* consiste em um conjunto de entradas e saídas esperadas. O teste executará um caminho específico através do sistema (caixa branca) ou testará algum comportamento definido (caixa preta).

Um caso de teste exercita uma funcionalidade específica para ver se o sistema se comporta como deveria. Se o sistema se comportar conforme o esperado, o caso de teste passa. Se o sistema não se comportar conforme o esperado, o caso de teste falha. Um caso de teste falho indica que existe um erro no sistema. Você sempre quer que todos os seus casos de teste passem 100% das vezes. Não tente ignorar um caso de teste falho, se cem outros casos passarem. Todo teste deve passar ou você não poderá continuar seu trabalho!

Existem duas maneiras de basear seus casos de teste: teste de caixa preta e de caixa branca. Uma estratégia de teste eficaz terá uma mistura de casos de teste baseados em caixa preta e em caixa branca.

**Novo Termo** O *teste de caixa preta* testa se o sistema funciona conforme o esperado. Dada uma entrada específica, o *teste de caixa preta* testa se a saída ou comportamento correto, visível externamente, resulta conforme definido pela especificação da classe ou do sistema.

**Novo Termo** No *teste de caixa branca*, os testes são baseados unicamente na implementação de um método. Os testes de caixa branca tentam atingir 100% de cobertura do código.

Ao testar classes individuais, o teste de caixa preta é baseado nos requisitos funcionais da classe. Ao testar o sistema inteiro, o teste de caixa preta é baseado nos casos de uso. Em qualquer caso, o teste de caixa preta verifica se um objeto ou sistema se comporta conforme o esperado. Por exemplo, se um método deve somar dois números, um teste de caixa preta enviará dois números para o método e, em seguida, verificará se a saída é ou não a soma correta dos dois números. Se um sistema deve permitir que você adicione e remova itens de um carrinho de compras, um teste de caixa preta tentará adicionar e remover itens do carrinho.

O teste de caixa branca, por outro lado, é baseado na implementação de um método. Seu objetivo é garantir que cada desvio do código seja executado. O teste de caixa preta é avaliado para cobrir apenas de um terço à metade do código real. Com o teste de caixa branca, você projeta seus testes de modo a exercitar cada desvio do código e na esperança de eliminar todos os erros latentes.

**DICA**

Os testes de caixa branca, exceto quanto aos programas mais simples, raramente podem alcançar uma cobertura razoável da combinação de caminhos através do programa. Existem dois passos que você pode dar para melhorar a eficiência de seus testes:

- Escreva seus programas de modo que eles tenham um número mínimo de caminhos.
- Identifique caminhos críticos e certifique-se de testá-los.

Por exemplo, se um método divide dois números e existe um desvio de erro que é executado quando você tenta dividir por 0, será preciso garantir que exista um caso de teste que exerce essa condição de erro. A não ser que seja especificado na documentação da interface, você saberia a respeito desse desvio examinando o próprio código. Assim, os testes de caixa branca devem ser baseados no próprio código.

Em qualquer caso, os testes de caixa preta e de caixa branca governam o modo como você cria seus casos de teste. Cada um desempenha um papel importante na forma de teste que você pode executar.

## Formas de teste

No todo, existem quatro formas importantes de teste. Esses testes variam de testes de nível mais baixo, que examinam os objetos individuais, até os testes de nível mais alto, que examinam o sistema inteiro. A execução de cada um ajudará a garantir a qualidade global de seu software.

### Teste de unidade

O teste de unidade é a unidade de nível mais baixo dos testes. Um teste de unidade examina apenas um recurso por vez.

**Novo Termo**

Um *teste de unidade* é o dispositivo de teste de nível mais baixo. Um teste de unidade envia uma mensagem para um objeto e depois verifica se ele recebe o resultado esperado do objeto. Um teste de unidade verifica apenas um recurso por vez.

Em termos de OO, um teste de unidade examina uma única classe de objeto. Um teste de unidade verifica um objeto enviando uma mensagem e verifica se ele retorna o resultado esperado. Você pode basear os testes de unidade no teste de caixa preta e no de caixa branca. Na verdade, você deve realizar ambos, para garantir que seus objetos funcionem corretamente. Embora cada classe escrita deva ter um teste de unidade correspondente, você provavelmente deve escrever o caso de teste antes de escrever a classe. Você vai ler mais sobre esse ponto posteriormente.

Hoje, focalizaremos o teste de unidade, pois ele é fundamental para a escrita de software OO confiável. Na verdade, você deve realizar os testes de unidade por todo o desenvolvimento.

## Teste de integração

Os sistemas OO são constituídos de objetos que interagem. Enquanto os testes de unidade examinam cada classe de objeto isoladamente, os testes de integração verificam se os objetos que compõem seu sistema interagem corretamente. O que poderia funcionar isoladamente pode não funcionar quando combinado com outros objetos! As fontes comuns de erros de integração são provenientes de erros ou mal-entendidos a respeito dos formatos de entrada/saída, conflitos de recurso e seqüência incorreta de chamadas de método.

**Novo Termo**

Um *teste de integração* verifica se dois ou mais objetos funcionam em conjunto corretamente.

Assim como os testes de unidade, os testes realizados durante os testes de integração podem ser baseados nos conceitos de caixa branca e de caixa preta. Você deve ter um teste de integração para cada iteração importante no sistema.

## Teste de sistema

Os testes de sistema verificam se o sistema inteiro funciona conforme descrito pelos casos de uso. Enquanto executa testes de sistema, você também deve testar o sistema de maneiras não descritas pelos casos de uso. Fazendo isso, você pode verificar se o sistema manipula e se recupera normalmente de condições imprevistas.

**DICA**

Tente fazer estes testes em seu sistema:

**Testes de ação aleatória**

Os testes de ação aleatória consistem em tentar executar operações em ordem aleatória.

**Testes de banco de dados vazio**

Os testes de banco de dados vazio garantem que o sistema pode falhar normalmente, caso exista um problema maior no banco de dados.

**Casos de uso mutantes**

Um caso de uso mutante transforma um caso de uso válido em um caso de uso inválido e garante que o sistema possa se recuperar corretamente da interação.

Você ficaria surpreso com o que um usuário pode tentar fazer com seu sistema. Ele pode não cair sob um dos casos de uso ‘normais’; portanto, é melhor estar preparado para o pior.

**Novo Termo**

Um *teste de sistema* examina o sistema inteiro. Um teste de sistema verifica se o sistema funciona conforme mencionado nos casos de uso e se ele pode manipular normalmente situações incomuns e inesperadas.

Os testes de sistema também incluem testes de esforço e de desempenho. Esses testes garantem que o sistema satisfaça quaisquer requisitos de desempenho e possa funcionar sob as cargas es-

peradas. Se possível, é melhor executar esses testes em um ambiente que corresponda o máximo possível ao ambiente de produção.

Os testes de sistema são um aspecto importante dos testes de aceitação. Os testes de sistema verificam unidades funcionais inteiras simultaneamente, de modo que um único teste pode mexer com muitos objetos e subsistemas diferentes. Para sair de uma iteração, o sistema deve passar nesses testes com êxito.

## Teste de regressão

Um teste é válido apenas enquanto o que for testado não mudar. Quando um aspecto do sistema mudar, essa parte — assim como todas as partes dependentes — deverá ser novamente testada. Teste de regressão é o processo de repetição dos testes de unidade, integração e de sistema após as alterações serem feitas.

### Novo Termo

Os *testes de regressão* examinam as alterações nas partes do sistema que já foram validadas. Quando uma alteração é feita, a parte que foi alterada — assim como todas as partes dependentes — deve ser novamente testada.

É absolutamente fundamental testar novamente, mesmo depois de uma pequena alteração. Uma pequena alteração pode introduzir um erro que poderia danificar o sistema inteiro. Felizmente, para executar o teste de regressão basta executar novamente seus testes de unidade, integração e sistema.

## Um guia para escrever código confiável

Embora cada forma de teste seja importante para a qualidade global de seu software, hoje você vai focalizar o que pode fazer de melhor em seu trabalho diário para garantir a qualidade dos sistemas que escreve. Para escrever código confiável, você precisa fazer o teste de unidade, aprender a diferenciar entre condições de erro e erros, e escrever documentação útil.

## Combinando desenvolvimento e teste

Um fato esquecido na Figura 14.1 é que os testes devem ser um processo dinâmico. O teste não deve ser algo a ser evitado, inserido somente no final, feito por outra pessoa ou completamente ignorado. Na verdade, pode ser impossível começar a testar de repente, quando o sistema estiver pronto. Em vez disso, você precisa aprender a começar a testar enquanto desenvolve. Para testar enquanto desenvolve, você precisa escrever testes de unidade para cada classe que criar.

14

## Um exemplo de teste de unidade

Vamos ver um teste de unidade para a classe `SavingsAccount`, apresentada pela primeira vez no Capítulo 5, “Herança: hora de escrever algum código”. A Listagem 14.1 apresenta a classe `SavingsAccountTest`.

**LISTAGEM 14.1** SavingsAccountTest.java

```
public class SavingsAccountTest {  
  
    public static void main( String [] args ) {  
        SavingsAccountTest sat = new SavingsAccountTest();  
        sat.test_applyingInterest();  
    }  
  
    public void test_applyingInterest() {  
  
        SavingsAccount acct = new SavingsAccount( 10000.00, 0.05 );  
  
        acct.addInterest();  
  
        print_getBalanceResult( acct.getBalance(), 10500.00, 1 );  
    }  
  
    private void print_getBalanceResult( double actual, double expected, int  
➥test ) {  
  
        if( actual == expected ) { // passou  
  
            System.out.println( "PASS: test #" + test + " interest applied  
➥properly" );  
            System.out.println();  
        } else { // falhou  
  
            System.out.println( "FAIL: test #" + test + " interest applied  
➥incorrectly" );  
            System.out.println( "Value returned: " + actual );  
            System.out.println( "Expected value: " + expected );  
            System.out.println();  
        }  
    }  
}
```

---

SavingsAccountTest é um teste de unidade que examina a classe SavingsAccount. Um teste de unidade pode ser constituído de vários casos de teste. Aqui, a classe SavingsAccountTest só tem um caso de teste: test\_applyingInterest. O caso de teste test\_applyingInterest verifica se uma instância de SavingsAccount aplica corretamente os juros em seu saldo. Um teste de unidade pode ter muitos casos de teste, mas cada caso de teste deve verificar apenas um recurso do objeto.

`SavingsAccountTest` é referido como teste de unidade porque examina o bloco de construção ou unidade de nível mais baixo no mundo OO: o objeto. Um teste de unidade deve examinar apenas um objeto por vez. Isso significa que cada objeto deve ser testado, assim como uma entidade independente. Se não for assim, você não poderá testar o objeto. Mesmo que isso seja possível, você poderá acabar testando inadvertidamente muitos objetos ao mesmo tempo.

Ao escrever testes de unidade, você deve evitar o máximo possível a validação manual da saída. Conforme você pode ver em `test_applyingInterest`, o caso de teste realiza toda a validação necessária automaticamente. Freqüentemente, a validação manual é demorada e propensa a erros. Ao executar casos de teste, você quer resultados precisos o mais rápido possível e com o menor esforço. Caso contrário, você poderá começar a negligenciar os testes.

## Por que você deve escrever testes de unidade

Os testes de unidade o ajudam a detectar erros. Se você danificar algo em sua classe, saberá imediatamente, pois o teste de unidade lhe informará. O teste de unidade é sua primeira linha de defesa contra erros. A captura de um erro em nível de unidade é muito mais fácil de manipular do que tentar rastrear um erro durante o teste de integração ou de sistema.

Os testes de unidade também permitem que você saiba quando terminou de escrever uma classe: uma classe está pronta quando todos os seus testes de unidade passam! Como desenvolvedor, pode ser extremamente útil ter tal ponto final bem definido. Caso contrário, poderá ser difícil saber quando uma classe está ‘pronta’. Saber que você terminou e pode prosseguir evita que fique tentado a acrescentar mais funcionalidade do que precisa em uma classe.

A escrita de testes de unidade pode ajudá-lo a pensar a respeito do projeto de suas classes, especialmente se você escrever seu caso de teste antes de escrever a classe; entretanto, para escrever o caso de teste, você precisa usar sua imaginação e fingir que a classe já existe. Fazer isso proporciona a você muita liberdade para experimentar a interface de suas classes. Quando terminar de escrever o teste, você poderá escrever a classe e, em seguida, tudo será compilado.

Os testes de unidade podem ajudá-lo a refazer seu código. É mais fácil fazer alterações em seu código, se você tiver um teste de unidade, pois assim tem um retorno instantâneo em relação às suas alterações. Você não precisa se preocupar muito quanto à introdução de erros; você pode simplesmente executar seu teste novamente, para ver se algo foi danificado. Usando testes de unidade, você não precisa ficar assombrado com a aborrecida pergunta, “será que eu danifiquei algo?” Você pode fazer alterações com mais confiança.

Finalmente, você nem sempre pode estar por perto para testar o sistema. Você pode mudar para outros projetos ou outros membros de sua equipe podem precisar realizar os testes. Os testes de unidade permitem que outra pessoa, que não seja o autor, teste um objeto.

## Escrevendo testes de unidade

`SavingsAccountTest e test_applyingInterest()` são exemplos muito simples de teste de unidade e de caso de teste; entretanto, escrever testes de unidade desde o início para cada classe, pode se tornar demorado. Imagine um sistema onde você precise escrever centenas de casos de teste. Se escrever cada teste de unidade desde o início, você acabará fazendo muito trabalho redundante. Você precisa criar ou reutilizar uma estrutura de teste.

**Novo Termo** Uma *estrutura* é um modelo de domínio reutilizável. A estrutura contém todas as classes comuns a um domínio inteiro de problemas e serve como a base para um aplicativo específico no domínio.

As classes de uma estrutura definem o projeto geral de um aplicativo. Como desenvolvedor, você simplesmente estende essas classes e, em seguida, fornece suas próprias classes específicas para o problema, para criar um aplicativo.

No caso de uma estrutura de teste, a estrutura define um esqueleto que você pode reutilizar para escrever e executar testes de unidade. Uma estrutura de teste permite que você escreva testes de unidade rápida e convenientemente, eliminando trabalho redundante e propenso a erros. Lembre-se de que tudo que você programa pode conter erros, até seu código de teste. Ter uma estrutura bem testada pode resolver muitos erros de teste. Sem uma estrutura, a sobrecarga extra dos testes poderia ser suficiente para impedi-lo de fazê-los.

Uma estrutura de teste completa conterá classes base para escrever testes de unidade, suporte interno para automação do teste e utilitários para ajudar a interpretar e relatar a saída. Hoje, você aprenderá a usar JUnit, uma estrutura de teste gratuita, lançada sob a “IBM Public License”, para testar classes Java. Você pode fazer download da JUnit a partir do endereço <http://www.junit.org/>.



O download de JUnit inclui o código-fonte. JUnit tem um projeto excelente e você faria muito bem em estudar o código e a documentação incluída. Em particular, a documentação realiza um excelente trabalho de documentar os padrões de projeto usados para escrever JUnit.

## JUnit

A JUnit fornece classes para escrever testes de unidade, para validar a saída e para executar os casos de teste em uma GUI ou em um ambiente de linha de comando. `junit.framework.TestCase` é a classe base para definir testes de unidade. Para escrever testes de unidade, você simplesmente escreve uma classe que herda de `TestCase`, sobrepor alguns métodos e fornece seus próprios métodos de caso de teste.



Ao escrever casos de teste JUnit, você deve iniciar o nome de qualquer método de caso de teste com `test`. A JUnit fornece um mecanismo que carregará e executará automaticamente qualquer método que comece com `test`.

A Listagem 14.2 apresenta uma versão JUnit de `SavingsAccountTest`.

#### **LISTAGEM 14.2** SavingsAccountTest.java

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class SavingsAccountTest extends TestCase {

    public void test_applyingInterest() {
        SavingsAccount acct = new SavingsAccount(10000.00, 0.05 );
        acct.addInterest();

        Assert.assertTrue( "interest applied incorrectly", acct.getBalance() ==
➥10500.00 );

    }

    public SavingsAccountTest( String name ) {
        super( name );
    }
}
```

A versão JUnit de `SavingsAccountTest` é muito mais simples do que a original. Ao se usar JUnit, não há motivo para reproduzir código de exibição ou testes lógicos. Você pode simplesmente usar a classe `Assert` fornecida pela JUnit. A classe `Assert` fornece vários métodos que recebem um valor booleano como argumento. Se o valor booleano for falso, um erro será gravado. Assim, aqui, o teste passa para `Assert` o valor booleano retornado pela comparação `acct.getBalance() == 10500.00`. Se a comparação for avaliada como falsa, JUnit sinalizará um erro.

Então, como você executa esses testes?

A JUnit oferece a você várias opções para a execução de seus casos de teste. Essas opções caem em duas categorias: *estáticas* e *dinâmicas*. Se você optar por usar o método estático, precisará sobrepor o método `runTest()` para chamar o teste que deseja executar.

Na linguagem Java, o modo mais conveniente é escrever uma classe anônima para que você não precise criar uma classe separada para cada teste que queira executar. A Listagem 14.3 mostra a declaração anônima:

**LISTAGEM 14.3** Uma Anonymous SavingsAccountTest

```
SavingsAccountTest test =  
    new SavingsAccountTest( "test_applyingInterest" ) {  
        public void runTest() {  
            test_applyingInterest();  
        }  
    };  
    test.run();
```

---

As classes anônimas são convenientes porque elas permitem que você sobreponha um método ao instanciar um objeto, tudo sem ter de criar uma classe nomeada em um arquivo separado. Aqui, o método principal instancia um `SavingsAccountTest`, mas sobrepõe o método `runTest()` para executar o caso de teste `test_applyingInterest()`.

**Novo Termo**

Uma *classe anônima* é uma classe que não tem um nome. As classes anônimas não têm nome porque elas são simplesmente definidas ao serem instanciadas. Elas não são declaradas em um arquivo separado ou como uma classe interna.

As classes anônimas são uma excelente escolha para classes usadas uma única vez (se a classe for pequena). Usando uma classe anônima, você evita a necessidade de criar uma classe nomeada separada.

Apesar de serem tão convenientes, as classes anônimas têm deficiências. Você precisará criar uma para cada método de caso de teste que queira chamar. Se você tiver muitos casos de teste, o uso de classes anônimas pode exigir muito código redundante.

Para superar essa deficiência, a JUnit também fornece um mecanismo dinâmico, que procurará e executará qualquer método que comece com `test`. Para os propósitos da lição de hoje, contaremos com esse mecanismo dinâmico.

**NOTA**

A JUnit também fornece um mecanismo, conhecido como *conjunto de teste*, para executar vários casos de teste. A JUnit fornece um mecanismo para definir estaticamente a bateria de testes a serem executados como um conjunto; entretanto, o mecanismo dinâmico pesquisará e encontrará cada método de teste. Hoje, contaremos com esse mecanismo automático para encontrar e executar os testes.

A JUnit também fornece alguns outros recursos convenientes. Considere a versão atualizada de `SavingsAccountTest`, na Listagem 14.4, que também testa o método `withdrawFunds()`.

**LISTAGEM 14.4** SavingsAccountTest.java

```
import junit.framework.TestCase;  
import junit.framework.Assert;
```

**LISTAGEM 14.4 SavingsAccountTest.java (continuação)**

```
public class SavingsAccountTest extends TestCase {  
    private SavingsAccount acct;  
  
    public void test_applyingInterest() {  
        acct.addInterest();  
  
        Assert.assertTrue( "interest applied incorrectly", acct.getBalance() ==  
➥ 10500.00 );  
  
    }  
  
    public void test_withdrawFunds() {  
        acct.withdrawFunds( 500.00 );  
  
        Assert.assertTrue( "incorrect amount withdrawn", acct.getBalance() ==  
➥ 9500.00 );  
  
    }  
  
    protected void setUp() {  
        acct = new SavingsAccount( 10000.00, 0.05 );  
    }  
  
    public SavingsAccountTest(String name) {  
        super( name );  
    }  
}
```

Nessa versão, você notará que o teste contém dois métodos novos: `test_withdrawFunds()` e `setUp()`. `setUp()` sobrepõe um método na classe base `TestCase`. Sempre que a JUnit chamar um método de teste, primeiro ela fará uma chamada a `setUp()` para estabelecer o acessório de teste.

Um *acessório* de teste define o conjunto de objetos sobre os quais um teste operará. Estabelecer um acessório de teste pode consumir a maior parte do tempo que leva para escrever casos de teste.

**Novo Termo**

O acessório de teste prepara o conjunto de objetos sobre os quais um caso de teste atuará. Os acessórios também são convenientes, pois eles permitem que você comunique o mesmo acessório entre um conjunto inteiro de casos de teste, sem ter de duplicar código.

A JUnit garante que os objetos acessórios estarão em um estado conhecido, chamando `setUp()` antes de executar cada teste. A JUnit também fornece um método `tearDown()` correspondente, para realizar toda limpeza do acessório, depois que o teste for executado.

Para executar seus casos de teste, a JUnit fornece executores de teste para exercitar e reunir os resultados de um teste. A JUnit fornece uma versão gráfica e uma versão baseada em linha de comando desse utilitário.

Para executar `SavingsAccountTest` graficamente, basta digitar:

```
java junit.swingui.TestRunner
```

A Figura 14.2 ilustra a janela principal de JUnit.

**FIGURA 14.2**  
*A UI principal de JUnit.*



Usando a UI, você pode navegar até a classe `SavingsAccountTest`. Uma vez carregada, você pode simplesmente executar o teste, pressionando o botão Run.

Conforme a Figura 14.3 mostra, a UI da JUnit exibe o número de testes executados, assim como o número de testes falhos. A UI também fornece uma barra gráfica que mostra se os testes falharam ou não.

A JUnit é uma ferramenta excelente, pois ela permite que você receba retorno claro e instantâneo de seus casos de teste. Assim, se aquela voz em sua mente importuná-lo com, “e se eu danifiquei algo?”, você pode descobrir rapidamente, executando novamente seus testes de unidade.

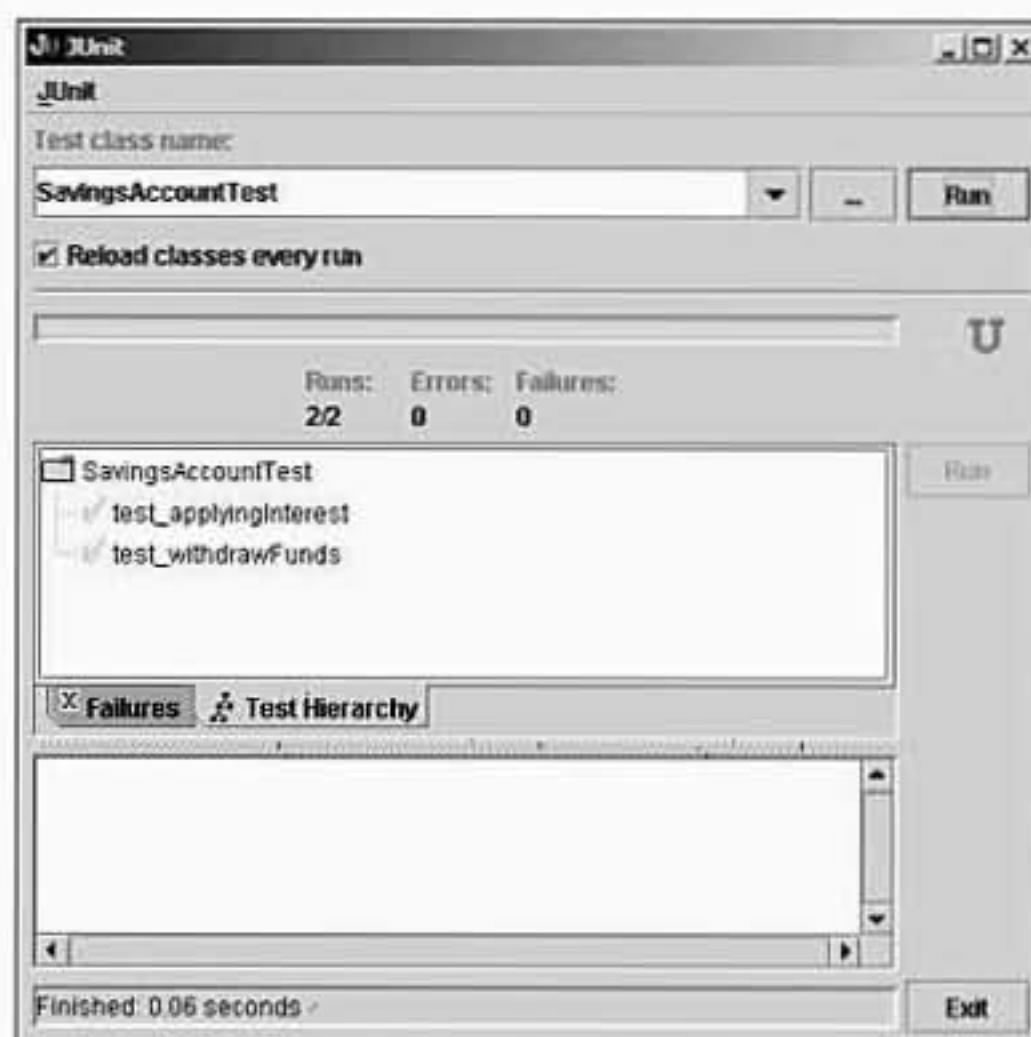
## Escrevendo testes de unidade avançados

Vamos considerar um exemplo ligeiramente mais complicado. O Exercício 1 do Capítulo 11, “Reutilizando projetos através de padrões de projeto”, apresentou uma possível implementação de um `Item`. Atualmente, para apresentar um `Item`, você deve chamar vários métodos de obtenção

e processar os dados para exibição. Infelizmente, solicitar os dados para um objeto não é a melhor estratégia de OO. Você deve pedir ao objeto para que faça algo com seus dados.

**FIGURA 14.3**

A UI principal de JUnit, após executar com êxito os casos de teste.



Considere a Listagem 14.5, que apresenta uma implementação alternativa de Item.

**LISTAGEM 14.5** Item.java

```
public class Item {  
  
    private int id;  
    private int quantity;  
    private float unitPrice;  
    private String description;  
    private float discount;  
  
    public Item( int id, int quantity, float unitPrice, float discount, String  
    desc){  
        this.id = id;  
        this.quantity = quantity;  
        this.unitPrice = unitPrice;  
        this.discount = discount;  
        this.description = desc;  
    }  
  
    public void display( ItemDisplayFormatter format ) {  
        format.quantity( quantity );  
    }  
}
```

**LISTAGEM 14.5** Item.java (*continuação*)

```
    format.id( id );
    format.unitPrice( unitPrice );
    format.discount( discount );
    format.description( description );
    format.adjustedPrice( getTotalPrice() );
}

public float getTotalPrice() {
    return ( unitPrice * quantity ) - ( discount * quantity );
}
}
```

---

Aqui, você pode pedir a `Item` para que se apresente, usando um formatador de exibição. O formatador cuidará da formatação dos dados para exibição. Ter um objeto de formatação separado é uma estratégia melhor do que fazer com que outro objeto chame métodos de obtenção ou incorporar a lógica de exibição no próprio objeto `Item`. Quando os requisitos de exibição mudarem, você poderá simplesmente criar novas implementações de `ItemDisplayFormatter`.

A Listagem 14.6 define a interface `ItemDisplayFormatter` e a Listagem 14.7 apresenta uma possível implementação da interface.

**LISTAGEM 14.6** ItemDisplayFormatter.java

```
public interface ItemDisplayFormatter {
    public void quantity( int quantity );
    public void id( int id );
    public void unitPrice( float unitPrice );
    public void discount( float discount );
    public void description( String description );
    public void adjustedPrice( float total );
    public String format();
}
```

---

**LISTAGEM 14.7** ItemTableRow.java

```
public class ItemTableRow implements ItemDisplayFormatter {

    private int    quantity;
    private int    id;
    private float  unitPrice;
    private float  discount;
```

**LISTAGEM 14.7** ItemTableRow.java (*continuação*)

```
private String description;
private float adjPrice;

public void quantity( int quantity ) {
    this.quantity = quantity;
}

public void id( int id ) {
    this.id = id;
}

public void unitPrice( float unitPrice ) {
    this.unitPrice = unitPrice;
}

public void discount( float discount ) {
    this.discount = discount;
}

public void description( String description ) {
    this.description = description;
}

public void adjustedPrice( float total ) {
    this.adjPrice = total;
}

public String format() {
    String row = "<tr>";
    row = row + "<td>" + id + "</td>";
    row = row + "<td>" + quantity + "</td>";
    row = row + "<td>" + description + "</td>";
    row = row + "<td>$" + unitPrice + "</td>";
    row = row + "<td>$" + adjPrice + "</td>";
    row = row + "</tr>";
    return row;
}
```

O formatador ItemTableRow cria uma representação de linha de tabela HTML para o Item, usando símbolos da moeda norte-americana. Outros formatadores poderiam formatar os dados de outras maneiras.

Esse exemplo apresenta alguns desafios de teste interessantes, assim como algumas oportunidades. Para essas classes, simplesmente chamar um método e verificar uma saída não resolverá.

O teste do método `display()` de `Item` é de especial interesse, pois um teste de unidade só deve verificar uma classe isoladamente; entretanto, para testar `display()`, você também deve passar para ele um `ItemDisplayFormatter`.

Felizmente, os objetos falsificados oferecem uma alternativa que ainda permitirá a você testar `Item` isoladamente.

**NOVO TERMO**

Um *objeto falsificado* é um substituto simplista de um objeto real. Ele é chamado de objeto falsificado porque o objeto foi falsificado para propósitos de teste. Embora o objeto falsificado possa ter uma implementação simplista, ele pode conter funcionalidade extra para ajudar nos testes.

Os objetos falsificados são intimamente relacionados aos objetos stubs. Objetos stubs realizam apenas trocas de mensagens enquanto que os objetos falsificados se diferem no sentido de que eles realmente executam alguma função, em vez de simplesmente aceitarem uma chamada e retornarem algum valor prévio.

Às vezes, os objetos falsificados são chamados de *simuladores*.

Um objeto falsificado fornece um substituto simplista para um objeto real. Esse substituto não aparecerá no sistema real, apenas no código de teste.

O objetivo de um objeto falsificado não é fornecer a funcionalidade real do objeto que imita. Em vez disso, o objeto falsificado deve fornecer uma implementação simplista que pode ter mais suporte para teste.

**ALERTA**

Mantenha os objetos falsificados o mais simples possível. Normalmente, um objeto falsificado deve ser um objeto independente que não conta com quaisquer outros objetos falsificados. Se seu objeto falsificado tiver dependências demais, provavelmente ele é muito complicado.

Por exemplo, considere um acessor de banco de dados que retorna objetos `Item`. Você codificaria o acessor falsificado para retornar o mesmo objeto `Item` repetidamente; entretanto, se você fizer testes de unidade em um objeto que recupera objetos `Item` usando o acessor, ele não saberá a diferença. Tal estratégia isola o objeto que está sendo testado dos defeitos nos objetos que utiliza.

Você pode usar um objeto falsificado para testar se o método `display()` de `Item` usa objetos `ItemDisplayFormatter` corretamente. A Listagem 14.8 apresenta um `ItemDisplayFormatter` falsificado.

**LISTAGEM 14.8** `MockDisplayFormatter.java`

---

```
import junit.framework.Assert;  
  
public class MockDisplayFormatter implements ItemDisplayFormatter {
```

**LISTAGEM 14.8** MockDisplayFormatter.java (*continuação*)

```
private int    test_quantity;
private int    test_id;
private float  test_unitPrice;
private float  test_discount;
private String test_description;
private float  test_adjPrice;

private int    quantity;
private int    id;
private float  unitPrice;
private float  discount;
private String description;
private float  adjPrice;

public void verify() {
    Assert.assertTrue( "quantity set incorrectly", test_quantity == quantity );
    Assert.assertTrue( "id set incorrectly", test_id == id );
    Assert.assertTrue( "unitPrice set incorrectly", test_unitPrice ==
➥unitPrice );
    Assert.assertTrue( "discount set incorrectly", test_discount == discount );
    Assert.assertTrue( "description set incorrectly", test_description ==
➥description );
    Assert.assertTrue( "adjPrice set incorrectly", test_adjPrice == adjPrice );
}

public void test_quantity( int quantity ) {
    test_quantity = quantity;
}

public void test_id( int id ) {
    test_id = id;
}

public void test_unitPrice( float unitPrice ) {
    test_unitPrice = unitPrice;
}

public void test_discount( float discount ) {
    test_discount = discount;
}

public void test_description( String description ) {
    test_description = description;
}
```

**LISTAGEM 14.8** MockDisplayFormatter.java (*continuação*)

```
public void test_adjustedPrice( float total ) {
    test_adjPrice = total;
}

public void quantity( int quantity ) {
    this.quantity = quantity;
}

public void id( int id ) {
    this.id = id;
}

public void unitPrice( float unitPrice ) {
    this.unitPrice = unitPrice;
}

public void discount( float discount ) {
    this.discount = discount;
}

public void description( String description ) {
    this.description = description;
}

public void adjustedPrice( float total ) {
    this.adjPrice = total;
}

public String format() { // não estamos testando funcionalidade de formatador
    return "NOT IMPLEMENTED";
}
```

---

De certa forma, MockDisplayFormatter é semelhante à implementação real; entretanto, você notará que ele não implementa um verdadeiro método `format()`. Você também notará que ele acrescenta vários métodos para configurar os valores esperados, assim como um método para verificar as entradas de `Item` em relação a esses valores.

A Listagem 14.9 ilustra como você poderia usar a exibição falsificada para fazer o teste de unidade da classe `Item`.

**LISTAGEM 14.9** ItemTest.java

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class ItemTest extends TestCase {

    private Item item;

    // constantes para valores do construtor
    private final static int ID          = 1;
    private final static int QUANTITY    = 10;
    private final static float UNIT_PRICE = 100.00f;
    private final static float DISCOUNT   = 5.00f;
    private final static String DESCRIPTION = "ITEM_TEST";

    protected void setUp() {
        item = new Item( ID, QUANTITY, UNIT_PRICE, DISCOUNT, DESCRIPTION );
    }

    public void test_displayValues() {
        MockDisplayFormatter formatter = new MockDisplayFormatter();
        formatter.test_id( ID );
        formatter.test_quantity( QUANTITY );
        formatter.test_unitPrice( UNIT_PRICE );
        formatter.test_discount( DISCOUNT );
        formatter.test_description( DESCRIPTION );

        float adj_total = ( UNIT_PRICE * QUANTITY ) - ( DISCOUNT * QUANTITY );
        formatter.test_adjustedPrice( adj_total );

        item.display( formatter );

        formatter.verify();
    }

    public ItemTest( String name ){
        super( name );
    }
}
```

O método `test_displayValues()` de `ItemTest` cria um `MockDisplayFormatter`, configura as entradas esperadas, passa isso para o objeto `Item` e usa o formatador para validar a entrada. Internamente, o método `verify()` do formatador usa a classe `Assert` de JUnit para validar a entrada.

Os objetos falsificados são um conceito poderoso, pois você pode programá-los para qualquer coisa. Você pode ter objetos falsificados que contém o número de vezes que um método é chamado ou um que controle o volume de dados que um objeto envia por uma rede. Tudo depende de seu aplicativo e do que você precisa monitorar. Os objetos falsificados permitem que você realize todos os tipos de monitoramento e teste que não são possíveis, caso um objeto simplesmente crie todos os objetos de que precisa.

Isso levanta a questão, “e se meus objetos criarem os objetos de que precisam?” A maior parte dos exemplos deste livro cai nessa categoria (espera-se, contudo, que os exemplos tenham permanecido mais inteligíveis!). Uma solução é editar as classes para que os objetos instanciem um objeto falsificado, em vez do objeto real. Tal estratégia, entretanto, não é limpa, pois o obriga a alterar o código que você está testando. Se você mudar a implementação do código antes de testar, não estará realmente testando a classe que entrará em seu sistema. A melhor solução é escrever código que seja fácil de testar.

Tal conselho pode parecer retrógrado. A sabedoria convencional diz que normalmente você escreve testes para testar código que já escreveu. Você não escreve código para que possa testar! O exemplo `Item` ilustra uma lição importante. Projetar suas classes de modo que elas sejam fáceis de testar pode resultar em um código mais OO! Nesse caso, passar os objetos dependentes tornou o objeto menos dependente de uma classe específica, tornando-o, assim, mais afeito à conexão.

**DICA**

Projete suas classes de modo que você possa testá-las facilmente. Projete lembrando dos objetos falsificados. Seu código pode se tornar mais orientado a objetos!

**DICA**

Escreva suas classes de modo que os objetos dependentes sejam passados e não instanciados dentro do próprio objeto. Essa prática leva a objetos independentes.

Embora seja verdade que o método `display()` de `Item` seja dependente da interface `ItemDisplayFormatter`, ele não é dependente de uma implementação específica da interface, como `TableRowFormatter` ou mesmo `MockDisplayFormatter`. Em vez disso, `Item` está livre para usar qualquer implementação de `ItemDisplayFormatter`, pois ele não se obriga a usar qualquer uma específica, criando a instância por si mesmo.

**DICA****Dicas para testes eficazes:**

- Você deve otimizar a velocidade de seus testes. Testes rápidos fornecem retorno instantâneo; portanto, você poderia estar mais apto a usá-los.
- Compile seus casos de teste junto com suas classes normais. Essa prática o obrigará a manter seus casos de teste atualizados com o código.
- Evite validação visual/manual de saída de teste, pois ela é propensa a erros. Em vez disso, use algum mecanismo automático.

Se você precisa manter uma base de código que não possui testes de unidade, escreva os testes à medida que precisar deles.

## Escrevendo código excepcional

Testar é uma maneira importante de garantir a qualidade do código que você escreve; entretanto, esse não é o único passo que você pode dar. Você também precisa aprender a identificar a diferença entre uma condição de erro e um erro. Uma condição de erro e um erro não são a mesma coisa!

Você sabe o que é um erro; um erro é um defeito. Uma condição de erro é ligeiramente diferente. Uma condição de erro ou exceção é uma falha previsível que acontece sob certas circunstâncias no domínio. Pegue como exemplo a loja on-line. Períodos de inatividade da rede acontecem o tempo todo. Um período de inatividade da rede não é um erro (a não ser que seu código tenha causado isso!). Em vez de tratar condições de erro como erros, você precisa codificar em torno delas.

Por exemplo, se sua conexão com o banco de dados falha, você precisa tentar reconectar. Se ela ainda estiver inativa, você precisará tratar da condição normalmente e informar o usuário do erro.

Toda linguagem tem sua própria maneira de relatar condições de erro. As linguagens Java e C++ empregam um mecanismo conhecido como exceções para sinalizar condições de erro. Linguagens como C contam com códigos de retorno. Qualquer que seja a linguagem usada, você deve escrever seu código para detectar e se recuperar normalmente de condições de erro.

As exceções Java e C++ funcionam de modo semelhante. As exceções são apenas outro tipo de objeto; entretanto, o compilador Java o obriga a tratar delas, se determinar que uma exceção pode ocorrer e você não tratar dela.

Aqui está um dos métodos da classe URL da linguagem Java:

```
public URLConnection openConnection() throws IOException
```

Você vê que o método tem algumas informações extras. O método indica que pode lançar IOException, significando que, sob condições normais, o método retornará um objeto URLConnection. Se houver um erro na abertura da conexão, entretanto, o método lançará uma exceção IOException.

Na linguagem Java, você trata de exceções em blocos try/catch. A Listagem 14.10 mostra como você poderia manipular uma chamada de openConnection.

**LISTAGEM 14.10** Tratando de uma exceção

---

```
java.net.URL url = new java.net.URL("http://www.samspublishing.com/");
java.net.URLConnection conn;
try {
    conn = url.openConnection();
} catch ( java.io.IOException e ) { // um erro ocorreu
    // registra um erro, escreve algo na tela
    // faz algo para tratar do erro
}
```

---

Quando você faz uma chamada para `openConnection`, faz isso normalmente; entretanto, você deve fazer a chamada dentro de blocos `try/catch` ou dizer explicitamente que o método no qual a chamada é feita também lança uma exceção `IOException`.

Se a chamada para `openConnection()` resultar no lançamento de uma exceção, `conn` não será configurado. Em vez disso, a execução continuará dentro do bloco `catch`, onde você pode tentar recuperar, registrar um erro, imprimir uma mensagem na tela ou lançar outra exceção.

Se você não capturar a exceção explicitamente ou lançar uma nova exceção, a exceção subirá como bolha na pilha de chamadas, até chegar ao topo ou que alguém finalmente a capture.

O importante é que você programe para condições de erro, usando o mecanismo incorporado à sua linguagem. Isso significa que, quando você projetar suas classes, também deverá considerar as diversas condições de erro, modelá-las através de objetos exceção e fazer com que seus métodos as lancem.

## Escrevendo documentação eficaz

Há mais um passo que você pode dar para melhorar a qualidade de seu trabalho: documentá-lo. Existem muitas formas de documentação, cada uma com seu próprio nível de eficácia.

### Código-fonte como documentação

O código-fonte, até seus testes de unidade, é uma forma de documentação. Quando outras pessoas precisam pegar e manter seu código, é importante que ele seja legível e bem organizado. Caso contrário, ninguém poderá ter idéia do que ele faz.

O código-fonte é a forma mais importante de documentação, pois é a única documentação que você tem de manter.

### Convenções de codificação

O primeiro passo que você pode dar para transformar seu código em boa documentação é escolher uma convenção de codificação e ficar com ela. As convenções de codificação podem cobrir tudo, desde como você endenta suas chaves até como nomeia suas variáveis. A convenção espe-

cífica não é tão importante assim. O importante é que sua equipe de projeto, e preferivelmente sua empresa, escolha uma convenção e fique com ela. Desse modo, qualquer um pode pegar um trecho de código e acompanhá-lo — bem, pelo menos não se atrapalhar com a formatação.

A Listagem 14.11 apresenta uma maneira de declarar classes Java.

#### **LISTAGEM 14.11** Um exemplo de classe

```
public class <ClassName>
    extends <ParentClassName>
    implements <LIST OF INTERFACES>
{
    // variáveis públicas
    // variáveis protegidas
    // variáveis privadas
    // constantes

    // métodos públicos
    // métodos protegidos
    // métodos privados
}
```



#### DICA

Os nomes de classe sempre devem começar com uma letra maiúscula. Os nomes de método sempre devem começar com uma letra minúscula. Os nomes de variável sempre devem começar com uma letra minúscula.

Qualquer nome contendo várias palavras deve ter as palavras reunidas e cada palavra começar com letra maiúscula.

Por exemplo o método `someMethod()` e a classe `HappyObject`.

As constantes sempre devem aparecer em MAIÚSCULAS. As variáveis normais devem aparecer em minúsculas.

(Note que essas convenções são voltadas à linguagem Java. As convenções para Smalltalk e C++ podem ser diferentes.)

Aqui está uma maneira de declarar métodos e instruções `if/else` aninhadas:

```
public void method() {
    if ( condicional ) {

    } else {
    }
}
```

## Constantes

As constantes também podem servir como uma forma de documentação. Use constantes quando você se achar utilizando um valor codificado. Uma constante bem nomeada pode dar uma idéia do objetivo de seu código.

## Comentários

Assim como uma constante bem colocada, nada ajuda a tornar o código mais inteligível do que um comentário bem colocado; entretanto, você precisa encontrar um equilíbrio em seus comentários. Coloque comentários demais e eles perderão o significado.

Aqui está um erro de comentário inútil, mas comum:

```
public void id( int id ) {  
    this.id = id; // configura id  
}
```

Comentários inúteis informam o que o código está fazendo. Se você precisa explicar código, então ele pode estar complicado demais. Os comentários devem dizer para que serve o código. Eles devem descrever implementações não intuitivas.



Note que um comentário sob uma assinatura de método não substitui um bom e claro nome de método.

## Nomes

Os nomes de variável, método e classe devem ser significativos. Solete-os e seja consistente em sua grafia. Por exemplo, sempre coloque inicial maiúscula na segunda palavra de um nome com várias palavras, como em `testCase`. Como alternativa, você pode dividir as palavras com um hífen (-), como em `test-case`. Novamente, o aspecto mais importante é a consistência. Torne sua regra de atribuição de nomes parte de sua convenção e então a utilize consistentemente.

## Cabeçalhos de método e classe

Quando você escrever uma classe ou um método, sempre se certifique de incluir um cabeçalho. Um cabeçalho de método incluirá uma descrição, uma lista de argumentos, uma descrição do retorno, assim como condições de uma exceção e efeitos colaterais. Um cabeçalho pode até incluir condições prévias. Um cabeçalho de classe normalmente incluirá uma descrição, o número da versão, a lista de autores e o histórico de revisão.

Quando você programar em Java, certifique-se de tirar proveito do Javadoc. (Veja o Apêndice B para mais informações.) O Javadoc fornece várias tags para escrever cabeçalhos. Se você usar Javadoc, poderá simplesmente executar suas classes através de um processador, que fará automaticamente documentação da API baseado na Web, conveniente para suas classes.



Quando você tiver documentado algo, terá se comprometido a manter essa documentação atualizada, seja com o Javadoc, um documento de projeto ou um comentário. Uma documentação obsoleta é inútil. Mantenha-a atualizada!

## Resumo

Hoje, você aprendeu sobre os testes e o que pode fazer como desenvolvedor para garantir a qualidade de seu trabalho. Ao todo, existem quatro formas gerais de teste:

- Teste de unidade
- Teste de integração
- Teste de sistema
- Teste de regressão

Em seu trabalho diário, o teste de unidade é sua primeira linha de defesa contra erros. O teste de unidade também tem as vantagens de obrigá-lo a considerar seu projeto do ponto de vista do teste e de fornecer a você um mecanismo que torna mais fácil refazer as coisas.

Você também viu a importância do tratamento correto das condições de erro e da manutenção da documentação. Todas essas práticas aumentam a qualidade de seu código.

## Perguntas e respostas

**P Por que os desenvolvedores detestam testar?**

**R** Parece haver uma cultura do ‘evitar teste’ entre os programadores. Achamos que esse é um problema cultural. Na maioria das empresas, o pessoal de controle de qualidade é um grupo separado dos desenvolvedores. Eles vêm, testam o sistema e redigem relatórios de erro. Essa estratégia coloca o programador na defensiva, especialmente se o gerente de projeto pressionar o desenvolvedor a refazer o trabalho. De certa forma, os testes se tornam uma punição e uma fonte de trabalho extra.

Ver os testes como trabalho extra também faz parte do problema. Muitas equipes de projeto deixam os testes para o final do desenvolvimento; assim, ele se torna algo que você faz quando já terminou de fazer o trabalho ‘real’. Infelizmente, quanto mais você retarda os testes, mais difíceis eles serão de fazer. Quando você começar a testar, provavelmente encontrará um grande número de erros, pois não tinha testado até esse ponto. Isso o leva de volta à fase de punição, especialmente porque você, provavelmente, está perto de seu prazo final. E se você estiver próximo ao seu prazo final, a pressão do gerente de projeto aumentará.

O evitar teste é um problema que se auto-alimenta.

**P Por que os testes são feitos freqüentemente por um grupo de controle de qualidade separado?**

**R** Testes independentes ajudam a garantir que o pessoal de teste não evite, subconscientemente, áreas onde podem existir problemas — uma falha na qual os desenvolvedores podem estar propensos.

**P Em todas as lições de hoje, você usou JUnit. Por que você escolheu JUnit?**

**R** JUnit é uma ferramenta de teste gratuita que faz bem seu trabalho. A JUnit é bem projetada e é suficientemente pequena para que você possa por suas mãos no projeto facilmente. Ela também não possui os detalhes que muitos outros produtos têm.

Em nossa opinião, evitar os detalhes durante os testes de unidade é uma vantagem. A JUnit o coloca mais perto do código, pois você precisa escrever seus próprios testes, configurar os dados e validar a saída. Ela o obriga a considerar seu projeto e até a aumentá-lo, para que seja fácil testar.

Com algumas das ferramentas de teste mais automatizadas, você pode perder essas vantagens; entretanto, a JUnit é uma estrutura de teste de unidade. Você precisará encontrar outras ferramentas para alguns dos testes de integração e de sistema.

**P O teste de unidade parece um fardo. Não tenho tempo para fazer o teste de unidade. O que devo fazer?**

**R** Os testes de unidade podem parecer um fardo na primeira vez que você escreve um deles. Com toda honestidade, às vezes, os testes de unidade são dispendiosos para escrever. Eles se pagam com o passar do tempo. Quando você escreve um teste, pode reutilizá-lo repetidamente. Sempre que você mudar a implementação de sua classe, bastará executar novamente seu teste de unidade. Os testes de unidade tornam muito mais fácil fazer alterações em seu código.

Não ter tempo é um argumento fraco. Imagine quanto será mais demorado encontrar, rastrear e corrigir erros, se você deixar os testes para o fim — um momento em que você normalmente está sob ainda mais pressão.

**P Como você sabe se já testou o suficiente?**

**R** Você já testou em um nível mínimo, quando tiver um teste de unidade para cada classe, um teste de integração para cada interação importante no sistema e um teste de sistema para cada caso de uso. Se você vai ou não fazer testes adicionais depende de seu prazo final, assim como de seu nível de bem-estar.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

### Teste

1. Como erros podem entrar em seu software? (Ou talvez a pergunta deveria ser, “como você causa erros em seu software?”)
2. O que é um caso de teste?
3. Quais são as duas maneiras nas quais você pode basear seus testes?
4. Defina teste de caixa branca e de caixa preta.
5. Quais são as quatro formas de teste?
6. Defina *teste de unidade*.
7. Qual é o objetivo por trás dos testes de integração e dos testes de sistema?
8. Por que você deve evitar ter de deixar os testes para o final de um projeto?
9. Por que você deve evitar validação manual ou visual ao testar? Qual é a alternativa?
10. O que é uma estrutura?
11. O que é um objeto falsificado?
12. Por que você deve usar objetos falsificados?
13. Qual é a diferença entre um erro e uma condição de erro?
14. Ao escrever seu código, como você pode garantir sua qualidade?

### Exercícios

1. Faça download da JUnit e leia cookstour, que é encontrado no diretório doc.
2. Escreva um teste de unidade para HourlyEmployee do Capítulo 7, “Polimorfismo: hora de escrever algum código”, que testa o método calculatePay().

# SEMANA 2

## Em revisão

Nesta semana, você aprendeu a respeito da estratégia iterativa para o processo de desenvolvimento de software. O processo de desenvolvimento de software inclui os estágios de análise, projeto, implementação e teste. Você aprendeu que a estratégia iterativa para o desenvolvimento de software permite voltar e refinar qualquer estágio do processo. Esse refinamento contínuo leva a uma solução mais completa e correta para seu problema.

O Dia 8 apresentou a Unified Modeling Language. No Dia 9, você aprendeu sobre AOO (Análise Orientada a Objetos), o primeiro passo do processo de desenvolvimento. A AOO permite que você entenda o problema que está tentando resolver. Um método de AOO é através do uso de casos de uso para descrever como os usuários utilizarão o sistema. Uma vez que você tenha seus casos de uso mapeados, pode usar linguagens de modelagem, como a UML, para visualizar graficamente o domínio de seu problema.

O Dia 10 descreveu o POO (Projeto Orientado a Objetos), o processo de pegar o modelo de domínio e criar o modelo de objetos que você usará durante a implementação. A lista a seguir apresenta os passos básicos necessários para completar o POO.

1. Gere uma lista de objetos inicial.
2. Refine as responsabilidades de seus objetos.
3. Desenvolva os pontos de interação.
4. Detalhe os relacionamentos entre objetos.
5. Construa seu modelo.

Os cartões CRC o ajudam no mapeamento das responsabilidades e das interações entre cada objeto. Os dois dias seguintes abordaram os padrões de projeto, conceitos de projeto reutilizável que fornecem atalhos para o POO. Você aprendeu sobre padrões, como Adapter, Proxy, Iterator, Abstract Factory, Singleton e Typesafe Enum, e quando é apropriado usá-los.

O Dia 13 explicou o projeto de UI com o padrão MVC. O padrão MVC fornece flexibilidade, desacoplando a UI do sistema subjacente. Ele também enfatiza a necessidade de executar o processo de projeto de software para a UI como para qualquer outra parte do sistema.

Finalmente, o Dia 14 apresentou os testes nos estágios de implementação e testes de cada iteração. Você aprendeu a respeito das duas maneiras de testar seu código: teste de caixa branca e teste de caixa preta. Você também aprendeu as quatro formas de teste: teste de unidade, de integração, de sistema e de regressão. O teste é a maneira final de garantir a qualidade de seu código. Se ele não for levado em consideração durante todo o processo de desenvolvimento, você poderá se deparar com sérias consequências.

Agora que você terminou as lições desta semana, deve entender o processo de desenvolvimento de software.

PÁGINA EM BRANCO

# SEMANA 3

## Reunindo tudo: um projeto OO completo

- 15 Aprendendo a combinar teoria e processo
- 16 Iteração 2 do jogo vinte-e-um: adicionando regras
- 17 Iteração 3 do jogo vinte-e-um: adicionando aposta
- 18 Iteração 4 do jogo vinte-e-um: adicionando uma GUI
- 19 Aplicando uma alternativa ao MVC
- 20 Divertindo-se com o jogo vinte-e-um
- 21 O último quilômetro

# Panorama

Na primeira semana, você aprendeu a teoria por trás da POO. A segunda semana forneceu um processo para seguir ao aplicar essa teoria. Nesta semana, você reunirá as lições das duas primeiras semanas, em um projeto de POO completo. Esse projeto fará você percorrer todas as fases do desenvolvimento — do início ao fim do projeto.

Seu projeto final será o desenvolvimento do jogo de cartas vinte-e-um. O Dia 15, o primeiro dia desta semana, o conduzirá na primeira iteração do jogo e adicionará as funcionalidades básicas da AOO, através de implementação e teste. Nos dias 16 e 17, você vai adicionar mais funcionalidades no jogo, através de mais duas iterações. O Dia 18 mostrará como se acrescenta uma GUI ao sistema.

O Dia 19 fornecerá uma outra maneira de implementar uma GUI, através do padrão de projeto PAC. No Dia 20, você vai aprender a adicionar vários jogadores não-humanos e verá como transformar seu jogo em um simulador. Finalmente, o Dia 21 reunirá tudo e retirará todas as aparas.

# SEMANA 2

DIA **15**

## Aprendendo a combinar teoria e processo

A Semana 1 o ajudou a entender a teoria por trás da POO. A Semana 2 forneceu um processo para seguir ao aplicar essa teoria. A Semana 3 mostrará a você como reunir as lições das semanas 1 e 2, apresentando um projeto de POO completo. Esse projeto o conduzirá por todas as fases do desenvolvimento, do início ao fim do projeto.

A lição de hoje apresentará o jogo vinte-e-um, um jogo de cartas popular. No final da lição de hoje, você completará a análise, projeto e implementação funcional iniciais do jogo. Você não criará um jogo vinte-e-um inteiro em uma única etapa grande. Em vez disso, você vai aplicar o processo iterativo durante a semana, à medida que completa o jogo vinte-e-um.

Hoje você aprenderá como:

- Aplicar análise e projeto OO em um jogo de cartas real
- Usar o processo iterativo para obter e ver resultados rápidos
- Evitar muitas tentações comuns do desenvolvimento
- Evitar características procedurais que podem penetrar em seu programa

### Jogo Vinte-e-um

O vinte-e-um é um jogo de cartas popular, onde o objetivo é obter a contagem mais alta de cartas do que a banca, sem ultrapassar 21. O objetivo desta semana é criar um jogo vinte-e-um bastante

completo, escrito em Java, usando os conceitos de POO que você aprendeu por todo o livro. Embora alguns recursos sejam omitidos por motivos de clareza e brevidade, você deverá ter um jogo tão viciante e demorado quanto os jogos de paciência ou FreeCell que acompanham muitos sistemas operacionais populares.



O autor não se responsabiliza pelo tempo perdido enquanto se joga este jogo!

## Por que vinte-e-um?

A pergunta pode vir à mente, “por que vinte-e-um?”

O objetivo de programar o vinte-e-um não é transformá-lo em um jogador profissional. Existem dois motivos para se usar o vinte-e-um como um projeto de POO introdutório:

- Quase todo mundo está familiarizado com pelo menos um jogo de cartas.
- Acontece que o jogo vinte-e-um se encaixa bem no paradigma da POO.

Em geral, a maioria das pessoas está familiarizada com o domínio do jogo de cartas. Uma parte importante da AOO e do POO é ter um especialista no domínio presente, enquanto se identifica os casos de uso e o modelo de domínio. Simplesmente não é possível conseguir um especialista no domínio como parte da conclusão destas lições.

Um jogo de cartas não exige um especialista no domínio. Em vez disso, sua própria experiência e um bom livro de regras ou site Web fornece tudo de que você precisa para completar a análise e o projeto por conta própria. Se ficar confuso, você pode até pegar um baralho e simular um cenário.

Como resultado, um jogo de cartas é muito mais acessível do que tentar aprender um domínio completamente desconhecido. Um domínio conhecido não o desviará do objetivo real destas lições — aprender a aplicar os princípios da OO.

O vinte-e-um — na verdade, como a maioria dos jogos de cartas em geral — também tende a se encaixar bem no paradigma da POO. As interações entre bancas, jogadores, suas mãos e as cartas podem ajudá-lo a ver que um sistema OO é constituído das interações entre os vários objetos.

Os jogos de cartas também tendem a manter características procedurais suficientes para que seja fácil ver como uma estratégia de POO pode ser muito diferente de uma procedural, especialmente ao se aplicar regras de jogo.

## Declaração da visão

Ao se iniciar um projeto, freqüentemente ajuda começar com uma declaração da visão ou objetivo. O objetivo da declaração é formar o propósito geral do sistema que você vai criar. O objetivo

da declaração não é capturar cada aspecto do problema que você está tentando resolver. Em vez disso, a declaração simplesmente informa o intento e fornece um ponto de partida para a análise.

**Novo Termo**

A *declaração da visão* forma o propósito geral do sistema que você vai criar e do problema que está tentando resolver.

Aqui está a declaração da visão do sistema do jogo vinte-e-um:

O jogo vinte-e-um permite que um jogador participe dele, de acordo com regras de cassino comuns.

Embora seja aparentemente evidente, essa declaração serve como um catalisador excelente para a análise.

15

## Requisitos de sobreposição

Antes de iniciar a análise, ajuda enumerar todos os requisitos de sobreposição. Existem apenas algumas restrições no jogo vinte-e-um. A única restrição importante agora é como o usuário vai interagir com o sistema.

O jogo vinte-e-um permite que o usuário interaja com o sistema através de uma linha de comando ou UI gráfica. É claro que as interações iniciais permitirão apenas uma interação através de uma interface de linha de comando.

O jogo vinte-e-um também deve ser implementado na linguagem de programação Java.

É importante listar tais restrições antecipadamente, para que você não apareça com um projeto que torne a obediência a essas restrições impossível.

## Análise inicial do jogo vinte-e-um

O Capítulo 9, “Introdução à (AOO) Análise Orientada a Objetos”, apresentou a AOO, assim como o processo de desenvolvimento iterativo. De acordo com as informações do Capítulo 9, este projeto seguirá um processo de desenvolvimento iterativo e iniciará cada iteração desse processo com a análise.

Ao se iniciar a análise, freqüentemente é útil começar com a declaração da visão mencionada anteriormente:

O jogo vinte-e-um permite que um jogador participe dele, de acordo com regras de cassino comuns.

Você pode usar a declaração para gerar uma lista inicial de perguntas. São essas perguntas que dirigirão sua análise.

## As regras do jogo vinte-e-um

A partir da declaração da visão do jogo vinte-e-um, você pode naturalmente perguntar, “quais são as regras de cassino comuns do jogo vinte-e-um?” Você vai usar essas regras para modelar o jogo vinte-e-um.

O objetivo principal do jogo vinte-e-um é reunir uma mão de cartas cujo valor seja maior que a mão da banca, sem ultrapassar 21. Cada carta recebe um valor numérico de 1 a 11, onde o ás vale 1 ou 11 (dependendo do que ofereça a você uma mão melhor), as cartas numéricas valem os seus respectivos números e todas as cartas de figura valem 10. O naipe não tem relação com o valor da carta.

A Figura 15.1 mostra alguns exemplos de mãos.

É claro que o jogo real é um pouco mais complicado. Vamos ver cada componente importante do jogo.

### Aposta

Antes que a banca distribua as cartas, cada jogador deve fazer uma aposta. Normalmente, a aposta está sujeita a algum limite, como US\$25 ou US\$50 por jogo.

### Distribuindo as cartas

Após cada jogador ter feito uma aposta, a banca pode distribuir as cartas. Começando com o primeiro jogador, a banca distribui uma carta aberta, para cada jogador, terminando consigo mesmo.

Então, a banca repete esse processo, mas distribui sua própria carta fechada. A carta fechada da banca é conhecida como carta oculta.

A distribuição de cartas termina quando a banca servir a cada jogador, incluindo ela mesma, duas cartas. O jogo começa quando a distribuição terminar.

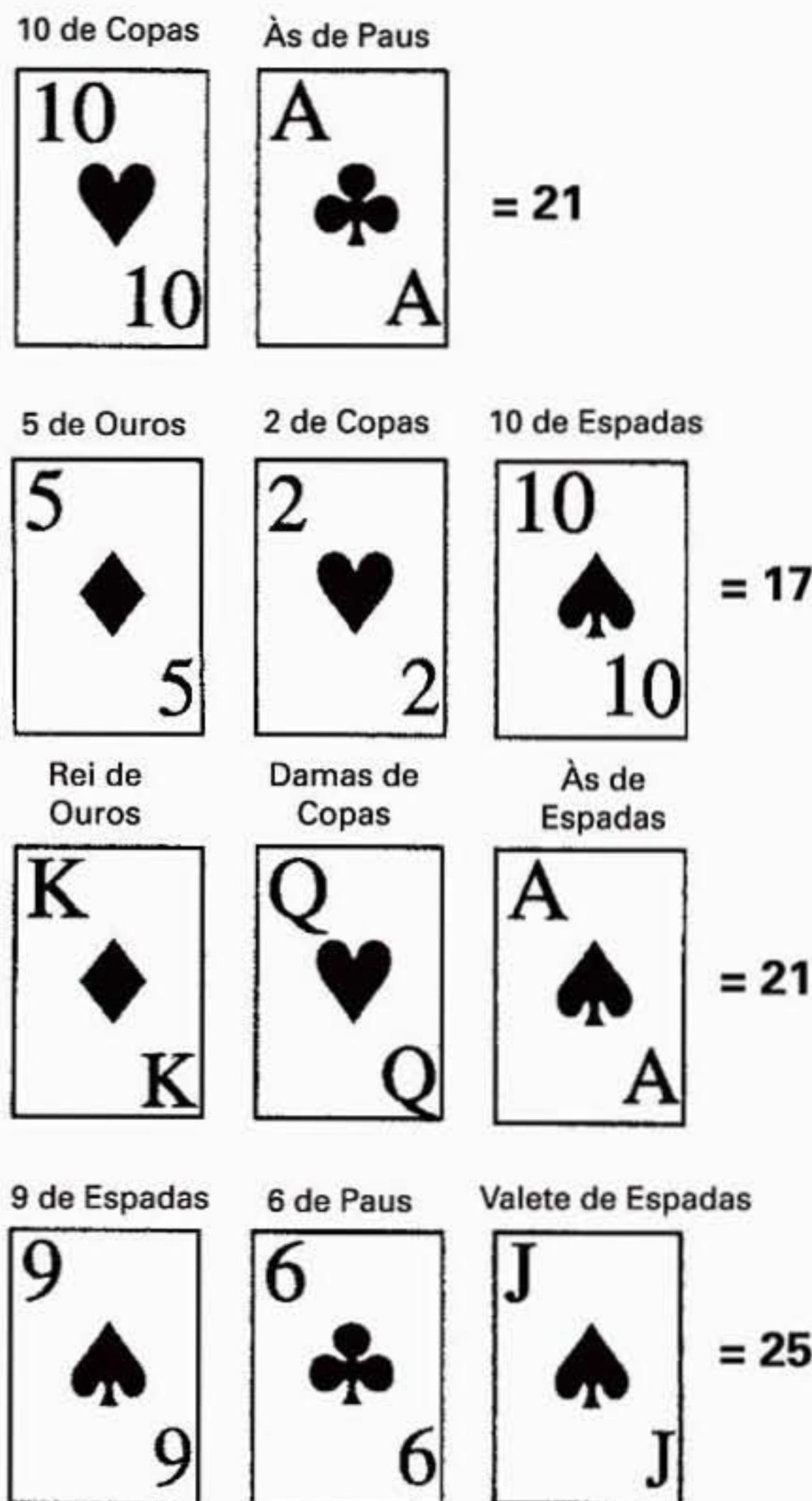
### Jogando

O jogo pode diferir dependendo da carta aberta da banca. Se a carta aberta da banca valer 10 (chamado de valor 10) ou for um ás (valor 11), a banca deverá verificar sua carta oculta. Se a carta oculta resultar em um total de 21 (chamado de 21 ou vinte-e-um natural), o jogo terminará automaticamente e se passará para o pagamento. Se a carta oculta não resultar em um total de 21, o jogo começará normalmente.

Se a carta aberta da banca não for de valor 10 ou um ás, o jogo passará automaticamente para o primeiro jogador. Se o jogador tiver um 21 natural, ele terá batido e o jogo passa para o jogador seguinte. Se o jogador não tiver 21, terá duas escolhas: receber mais cartas ou parar:

**FIGURA 15.1**

*Exemplo de mãos do jogo vinte-e-um.*



**receber mais cartas** — Se o jogador não estiver satisfeito com sua mão, ele pode optar por tirar outra carta, o que é chamado de receber mais cartas. O jogador pode receber mais cartas até ultrapassar 21 (estouro) ou parar (não precisa de mais cartas).

**parar** — Se o jogador estiver satisfeito com sua mão, ele pode optar por parar e não receber mais cartas.

Após o jogador estourar ou parar, o jogo passa para o jogador seguinte. Esse processo se repete até que cada jogador tenha jogado. Após cada jogador participar, a banca então joga suas cartas. Quando a banca completa sua vez, o jogo passa para o pagamento, onde ocorre a contagem de pontos e o pagamento.

## Pagamento

Após a banca terminar sua vez (ou saber que tem vinte-e-um pontos), o jogo passa para o pagamento. Durante o pagamento, cada jogador que estourou perde sua aposta; cada jogador com uma mão menor que a da banca perde sua aposta; cada jogador com uma mão melhor que a da banca ganha sua aposta; e cada jogador com uma contagem igual à da banca empata e nenhum pagamento é feito. As apostas são pagas uniformemente.

Se um jogador tiver um vinte-e-um e a banca não, o jogador receberá uma vez e meia o que apostou. Por exemplo, se o jogador tivesse apostado US\$100, receberia US\$150 ( $[100 * 3]/2$ ).

## Miscelânea

Precisamos mencionar mais alguns detalhes importantes sobre o jogo vinte-e-um:

**O baralho** — O jogo vinte-e-um é jogado com quatro baralhos padrão de 52 cartas. Esses quatro baralhos são combinados em uma única pilha de cartas grande.

**Número de jogadores** — De um a sete jogadores podem jogar vinte-e-um.

**Dobrando** — Após receber suas duas cartas, o jogador pode optar por dobrar. Quando o jogador dobra, ele duplica sua aposta, recebe mais uma carta e termina sua vez.

**Seguro** — Quando a carta aberta da banca é um ás, o jogador pode optar por fazer uma aposta segura. A aposta segura é igual à metade da aposta original. Se a carta oculta da banca fornecer à banca um 21 natural, o jogador não ganha nem perde. Se a carta oculta da banca não fornecer a ela o 21, o jogador perderá sua aposta segura.

**Dividindo pares** — Diz-se que um jogador tem um par, se as duas cartas iniciais distribuídas tiverem o mesmo valor. Se o jogador recebeu um par, ele pode optar por desdobrar as cartas em duas novas mãos. Se o jogador dividir o par, a banca distribuirá a cada nova mão mais uma carta e, em seguida, o jogador deverá colocar uma aposta igual na nova mão. Um jogador pode dividir qualquer par que resulte de uma divisão subsequente (exceto quanto a um par de ases). Digno de nota também é o fato de que qualquer contagem 21 resultante de uma divisão não é tratada como um 21 natural. Uma vez feita a divisão, o jogador aposta normalmente ou pára, para cada mão.

## Identificando os atores

Existem dois motivos para se fazer a análise. Através da análise, você cria o modelo de caso de uso: a descrição de como o usuário vai usar o sistema. Através da análise, você também cria o modelo de domínio: uma descrição do vocabulário principal do sistema.

O primeiro passo na definição de seus casos de uso é definir os atores que usarão o sistema. A partir da descrição da última seção, é fácil ver que existem dois atores principais no jogo vinte-e-um — o(s) jogador(es) e a banca.

Esses dois atores respondem à pergunta, “quem usará principalmente o sistema?” Através dos casos de uso, você pode definir como esses atores usarão o sistema.

## Criando uma lista preliminar de casos de uso

Uma maneira eficaz de gerar os casos de uso iniciais é perguntar o que cada um desses atores pode fazer.

Os jogadores podem fazer o seguinte:

1. Fazer apostas.
2. Pedir mais cartas.
3. Parar.
4. Estourar.
5. Conseguir um vinte-e-um.
6. Fazer um seguro.
7. Dividir pares.
8. Dobrar.
9. Decidir jogar novamente.
10. Sair.

A banca pode fazer o seguinte:

1. Distribuir cartas.
2. Efetuar o jogo.
3. Receber mais cartas.
4. Parar.
5. Estourar.
6. Conseguir um vinte-e-um.

Podem existir mais casos de uso, mas essa lista fornece muito material inicial.

## Planejando as iterações

O Capítulo 9 apresentou o processo de desenvolvimento iterativo. Seguir um processo de desenvolvimento iterativo permitirá que você construa rapidamente uma implementação de vinte-e-um simples. Cada iteração continuará a acrescentar mais funcionalidades no jogo.

Tal estratégia possibilita resultados rápidos e quantificáveis. Seguir tal estratégia permite a você tratar dos problemas conforme eles aparecerem — não tudo de uma vez, no final do desenvolvimento. Uma estratégia iterativa impede que você seja atropelado por uma avalanche de problemas no final do desenvolvimento.

Fundamental para um desenvolvimento iterativo é o planejamento das iterações o melhor que você puder, desde o início. Quando novos fatos se apresentarem, é totalmente aceitável rever o plano; entretanto, um esboço das iterações desde o início, proporciona uma orientação para o projeto, objetivos e, mais importante, dá uma idéia do que se vai obter quando os objetivos forem atingidos.

Normalmente, você planeja suas iterações classificando os casos de uso pela importância. Ao trabalhar com um cliente, é melhor deixá-lo classificar a importância de cada caso de uso. No caso do jogo vinte-e-um, você deve classificar os casos de uso com base em sua importância para o jogo. Escolha os casos de uso absolutamente necessários para jogar e faça esses casos de uso primeiro. Outros casos de uso podem esperar iterações posteriores. Tal estratégia permite que você crie um jogo funcional o mais rápido possível.

Para os propósitos do projeto do jogo vinte-e-um, existirão quatro iterações principais.

## **Iteração 1: jogo básico**

A Iteração 1 criará o jogo básico. Ela refinará e implementará os casos de uso preliminares a seguir.

Casos de uso do jogador:

1. Receber mais cartas.
2. Parar.
3. Estourar.

Casos de uso da banca:

1. Distribuir cartas.
2. Receber mais cartas.
3. Parar.
4. Estourar.

No final da Iteração 1, você deverá ter um jogo que é jogado na linha de comando. O jogo terá dois participantes: a banca e um jogador. A banca distribuirá as cartas e permitirá que cada jogador receba mais cartas, até decidir parar ou estourar. Após cada jogador jogar, o jogo terminará.

## Iteração 2: regras

A Iteração 2 adicionará regras ao jogo. A Iteração 2 refinará e implementará os casos de uso preliminares a seguir:

Casos de uso do jogador:

1. Conseguir um vinte-e-um.

A banca pode:

1. Efetuar o jogo (detectar ganhadores, perdedores e empates).
2. Conseguir um vinte-e-um.

No final da Iteração 2, tudo da iteração 1 ainda funcionará. Além disso, o jogo detectará e indicará quando um jogador tiver um vinte-e-um, estourar, parar, ganhar, perder e empatar.

15

## Iteração 3: aposta

A Iteração 3 acrescentará aposta básica e em dobro ao jogo. A Iteração 3 refinará e implementará os casos de uso preliminares a seguir:

Casos de uso do jogador:

1. Fazer apostas.
2. Dobrar.

Casos de uso da banca:

1. Efetuar o jogo (para aposta).

No final da Iteração 3, tudo das Iterações 2 e 1 ainda funcionará. Além disso, o jogo permitirá aposta básica e em dobro.

## Iteração 4: interface com o usuário

A Iteração 4 dará alguns toques finais na UI de linha de comando e construirá uma UI gráfica. A Iteração 4 refinará e implementará os casos de uso preliminares a seguir:

Casos de uso do jogador:

1. Decidir jogar novamente.
2. Sair.

**NOTA**

Para os propósitos deste projeto, a aposta segura e a divisão de pares foram omitidas. Esses recursos são deixados como exercício para o leitor. Vinte-e-um é um jogo com muitas variantes. Freqüentemente, o seguro não é permitido e a divisão complica demasiadamente o sistema. Considere essa variante como um novo livro deste, dedicado ao jogo vinte-e-um!

Lembre-se apenas de que você ouviu falar sobre isso primeiro aqui.

## Iteração 1: jogo básico

A lição de hoje o conduzirá através da primeira iteração do jogo de cartas vinte-e-um. No final da lição de hoje, você terá o esqueleto básico de um jogo vinte-e-um.

**NOTA**

Antes de cada seção, talvez você queira deixar o livro de lado e tentar fazer a análise, projeto ou implementação.

Se você fizer suas próprias experiências, certifique-se de voltar, ler cada seção e comparar seu trabalho com o material apresentado, antes de continuar. Tente avaliar criteriosamente todas as discrepâncias entre sua solução e a solução do livro. Embora você possa ter uma solução superior (existem muitas maneiras de encarar esse jogo). Certifique-se de estar correto e de que sua solução segue os princípios da POO.

## Análise do jogo vinte-e-um

A iteração de hoje refinará e implementará os casos de uso a seguir:

Casos de uso do jogador:

1. Receber mais cartas.
2. Parar.
3. Estourar.

Casos de uso da banca:

1. Distribuir cartas.
2. Receber mais cartas.
3. Parar.
4. Estourar.

Após ter um conjunto de casos de uso refinados, você pode criar um modelo inicial do domínio e iniciar o projeto.

## Refinando os casos de uso

Vamos começar com o caso de uso de distribuição de cartas da banca, pois essa ação inicia o jogo.

Primeiro, você precisa descrever o caso de uso em um parágrafo:

15

Começando com o primeiro jogador, a banca distribui uma carta aberta para cada jogador, terminando com ela mesma. Então, a banca repete esse processo, mas distribui sua própria carta fechada. A distribuição de cartas termina e então o jogo começa, quando a banca tiver distribuído para cada jogador, incluindo ela mesma, duas cartas.

- Distribuição de cartas:
  1. A banca distribui uma carta aberta para cada jogador, incluindo ela mesma.
  2. A banca distribui uma segunda carta aberta para todos os jogadores, menos para ela mesma.
  3. A banca distribui uma carta fechada para ela mesma.
- Condições prévias:
  - Novo jogo.
- Condições posteriores:
  - Todos os jogadores e a banca têm uma mão com duas cartas.

Os casos de uso *Jogador recebe mais cartas* e *Jogador pára* seguem naturalmente. Vamos começar com o caso de uso *Jogador recebe mais cartas*:

O jogador decide que não está satisfeito com sua mão. O jogador ainda não estourou; portanto, ele decide receber mais cartas. Se o jogador não estourar, ele pode optar por receber mais cartas novamente ou parar. Se o jogador estourar, o jogo passará para o jogador seguinte.

- Jogador recebe mais cartas:
  1. O jogador decide que não está satisfeito com sua mão.
  2. O jogador solicita outra carta da banca.
  3. O jogador pode decidir receber mais cartas novamente ou parar, se o total em sua mão for menor ou igual a 21.
- Condições prévias:
  - O jogador tem uma mão cujo valor total é menor ou igual a 21.
- Condições posteriores:
  - Uma nova carta é acrescentada na mão do jogador.
- Alternativa: o jogador estoura

Uma nova carta faz a mão do jogador ultrapassar 21. O jogador estoura (perde). Em seguida, começa a vez do próximo jogador/banca.

*Jogador pára* é um caso de uso simples:

O jogador decide que está satisfeito com sua mão e pára.

- Jogador pára:
  1. O jogador decide que está contente com sua mão e pára.
- Condições prévias:
  - O jogador tem uma mão cujo valor é menor ou igual a 21.
- Condições posteriores:
  - A vez do jogador termina.

Neste ponto, está se tornando claro que *Jogador/Banca estoura* não é um caso de uso, pois isso é um subproduto de outras ações. A banca e o jogador nunca executarão uma ação de estouro; entretanto, os jogadores optarão por receber mais cartas ou parar.

Com os casos de uso *Estouro* removidos, apenas os casos de uso *Banca recebe mais cartas* e *Banca pára* permanecem. Vamos começar com o caso de uso *Banca recebe mais cartas*:

A banca deve receber mais cartas se o total em sua mão for menor que 17. Se a banca não estourar após receber mais cartas e o total em sua mão ainda for menor que 17, ela deve receber mais cartas novamente. A banca deve parar em qualquer total maior ou igual a 17. Quando a banca estoura ou pára, o jogo termina.

- Banca recebe mais cartas:
  1. A banca recebe mais cartas se sua mão é menor que 17.
  2. Nova carta acrescentada na mão da banca.
  3. Se o total for menor que 17, a banca deve receber mais cartas novamente.
- Condições prévias:
  - A banca tem uma mão cujo total é menor que 17.
- Condições posteriores:
  - Nova carta na mão da banca.
  - O jogo termina.
- Alternativa: banca estoura
  - A nova carta faz a mão da banca ser maior que 21. A banca estoura.
- Alternativa: banca pára
  - A nova carta faz a mão da banca ser maior ou igual a 17. A banca pára.

Assim como *Jogador pára*, *Banca pára* é relativamente simples:

A banca tem na mão um total maior ou igual a 17 e pára.

- A banca pára:

4. A mão da banca tem um total maior ou igual a 17 e pára.

- Condições prévias:
  - Mão da banca maior ou igual a 17.
- Condições posteriores:
  - Jogo termina.

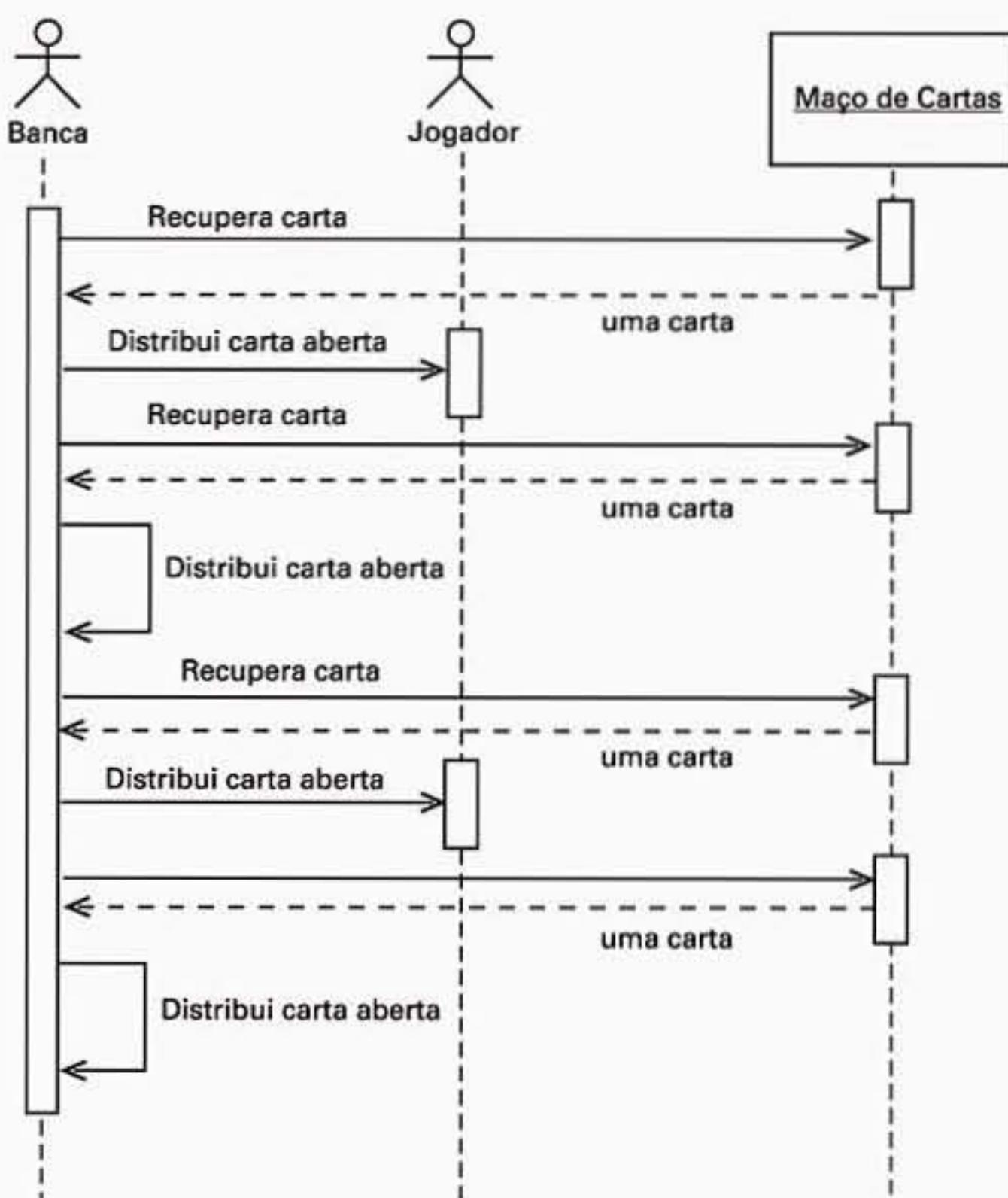
## Modelando os casos de uso

Para a Iteração 1, os casos de uso são muito simples. Os modelos de caso de uso seriam mais pesados do que interessantes; portanto, eles serão omitidos.

As interações entre a banca e os jogadores são um pouco mais interessantes. A Figura 15.2 mostra a seqüência de eventos seguida dentro do caso de uso *Distribuir cartas*. A Figura 15.3 mostra a seqüência de eventos seguida dentro do caso de uso *Jogador recebe mais cartas*.

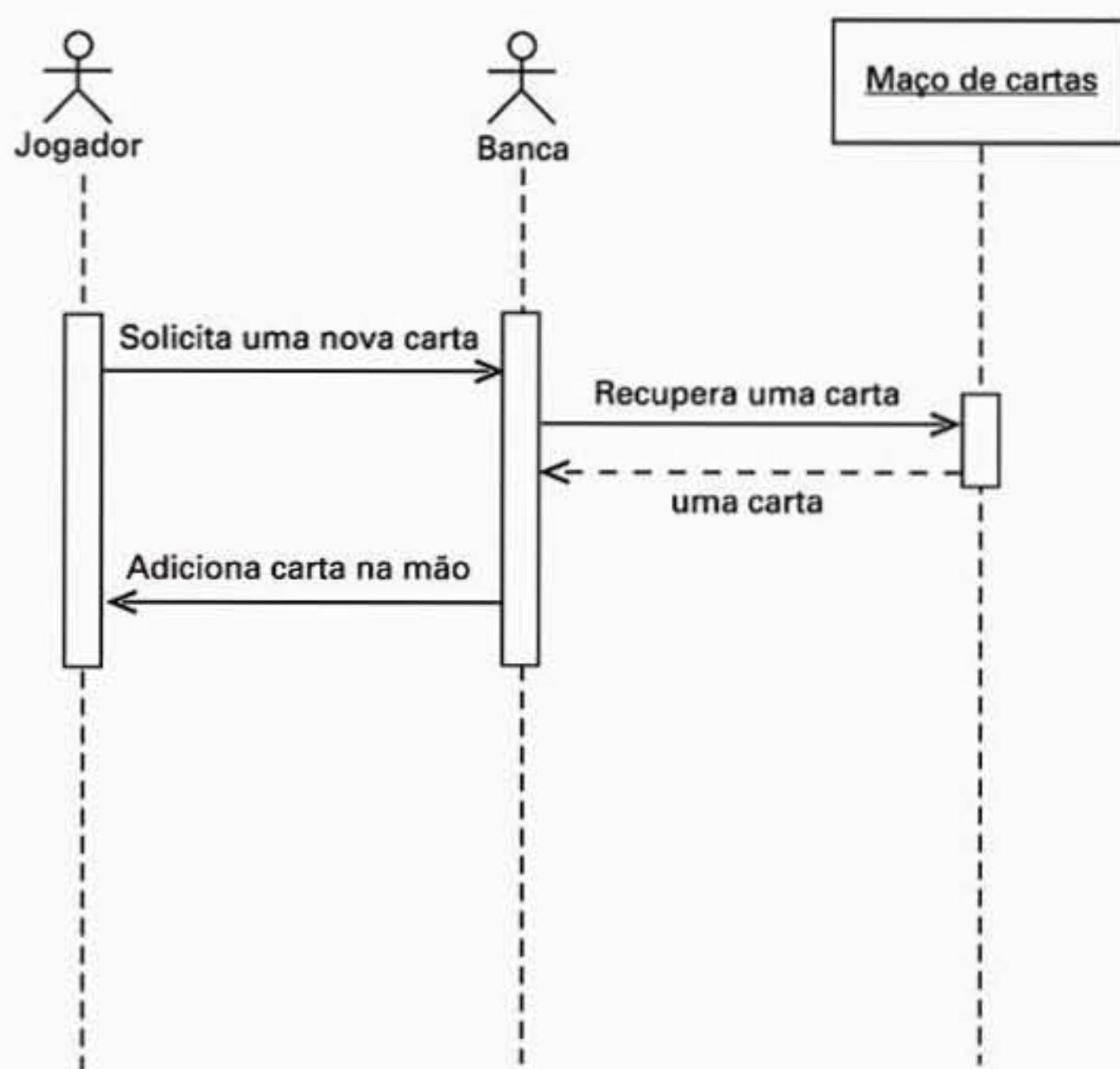
**FIGURA 15.2**

O diagrama da seqüência do caso de uso *Distribui cartas*.



**FIGURA 15.3**

O diagrama da seqüência do caso de uso Jogador recebe mais cartas.



## Modelando o domínio

Usando os casos de uso como base para o modelo de domínio, você pode isolar sete objetos do domínio diferentes: BlackjackGame, Dealer, Player, Card, Deck, DeckPile e Hand (Jogo vinte-e-um, Banca, Jogador, Carta, Baralho, Maço de cartas e Mão). A Figura 15.4 mostra o diagrama do modelo de domínio resultante.

## Projeto do jogo vinte-e-um

Aplicando projeto orientado a objetos na análise anterior, você chegará a um modelo das classes principais do projeto, suas responsabilidades e uma definição de como elas vão interagir e obter suas informações. Você pode então pegar o projeto e trabalhar na implementação.

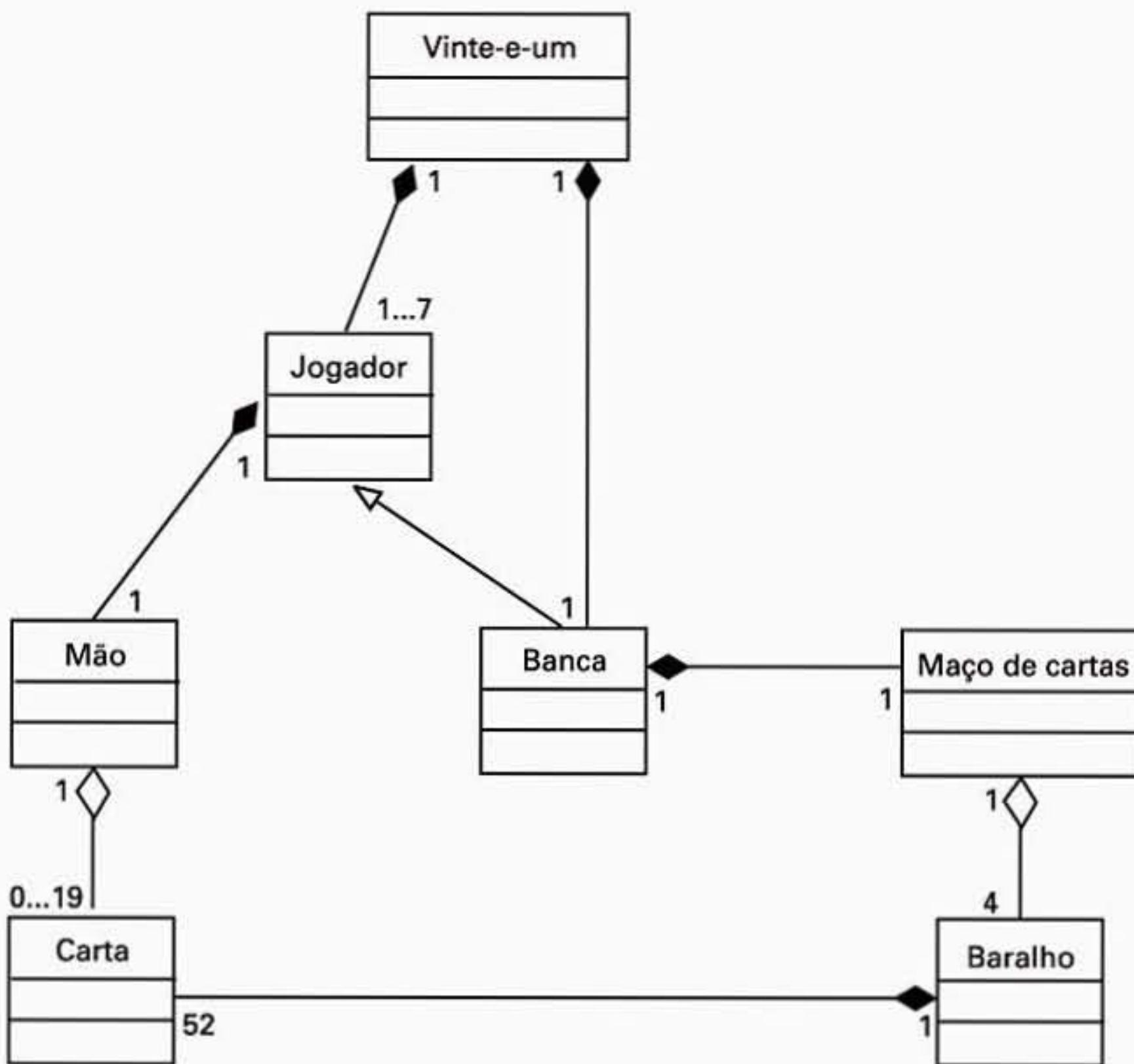
## Cartões CRC

O modelo de domínio fornece um bom ponto de partida para uma lista inicial de objetos. Com essa lista de objetos, você pode usar cartões CRC para mostrar as várias responsabilidades e colaborações dos objetos.



Antes de continuar, pode ser um bom exercício para você, tentar gerar uma lista de cartões CRC por conta própria.

**FIGURA 15.4**  
*O modelo de domínio do jogo vinte-e-um.*



As figuras 15.5 a 15.11 ilustram a possível saída de uma sessão de cartão CRC.

**FIGURA 15.5**  
*O cartão CRC do jogo vinte-e-um.*

**FIGURA 15.6**  
*O cartão CRC Baralho.*

**FIGURA 15.7**  
*O cartão CRC Carta.*

**FIGURA 15.8**  
*O cartão CRC*  
*Jogador.*

**FIGURA 15.9**  
*O cartão CRC Banca.*

**FIGURA 15.10**  
*O cartão CRC Mão.*

Mão	
segura as cartas	Carta
acrescenta cartas para si mesma	Carta
reconfigura a si mesma	
vira todas as cartas	Carta
se mostra	Carta, Console
calcula seu total	Carta, Número
detecta estouro	Carta

**FIGURA 15.11**  
*O cartão Maço de cartas.*

Maço de cartas	
aceita cartas para segurar	Carta
embaralha as cartas	Carta
distribui cartas abertas	Carta
distribui cartas fechadas	Carta
reconfigura-se	

## A UI de linha de comando

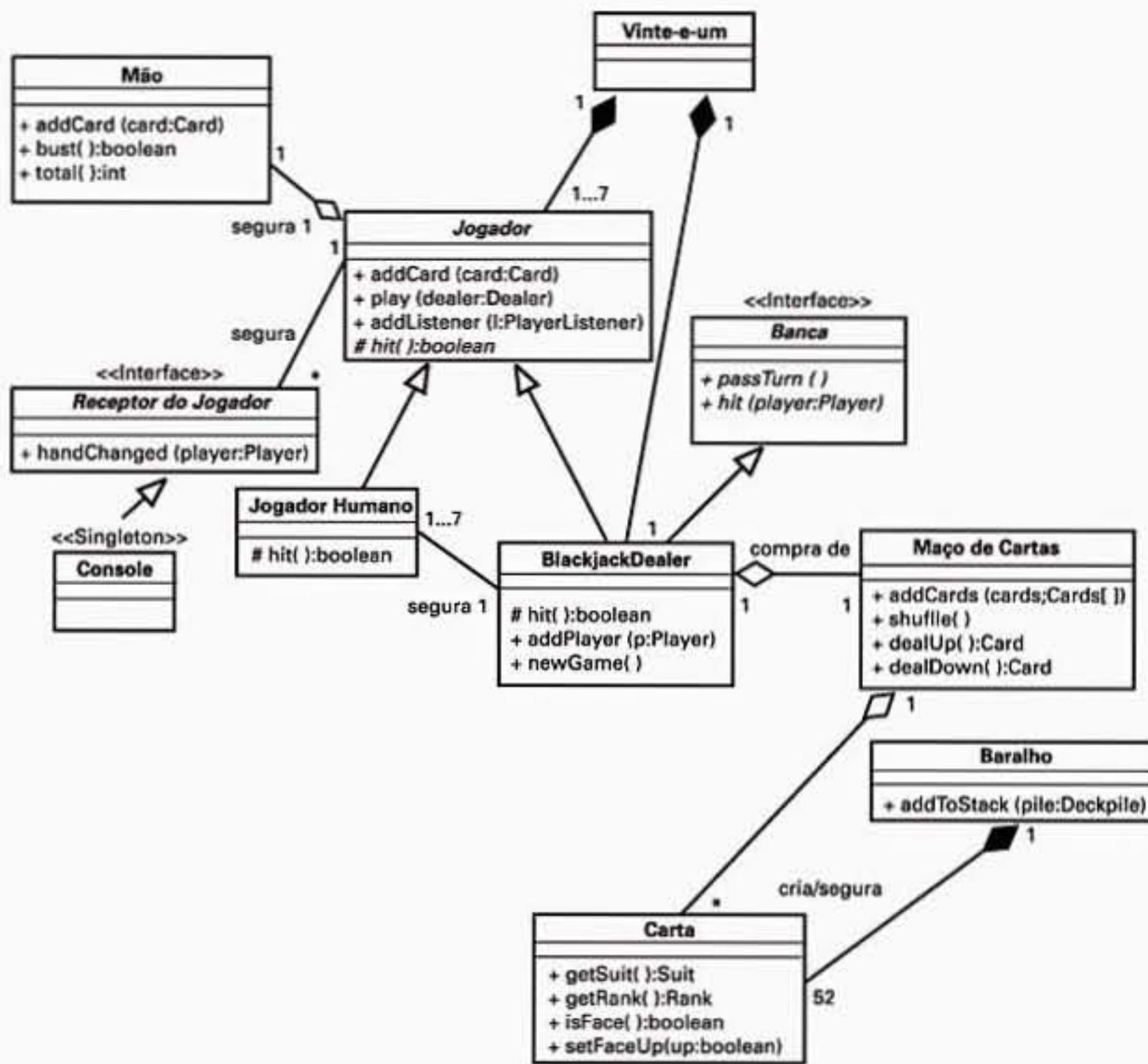
O Capítulo 13, “OO e programação de interface com o usuário”, apresentou o padrão de projeto MVC. A UI do jogo vinte-e-um utilizará o padrão de projeto MVC. Como resultado dessa decisão de projeto, você precisa adicionar um mecanismo observador no Jogador, assim como um objeto Console para apresentar os jogadores e recuperar entrada do usuário.

Como existe apenas um Console, o objeto Console é um candidato para o padrão Singleton.

## O modelo do jogo vinte-e-um

Ao todo, nove classes e duas interfaces constituem o modelo de classe Blackjack (vinte-e-um) completo. A Figura 15.12 ilustra o modelo.

**FIGURA 15.12**  
O modelo de classe Blackjack.



A próxima seção detalhará a implementação desse modelo.

## A implementação

As seções a seguir fornecem a implementação das partes principais do modelo ilustrado na Figura 15.12.

### NOTA

Todo o código-fonte está disponível para download na página da Web deste livro. Visite o endereço [www.samspublishing.com](http://www.samspublishing.com) e na guia BookStore digite o número ISBN 0672321092; em seguida, na página do livro Sams Teach Yourself Object Oriented Programming in 21 Days, dê um clique no link Downloads e depois em Source Code.

## A classe Card

A classe Card (Carta) é implementada de forma muito parecida com a classe Card apresentada no Capítulo 12, “Padrões avançados de projeto”. A classe Rank (Número) mudou um pouco. A Listagem 15.1 apresenta a nova implementação da classe Rank (Número).

**LISTAGEM 15.1** Rank.java

```
import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public final class Rank {

    public static final Rank TWO    = new Rank( 2, "2" );
    public static final Rank THREE = new Rank( 3, "3" );
    public static final Rank FOUR   = new Rank( 4, "4" );
    public static final Rank FIVE  = new Rank( 5, "5" );
    public static final Rank SIX   = new Rank( 6, "6" );
    public static final Rank SEVEN = new Rank( 7, "7" );
    public static final Rank EIGHT = new Rank( 8, "8" );
    public static final Rank NINE  = new Rank( 9, "9" );
    public static final Rank TEN   = new Rank( 10, "10" );
    public static final Rank JACK  = new Rank( 10, "J" );
    public static final Rank QUEEN = new Rank( 10, "Q" );
    public static final Rank KING  = new Rank( 10, "K" );
    public static final Rank ACE   = new Rank( 11, "A" );

    private static final Rank [] VALUES =
        { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
          EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE };

    // fornece uma lista não modificável para fazer laço
    public static final List RANKS =
        Collections.unmodifiableList( Arrays.asList( VALUES ) );

    private final int    rank;
    private final String display;

    private Rank( int rank, String display ) {
        this.rank = rank;
        this.display = display;
    }

    public int getRank() {
        return rank;
    }

    public String toString() {
        return display;
    }
}
```

Você notará que as constantes da classe Rank (Número) foram atualizadas para refletir os valores numéricos do jogo vinte-e-um. A classe também foi alterada para conter um objeto público List não modificável, em vez do array modificável apresentado no Capítulo 12. Usar um objeto List não modificável impede modificações inadvertidas da enumeração List.

## As classes Deck e Deckpile

A classe Deck (Baralho) mudou consideravelmente em relação ao apresentado no Capítulo 12. A Listagem 15.2 apresenta a nova implementação.

15

### LISTAGEM 15.2 Deck.java

```
import java.util.Iterator;
import java.util.Random;

public class Deck {

    private Card [] deck;
    private int index;

    public Deck() {
        buildCards();
    }

    public void addToStack( Deckpile stack ) {
        stack.addCards( deck );
    }

    private void buildCards() {
        deck = new Card [52];

        Iterator suits = Suit.SUITS.iterator();

        int counter = 0;
        while( suits.hasNext() ) {
            Suit suit = (Suit) suits.next();
            Iterator ranks = Rank.RANKS.iterator();
            while( ranks.hasNext() ) {
                Rank rank = (Rank)ranks.next();
                deck[counter] = new Card( suit, rank );
                counter++;
            }
        }
    }
}
```

A classe Deck (Baralho) simplesmente sabe como construir seus objetos Card (Cartas) e depois inserir-se em Deckpile (Maço de cartas). A Listagem 15.3 apresenta Deckpile (Maço de cartas).

**LISTAGEM 15.3** Deckpile.java

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

public class Deckpile {

    private ArrayList stack = new ArrayList();
    private int index;
    private Random rand = new Random();

    public void addCards( Card [] cards ) {
        for( int i = 0; i < cards.length; i ++ ) {
            stack.add( cards [i] );
        }
    }

    public void shuffle() {
        reset();
        randomize();
        randomize();
        randomize();
        randomize();
    }

    public Card dealUp() {
        Card card = deal();
        if( card != null ) {
            card.setFaceUp( true );
        }
        return card;
    }

    public Card dealDown() {
        Card card = deal();
        if( card != null ){
            card.setFaceUp( false );
        }
        return card;
    }

    public void reset() {
        index = 0;
```

**LISTAGEM 15.3 Deckpile.java (continuação)**

```
Iterator i = stack.iterator();
while( i.hasNext() ) {
    Card card = (Card) i.next();
    card.setFaceUp(false);
}
}

private Card deal() {
    if( index != stack.size() ) {
        Card card = (Card) stack.get( index );
        index++;
        return card;
    }
    return null;
}

private void randomize() {
    int num_cards = stack.size();
    for( int i = 0; i < num_cards; i ++ ) {
        int index = rand.nextInt( num_cards );
        Card card_i = (Card) stack.get( i );
        Card card_index = (Card) stack.get( index );
        stack.set( i, card_index );
        stack.set( index, card_i );
    }
}
}
```

15

A classe `Deckpile` (Maço de cartas) sabe como embaralhar seus objetos `Card` (Cartas), distribuí-los e acrescentar objetos `Card` (Cartas) para si mesmo. Ao contrário da classe `Deck` (Baralho) original, `Deckpile` (Maço de cartas) mantém uma referência para todos os objetos `Card` (Carta) que retorna. Desse modo, ele pode recuperar facilmente todos os objetos `Card` (Carta) e se reconfigurar. Embora isso não modele completamente o mundo real, simplifica muito o gerenciamento de cartas.

Entender o raciocínio existente por trás dessas alterações é importante. `Deck` (Baralho) e `Deckpile` (Maço de cartas) implementam apenas os comportamentos de que o jogo precisa. Essas classes não fornecem funcionalidade adicional ‘apenas para o caso de precisarmos algum dia’. É impossível prever o futuro e você não tem quaisquer requisitos de funcionalidade extra; portanto, você não deve acrescentá-la até saber que precisa dela.

Se você tentar implementar toda possibilidade de ‘e se’ ou realizar uma abstração prematura, nunca acabará de implementar suas classes. Se você conseguir terminá-las, são boas as chances

de que a funcionalidade ou abstração acrescentada não esteja correta. Além disso, você só terá mais trabalho a fazer, pois precisará manter uma funcionalidade que ninguém mais precisa ou usa.

Tentar programar toda hipótese ‘e se’ é um problema comum, encontrado por programadores iniciantes em OO. Você deve evitar a tentação de adicionar mais funcionalidade do que a absolutamente exigida para seus objetos! Entretanto, você deve isolar as partes do sistema que sabe que mudarão.

## As classes Player e HumanPlayer

A classe Player (Jogador) contém um objeto Hand (Mão). A Listagem 15.4 apresenta a classe Hand.

### LISTAGEM 15.4 Hand.java

---

```
import java.util.ArrayList;
import java.util.Iterator;

public class Hand {

    private ArrayList cartas = new ArrayList();
    private static final int BLACKJACK = 21;

    public void addCard( Card card ){
        cards.add( card );
    }

    public boolean bust() {
        if( total() > BLACKJACK ) {
            return true;
        }
        return false;
    }

    public void reset() {
        cards.clear();
    }

    public void turnOver(){
        Iterator i = cards.iterator();
        while( i.hasNext() ) {
            Card card = (Card)i.next();
            card.setFaceUp( true );
        }
    }
}
```

**LISTAGEM 15.4** Hand.java (*continuação*)

```
public String toString() {
    Iterator i = cards.iterator();
    String string = "";
    while( i.hasNext() ) {
        Card card = (Card)i.next();
        string = string + " " + card.toString();
    }
    return string;
}

public int total() {
    int total = 0;
    Iterator i = cards.iterator();
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        total += card.getRank().getRank();
    }
    return total;
}
}
```

A classe Hand (Mão) sabe como adicionar objetos Card (Cartas) em si mesma, reconfigurar-se, virar suas cartas, calcular seu total e representar-se como uma String. Você pode notar que a classe Hand (Mão) conta ases apenas como 11. A próxima iteração acrescentará suporte para contagem de ases como 1 ou 11.

As listagens 15.5 e 15.6 apresentam as classes Player (Jogador) e HumanPlayer (Jogador Humano), respectivamente.

**LISTAGEM 15.5** Player.java

```
import java.util.ArrayList;
import java.util.Iterator;

public abstract class Player {

    private Hand hand;
    private String name;
    private ArrayList listeners = new ArrayList();

    public Player( String name, Hand hand ) {
        this.name = name;
        this.hand = hand;
    }

    void addListener( Listener listener ) {
        listeners.add( listener );
    }

    void removeListener( Listener listener ) {
        listeners.remove( listener );
    }

    void fireEvent( String event ) {
        for( Listener listener : listeners ) {
            listener.update( event );
        }
    }
}
```

**LISTAGEM 15.5 Player.java (continuação)**

```
public void addCard( Card card ) {
    hand.addCard( card );
    notifyListeners();
}

public void play( Dealer dealer ) {
    // como antes, joga até que o jogador estoure ou pare
    while( !isBusted() && hit() ) {
        dealer.hit( this );
    }
    // mas, agora, diz à banca que o jogador terminou, caso contrário, nada
    // acontecerá quando o jogador retornar
    stopPlay( dealer );
}

public void reset() {
    hand.reset();
}

public boolean isBusted() {
    return hand.bust();
}

public void addListener( PlayerListener l ) {
    listeners.add( l );
}

public String toString() {
    return ( name + ":" + hand.toString() );
}

protected Hand getHand() {
    return hand;
}

protected void notifyListeners() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener)i.next();
        pl.handChanged( this );
    }
}
/***
 * A chamada de passTurn DEVE ser dentro de um método protegido. Dealer

```

**LISTAGEM 15.5 Player.java (continuação)**

```
* precisa sobrepor esse comportamento! Caso contrário, o laço será infinito.  
*/  
protected void stopPlay( Dealer dealer ) {  
    dealer.passTurn();  
}  
  
protected abstract boolean hit();  
}
```

15

**LISTAGEM 15.6 HumanPlayer.java**

```
public class HumanPlayer extends Player {  
  
    private final static String HIT    = "H";  
    private final static String STAND  = "S";  
    private final static String MSG    = "[H]it or [S]tay";  
    private final static String DEFAULT = "invalid";  
  
    public HumanPlayer( String name, Hand hand ) {  
        super( name, hand );  
    }  
  
    protected boolean hit() {  
        while( true ) {  
            Console.INSTANCE.printMessage( MSG );  
            String response = Console.INSTANCE.readInput( DEFAULT );  
            if( response.equalsIgnoreCase( HIT ) ) {  
                return true;  
            } else if( response.equalsIgnoreCase( STAND ) ) {  
                return false;  
            }  
            // se chegarmos aqui, faz um laço até obtermos entrada significativa  
        }  
    }  
}
```

A classe abstrata Player (Jogador) define todos os comportamentos e atributos comuns a objetos Player (Jogador) e Dealer (Banca). Esses comportamentos incluem manipular a classe Hand (Mão), controlar objetos PlayerListener e jogar uma rodada.

Player define um método abstrato: `public boolean hit()`. Durante o jogo, a classe base Player (Jogador) fará uma chamada para esse método, para determinar se deve continuar a receber cartas ou parar. Subclasses podem implementar esse método para fornecer seus próprios comporta-

mentos específicos. Por exemplo, `HumanPlayer` (Jogador Humano) pergunta ao usuário se vai continuar recebendo cartas ou não ou se vai parar. Quando o objeto `Player` (Jogador) acaba de jogar, ele informa o objeto `Dealer` (Banca), chamando o método `passTurn()` de `Dealer` (Banca). Quando o objeto `Player` (Jogador) chama esse método, `Dealer` (Banca) diz a `next Player` (Jogador) para que jogue.

## Dealer – Banca

`Dealer` (Banca) é uma interface que especifica os métodos extras que um objeto `Dealer` (Banca) vai expor. A Listagem 15.7 apresenta a interface `Dealer` (Banca).

---

### **LISTAGEM 15.7** Dealer.java

---

```
public interface Dealer {  
    public void hit( Player jogador );  
    public void passTurn();  
}
```

---

A Listagem 15.8 apresenta `BlackjackDealer`.

---

### **LISTAGEM 15.8** BlackjackDealer.java

---

```
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class BlackjackDealer extends Player implements Dealer {  
  
    private Deckpile cards;  
    private ArrayList players = new ArrayList();  
    private int player_index;  
  
    public BlackjackDealer( String name, Hand hand, Deckpile cards ) {  
        super( name, hand );  
        this.cards = cards;  
    }  
  
    public void passTurn() {  
        if( player_index != players.size() ) {  
            Player player = (Player) players.get( player_index );  
            player_index++;  
            player.play( this );  
        } else {  
    }
```

**LISTAGEM 15.8 BlackjackDealer.java (continuação)**

```
        this.play( this );
    }

public void addPlayer( Player player ) {
    players.add( player );
}

public void hit( Player player ) {
    player.addCard( cards.dealUp() );
}

// sobrepõe para que a banca mostre suas cartas antes que comece a jogar
public void play( Dealer dealer ) {
    exposeCards();
    super.play( dealer );
}

public void newGame() {
    // distribui as cartas e diz ao primeiro jogador para começar
    deal();
    passTurn();
}

public void deal() {

    cards.shuffle();
    // reconfigura cada jogador e distribui uma carta aberta para cada um e
para si mesma
    Player [] player = new Player [players.size()];
    players.toArray( player );
    for( int i = 0; i < player.length; i ++ ) {
        player [i].reset();
        player [i].addCard( cards.dealUp() );
    }
    this.addCard( cards.dealUp() );

    // distribui mais uma carta aberta para cada jogador e uma fechada para si
mesma
    for( int i = 0; i < player.length; i ++ ) {
        player [i].addCard( cards.dealUp() );
    }

    this.addCard( cards.dealDown() );
```

**LISTAGEM 15.8 BlackjackDealer.java (continuação)**

```
}

protected void stopPlay( Dealer dealer ) {
    // não faz nada aqui, na banca, simplesmente deixa o jogo parar
    // se isso não fosse sobreposto, chamaria passTurn() e
    // faria um laço infinito
}

protected boolean hit() {
    if( getHand().total() <= 16 ) {
        return true;
    }
    return false;
}

private void exposeCards() {
    getHand().turnOver();
    notifyListeners();
}
}
```

---

BlackjackDealer (Banca21) herda de Player (Jogador) porque um objeto Dealer (Banca) também é um objeto Player (Jogador). Além dos comportamentos fornecidos por um objeto Player (Jogador), a banca também contém o objeto Player (Jogador), que distribui cartas para esses objetos Player (Jogador) e diz a cada um para que jogue, quando chega sua vez. Quando um objeto Player (Jogador) chama o método `passTurn()` de Dealer (Banca), ele sabe deixar o próximo objeto Player (Jogador) jogar.

A Figura 15.13 ilustra a interação entre a banca e os jogadores.

BlackjackDealer (Banca21) sobrepõe seu método `stopPlay()` para que ele termine de jogar. A banca também implementa o método `hit()` para que ele retorne `true` quando a mão for menor que 17 e `false` quando a mão for igual ou maior que 17.

## BlackjackGame (Jogo 21)

As listagens 15.9 e 15.10 apresentam as classes `Blackjack` e `Console`, respectivamente.

**LISTAGEM 15.9 Blackjack.java**

```
public class Blackjack {

    public static void main( String [] args ) {
        Deckpile cards = new Deckpile();
```

**LISTAGEM 15.9** Blackjack.java (*continuação*)

```

for( int i = 0; i < 4; i ++ ) {
    cards.shuffle();
    Deck deck = new Deck();
    deck.addToStack( cards );
    cards.shuffle();
}

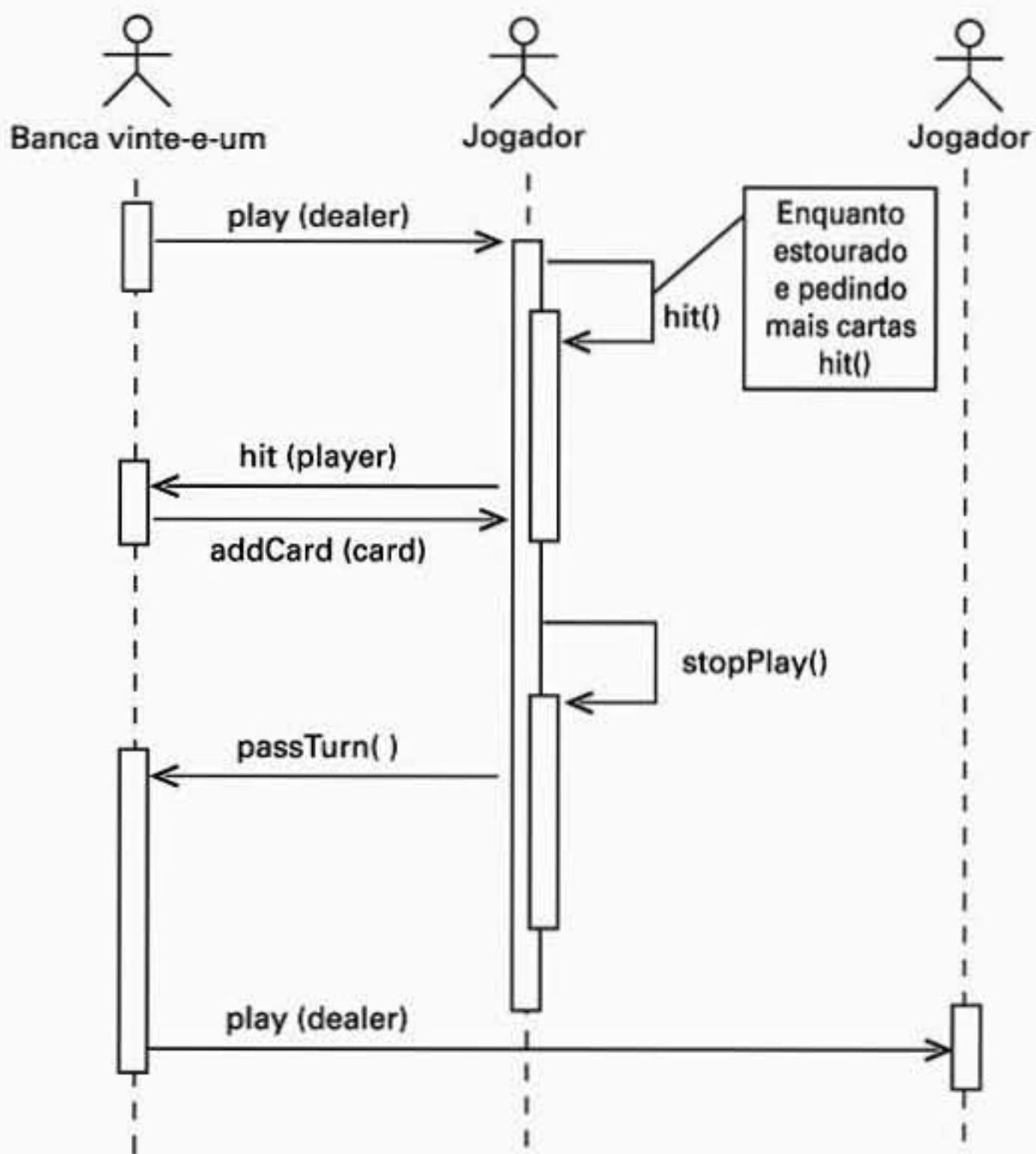
Hand dealer_hand = new Hand();
BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand,
cards );
Hand human_hand = new Hand();
Player player = new HumanPlayer( "Human", human_hand );
dealer.addListener( Console.INSTANCE );
player.addListener( Console.INSTANCE );
dealer.addPlayer( player );

dealer.newGame();
}
}

```

**FIGURA 15.13**

*A interação entre jogadores e a banca.*



**LISTAGEM 15.10** Console.java

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Console implements PlayerListener {

    // console singleton
    public final static Console INSTANCE = new Console();

    private BufferedReader in =
        new BufferedReader( new InputStreamReader( System.in ) );

    public void printMessage( String message ) {
        System.out.println( message );
    }

    public String readInput( String default_input ) {
        String response;
        try {
            return in.readLine();
        } catch (IOException ioe) {
            return default_input;
        }
    }

    public void handChanged( Player player ) {
        printMessage( player.toString() );
    }

    // private para impedir instanciação
    private Console() {}
}
```

---

A classe Blackjack constrói objetos Dealer (Banca), Hand (Mão), Deckpile (Maço de cartas), Deck (Baralho) e conecta todos eles. Uma vez conectados, ela começa o jogo dizendo ao objeto Dealer (Banca) para que inicie um novo jogo.

Console é um singleton que dá acesso à linha de comando. Ele também recebe dados dos objetos Player (Jogador) e, em seguida, os imprime na tela, sempre que eles são atualizados.

### **Uma armadilha procedural**

Se você vem de uma base procedural, poderia ser tentador implementar o método newGame() de BlackjackDealer como ilustrado na Listagem 15.11.

**LISTAGEM 15.11** Uma implementação procedural de BlackjackDealer

```
public void newGame() {  
  
    cards.shuffle();  
  
    // reconfigura cada jogador e distribui uma carta aberta para cada um e para  
    // si mesma  
    Player [] player = new Player[players.size()];  
    players.toArray( player );  
    for( int i = 0; i < player.length; i ++ ) {  
        player [i].reset();  
        player [i].addCard( cards.dealUp() );  
    }  
    this.addCard( cards.dealUp() );  
  
    // distribui mais uma carta aberta para cada jogador e uma fechada para si mesma  
    for( int i = 0; i < player.length; i ++ ) {  
        player [i].addCard( cards.dealUp() );  
    }  
    this.addCard( cards.dealDown() );  
  
    // cada jogador joga e depois a banca  
    for( int i = 0; i < player.length; i ++ ) {  
        player [i].play(this);  
    }  
    exposeCards();  
    this.play( this );  
}
```

15

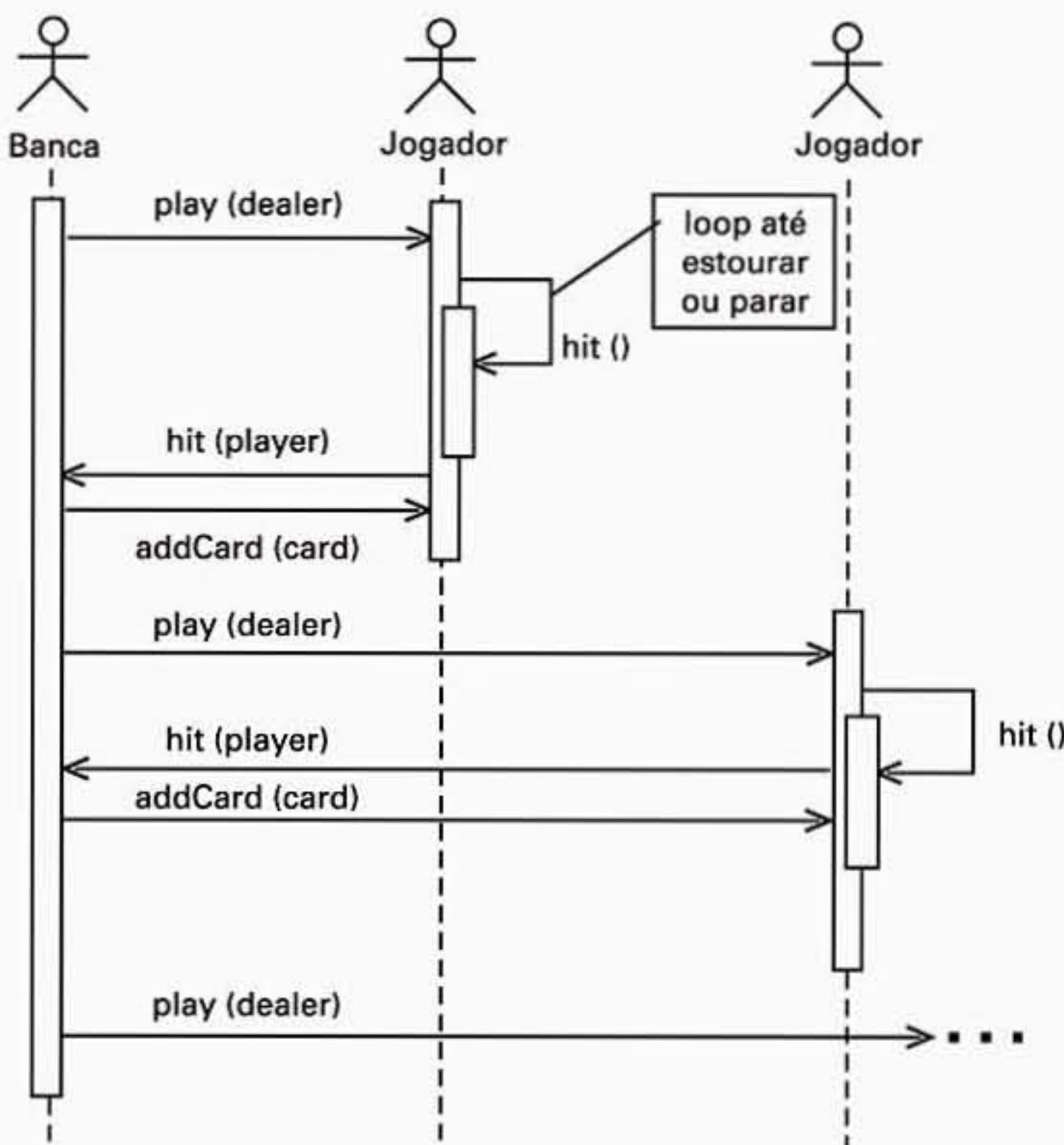
Essa implementação elimina a necessidade do método `passTurn()`; entretanto, essa é uma estratégia procedural para o laço do jogo. Em vez dos objetos `Player` (Jogador) comunicarem ao objeto `Dealer` (Banca) que terminaram de jogar, `Dealer` (Banca) simplesmente faz um laço seqüencial através dos `Players` (Jogadores).

A Figura 15.14 ilustra a interação entre `Dealer` (Banca) e seus objetos `Player` (Jogador).

Você notará que as interações da Figura 15.13 são muito mais dinâmicas do que as interações que ocorrem na Figura 15.14. A Figura 15.13 é verdadeiramente um sistema de objetos interagindo. Na Figura 15.14, `Dealer` (Banca) simplesmente espera que o objeto `Player` (Jogador) retorne e, então, chama seqüencialmente o próximo objeto `Player` (Jogador). Não há nenhuma interação além de `Dealer` (Banca) dizer estaticamente para que cada objeto `Player` (Jogador) jogue em sua vez. Embora essa estratégia funcione, ela não é muito flexível e certamente não é tão orientada a objetos quanto a estratégia apresentada na Figura 15.13.

**FIGURA 15.14**

*A interação procedural entre Players (jogadores) e Dealer (banca).*



## Testando

Conforme mostrou o Capítulo 14, “Construindo software confiável através de testes”, os testes devem ser um processo dinâmico. Um conjunto completo de testes está disponível, junto com o código-fonte, para download. Esses testes consistem em um conjunto de testes de unidade e objetos falsificados que testam completamente o jogo vinte-e-um. O estudo dos testes será deixado como um exercício importante para o leitor.

## Resumo

Hoje, você analisou, projetou e implementou um jogo vinte-e-um básico. Usando uma estratégia iterativa, você obteve os resultados que puderam ser vistos rapidamente. No decorrer da semana, você continuará a acrescentar funcionalidades nesse jogo vinte-e-um.

Você também aprendeu algumas lições novas hoje. Ao programar, você precisa evitar cair nas armadilhas procedurais, mesmo que uma estratégia procedural possa parecer a solução natural. Você também deve aprender a evitar a tentação de acrescentar mais detalhes em uma definição de classe do que precisa.

## Perguntas e respostas

15

**P Se os testes são tão importantes, por que você os pulou?**

**R** Não pulamos os testes. Todos os códigos-fonte que podem ser carregados por download estão repletos de casos de teste.

O texto pulou a discussão dos testes por restrições de espaço e eficácia. Não achamos que você teria gostado, se fosse obrigado a ler incontáveis páginas de código de teste. O estudo e o entendimento do código de teste (e, na verdade, todo o código) são deixados como exercício para o leitor.

**P Parece haver muito mais código do que o publicado no capítulo — o que aconteceu?**

**R** Há muito mais código. Simplesmente não é possível abordar todo o código eficientemente dentro do texto. Considere o texto como o ‘filme dos destaques’ do código. Você precisará dedicar uma quantidade considerável de estudo pessoal para entender o código completamente.

O objetivo destes capítulos finais é apresentar um projeto global, do qual o código é apenas um componente. A análise e o projeto são igualmente importantes. Você precisará dedicar um tempo extra dentro dessa estrutura, estudando o código, se quiser entendê-lo totalmente.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Liste dois dos padrões de projeto que você viu hoje. Onde os padrões foram usados?
2. Encontre um exemplo de polimorfismo no código-fonte.
3. Encontre um exemplo de herança no código-fonte.
4. Como a classe Deck (Baralho) encapsula suas cartas?
5. Como BlackjackDealer (Banca 21) e HumanPlayer (Jogador Humano) atuam de forma polimórfica?

## Exercícios

1. Faça o download do código-fonte da iteração de hoje. Após ter o código, compile-o, faça-o funcionar executando o jogo vinte-e-um e, depois, tente entender como ele funciona. Ter um entendimento total do código exigirá algum tempo e paciência.
2. A lição de hoje foi bastante longa. Não há outros exercícios. Examine o código-fonte e reveja a lição.

PÁGINA EM BRANCO

# SEMANA 3

DIA **16**

## Iteração 2 do jogo vinte-e-um: adicionando regras

Ontem, você realizou a análise e projeto iniciais de um jogo vinte-e-um. Hoje, você continuará esse processo e vai adicionar mais regras no jogo vinte-e-um.

Hoje você aprenderá como:

- Modelar os estados do jogo vinte-e-um
- Usar estados para remover lógica condicional

### Regras do jogo vinte-e-um

Ontem, você construiu um jogo vinte-e-um simples. O jogo que você projetou e construiu distribuía cartas e permitia a um jogador jogar até parar ou estourar. Um jogo vinte-e-um real fará um pouco mais. Em um jogo vinte-e-um real, os ases valem 1 ou 11 pontos. Os jogadores podem obter um vinte-e-um, estourar, empatar, perder ou ganhar. A banca nem mesmo jogará se todos os jogadores estourarem ou se receber uma mão que dê vinte-e-um.

Hoje, você vai adicionar a lógica necessária para suportar esses e outros recursos adicionais no jogo. Como sempre, você começará analisando os casos de uso.

## Análise das regras

Para entender completamente todas as regras do jogo vinte-e-um, você precisará rever cada um dos casos de uso de ontem e acrescentar todos os novos casos de uso exigidos. Uma vez identificados os casos de uso, você precisará atualizar o modelo de domínio.

### Análise de caso de uso das regras do jogo vinte-e-um

A adição de regras afeta muitos dos casos de uso identificados ontem. Existe também um novo caso de uso: *Banca efetua jogo*. Vamos começar com o caso de uso *Distribuir cartas*, que você descobriu ontem:

Começando com o primeiro jogador, a banca distribui uma carta aberta para cada jogador, terminando consigo mesma. A banca repete então esse processo, mas distribui sua própria carta fechada. A distribuição de cartas termina e então o jogo começa, quando a banca tiver distribuído para cada jogador, incluindo ela mesma, duas cartas.

- Distribuir cartas
  1. A banca distribui uma carta aberta para cada jogador, incluindo ela mesma.
  2. A banca distribui uma segunda carta aberta para todos os jogadores, menos para ela.
  3. A banca distribui uma carta fechada para ela mesma.
- Condições prévias
  - Novo jogo.
- Condições posteriores
  - Todos os jogadores e a banca têm uma mão com duas cartas.
  - A vez do primeiro jogador que não tenha vinte-e-um (duas cartas totalizando 21), começa.
  - O jogo continua para cada jogador que não tenha vinte-e-um.
- Alternativa: A banca tem vinte-e-um

Se a banca tiver um 21 natural, o jogo passará para o cumprimento. Os jogadores não terão sua vez.

O caso de uso *Distribuir cartas* acrescenta agora várias novas condições posteriores, assim como uma alternativa. O importante a notar é que, se a banca tiver vinte-e-um, o jogo terminará automaticamente. Do mesmo modo, qualquer jogador com vinte-e-um não joga.

Em seguida, reveja *Jogador recebe mais cartas*:

O jogador decide que não está satisfeito com sua mão. O jogador ainda não estourou, de modo que decide receber mais cartas. Se o jogador não estourar, ele pode optar por receber mais cartas novamente ou parar. Se o jogador estourar, o jogo passará para o jogador seguinte.

- Jogador recebe mais cartas
  1. O jogador decide que não está satisfeito com sua mão.
  2. O jogador solicita outra carta da banca.
  3. O jogador pode decidir receber mais cartas novamente ou parar, se o total em sua mão for menor ou igual a 21.
- Condições prévias
  - O jogador tem uma mão cujo valor total é menor ou igual a 21.
  - O jogador não tem vinte-e-um.
  - A banca não tem vinte-e-um.
- Condições posteriores
  - Uma nova carta é acrescentada na mão do jogador.
- Alternativa: O jogador estoura  
A nova carta faz a mão do jogador ser maior que 21. O jogador estoura (perde); a vez do próximo jogador/banca começa.
- Alternativa: mão do jogador  $> 21$ , mas o jogador tem um ás  
A nova carta faz a mão do jogador ser maior que 21. O jogador tem um ás. O valor do ás muda de 11 para 1, fazendo a mão do jogador ser menor ou igual a 21. O jogador pode decidir receber cartas novamente ou parar.

Digno de nota é o fato de que um jogador só pode jogar se ele ou a banca não tiver vinte-e-um. Esse caso de uso também introduz o fato de que um ás pode ter um valor igual a 1 ou 11, dependendo do que torna a mão melhor.

*Jogador pára* também recebe mais algumas condições prévias.

O jogador decide que está satisfeito com sua mão e pára.

- *Jogador pára*
  1. O jogador decide que está contente com sua mão e pára.
- Condições prévias
  - O jogador tem uma mão cujo valor é menor ou igual a 21.
  - O jogador não tem vinte-e-um.
  - A banca não tem vinte-e-um.
- Condições posteriores
  - A vez do jogador termina.

Assim como em *Jogador recebe mais cartas*, você precisará atualizar *Banca recebe mais cartas*:

- A banca deve receber mais cartas se o total de sua mão for  $< 17$ . Se a banca não estourar, após receber mais cartas, e o total de sua mão ainda for  $< 17$ , ela deverá receber mais car-

tas novamente. A banca deve parar em qualquer total  $\geq 17$ . Quando a banca estourar ou parar, o jogo termina.

- Banca recebe mais cartas

1. A banca recebe mais cartas se sua mão for menor que 17.

2. Nova carta acrescentada na mão da banca.

3. O total é menor que 17, a banca deve receber mais cartas novamente.

- Condições prévias

- A banca tem uma mão cujo total é menor que 17.

- Deve ser um jogador no estado de parada.

- Condições posteriores

- Nova carta na mão da banca.

- Jogo termina.

- Alternativa: Banca estoura

A nova carta faz a mão da banca ser maior que 21; a banca estoura.

- Alternativa: Banca pára

A nova carta faz a mão da banca ser maior ou igual a 17; a banca pára.

- Alternativa: Banca tem ás, pára

A nova carta faz a mão da banca ser maior ou igual a 21, mas inclui um ás. O valor do ás muda de 11 para 1, fazendo o total da banca ser 17; a banca pára.

- Alternativa: Banca tem ás, recebe mais cartas

A nova carta faz a mão da banca ser maior ou igual a 21, mas inclui um ás. O valor do ás muda de 11 para 1, fazendo o total da banca ser menor que 17; a banca recebe mais cartas.

Esse caso de uso acrescenta várias condições prévias e variantes. A condição prévia, “Deve ser um jogador no estado de parada”, significa que a banca só receberá mais cartas se houver um jogador para bater. Se nenhum jogador estiver parado, isso significa que todos os jogadores estouraram ou têm vinte-e-um. Além disso, as novas alternativas levam em conta o fato de que um ás pode ser contado como 1 ou como 11.

Do mesmo modo, a banca parará automaticamente, se não houver nenhum outro jogador parado.

A banca tem uma mão cujo total é  $\geq 17$  e pára.

- Banca pára

1. A mão da banca dá um total maior ou igual a 17 e pára.

- Condições prévias

- Mão da banca maior ou igual a 17.

- Deve ser um jogador no estado de parada.

- Condições posteriores
  - Jogo termina.
- Alternativa: nenhum jogador parado

Se não houver nenhum jogador parado, a banca parará automaticamente, independente da contagem da mão.

Quando o jogo tiver terminado, a banca precisará descobrir quem ganhou, quem perdeu e os empates. O caso de uso *Banca efetua jogo* trata desse uso:

Após todo o jogo ter terminado, a banca verifica cada mão e determina, para cada jogador, se ele ganhou ou perdeu, ou se o jogo deu empate.

16

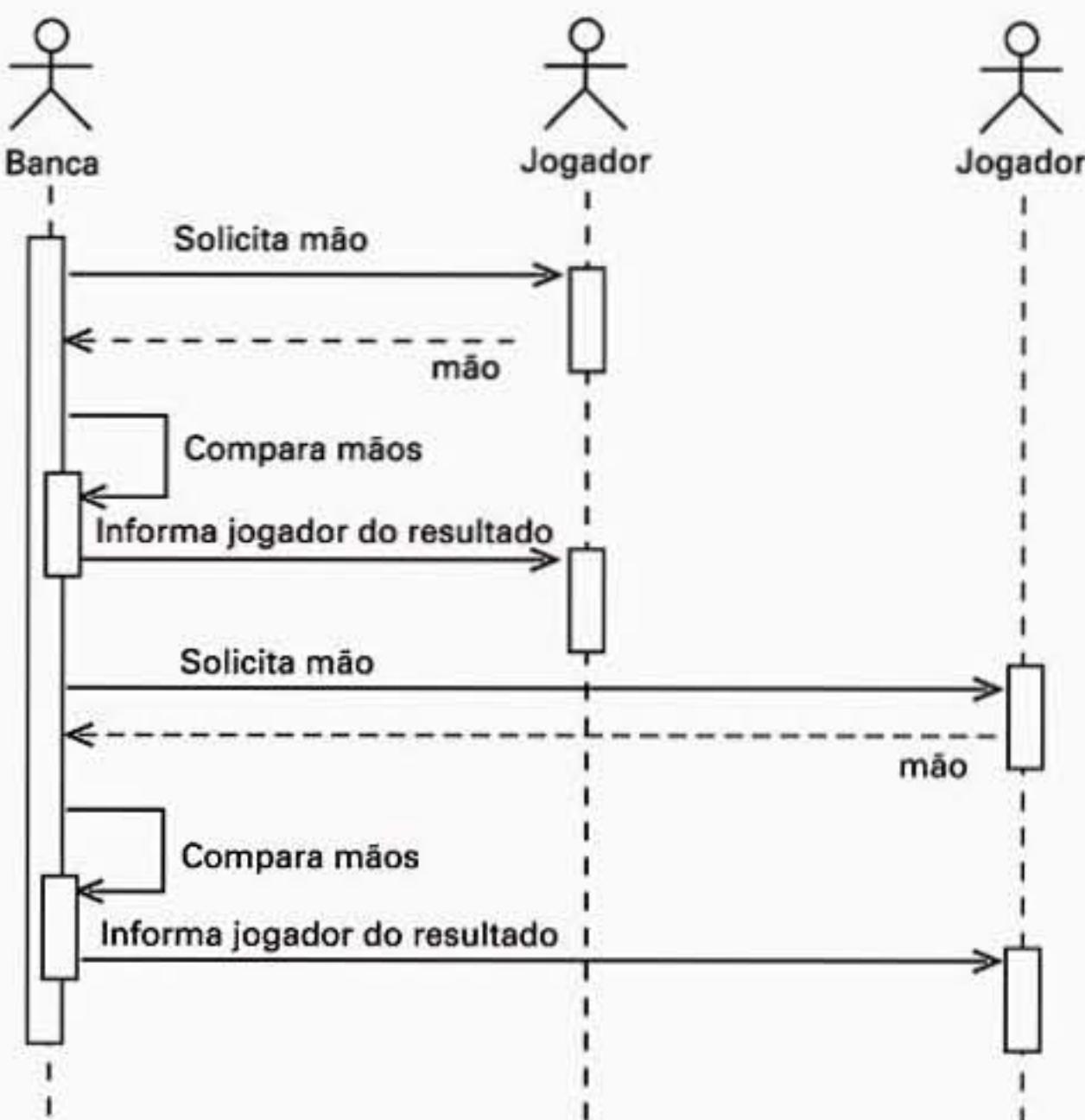
- Banca efetua jogo
  1. A banca verifica a mão do primeiro jogador e a compara com a sua própria.
  2. A mão do jogador é maior que a da banca, mas não estourou; o jogador ganha.
  3. A banca repete essa comparação para todos os jogadores.
- Condições prévias
  - Cada jogador passou por sua vez.
  - A banca passou por sua vez.
- Condições posteriores
  - Resultados finais do jogador determinado.
- Alternativa: Jogador perde
  - A banca verifica a mão do jogador e a compara com a sua própria. A mão do jogador é menor que a da banca. O jogador perde.
- Alternativa: Empate
  - A banca verifica a mão do jogador e a compara com a sua própria. A mão do jogador é igual à da banca. O jogo deu empate.
- Alternativa: Banca estoura
  - Se a banca estourou, cada jogador que estiver parado e com vinte-e-um, ganha. Todos os outros perdem.

## Modelando os casos de uso

A maior parte das alterações nos casos de uso é simples. Entretanto, pode ser útil desenhar a seqüência de eventos para *Banca efetua jogo*. A Figura 16.1 ilustra a seqüência de eventos encontrada no caso de uso *Banca efetua jogo*.

**FIGURA 16.1**

O diagrama da sequência do caso de uso Banca efetua jogo.



## Atualizando o modelo de domínio do jogo vinte-e-um

Da perspectiva do domínio, esses casos de uso novos e atualizados não o alteram.

## Projeto das regras

Neste ponto, pode ser extremamente tentador ir diretamente para a implementação. Superficialmente, parece que você pode implementar as regras através de condicionais. Na verdade, você pode implementá-las através de condicionais. A Listagem 16.1 apresenta como poderia ser uma dessas condicionais.

### **LISTAGEM 16.1** Regras condicionais dentro de BlackjackDealer (Banca 21)

```

protected void stopPlay( Dealer dealer ) {
    // o jogo terminou, identifica os ganhadores e os perdedores
    if( isBusted() ) {
        Iterator i = players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            if( !player.isBusted() ) {
                Console.INSTANCE.printMessage( player.toString() + " WINNER!!" );
            }
        }
    } else {
    }
}
  
```

**LISTAGEM 16.1** Regras condicionais dentro de BlackjackDealer (Banca 21) (cont.)

```
if( hasBlackjack() ) {
    Iterator i = players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        if( player.hasBlackjack() ) {
            Console.INSTANCE.printMessage( player.toString() +
" STANDOFF!!" );
        } else {
            Console.INSTANCE.printMessage( player.toString() +
" LOSER!!" );
        }
    }
} else { // a banca não estourou e não tem vinte-e-um
    Iterator i = players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        if( player.hasBlackjack() ) {
            Console.INSTANCE.printMessage( player.toString() +
" WINNER WITH BLACKJACK!!" );
        } else if( player.isBusted() ) {
            Console.INSTANCE.printMessage( player.toString() +
" BUSTED!!" );
        } else if( player.getHand().greaterThan( getHand() ) ){
            Console.INSTANCE.printMessage( player.toString() +
" WINNER!!" );
        } else if( player.getHand().equalTo(getHand() ) ) {
            Console.INSTANCE.printMessage( player.toString() +
" STANDOFF!!" );
        } else {
            Console.INSTANCE.printMessage( player.toString() +
" LOSER!!" );
        }
    }
}
```

16

É claro que essa condicional trata apenas do cumprimento do jogo. A banca precisará de muito mais condicionais para saber se deve ou não começar a jogar após a distribuição e se deve ou não deixar que um jogador jogue. Por exemplo, se a banca tiver uma mão com vinte-e-um, o jogo deverá terminar automaticamente. Você precisará de uma condicional para esse e todos os outros desvios do jogo.

Tal estratégia é frágil, difícil de manter, propensa a erros e simplesmente horrível. Ao tratar com condicionais, você freqüentemente verá que o acréscimo de uma nova condicional faz um antigo comportamento falhar. Também é extremamente difícil entender código repleto de condicionais. Boa sorte para aqueles que precisam manter tal bagunça condicional.

As estruturas condicionais não são particularmente orientadas a objetos. O uso incorreto de condicionais estraga as divisões corretas de responsabilidades, que são tão importantes para a POO. Ao usar condicionais, a banca degenera para uma função procedural que verifica um flag no jogador, toma uma decisão e, então, diz ao jogador o que fazer. A OO não funciona assim! As condicionais têm seus usos; entretanto, elas nunca devem extrair responsabilidades de um objeto.

Em vez disso, o conhecimento de se um jogador estourou ou se tem vinte-e-um deve estar contido dentro do próprio objeto `Player` (Jogador); então, quando um desses eventos ocorrer, o objeto `Player` (Jogador) poderá executar a ação correta e informar a `BlackjackDealer` (Banca 21), se apropriado. Em vez da banca instruir os jogadores sobre o que fazer, os jogadores devem usar seus próprios estados internos para tomar essas decisões. Tal estratégia modela mais precisamente o jogo vinte-e-um real.

O primeiro passo para se desfazer das condicionais é perceber quais eventos e estados dirigem o jogo vinte-e-um. Por todo o jogo vinte-e-um, os vários jogadores se movem através de estados. Em um ponto o jogador está esperando; depois está jogando. Após jogar, o jogador passa para um estado de espera ou de estouro. Do mesmo modo, a banca passa da distribuição para a espera por sua vez, para o jogo e, finalmente, para a parada ou estouro.

Também existem transições de estado alternativas. Após receber suas cartas, a banca ou o jogador pode mudar automaticamente para o estado de vinte-e-um, se receber uma mão que totalize vinte-e-um.

Para ter total entendimento dos vários estados, assim como dos eventos que movem o jogador de um estado para outro, ajuda modelar os diversos estados através de um diagrama de estado.

## Diagramas de estado

A UML define um rico conjunto de notações para modelar diagramas de estado. Em vez de nos perdermos nos detalhes, usaremos apenas os aspectos necessários para modelar o jogo vinte-e-um.

Para os propósitos da modelagem do jogo vinte-e-um, existem estados, transições, eventos, atividades e condições.

No jogo vinte-e-um, um estado é a condição de jogo corrente de um jogador. Tais condições incluem espera, jogo, estouro, parada e vinte-e-um, dentre outras.

As transições ocorrem quando um jogador se move entre seus estados. Por exemplo, um jogador muda do estado de jogo para o estado de estouro, quando ele estoura.

Os eventos são um tipo de estímulo que pode fazer um jogador mudar entre seus estados. Por exemplo, quando a mão do jogador causar um estouro, o jogador mudará do estado de jogo para estouro.

Atividades são as ações executadas quando se está em um estado. Por exemplo, ao jogar no estado de jogo, um jogador optará por receber mais cartas até decidir parar ou estourar.

Condições de guarda são uma expressão booleana que coloca algum tipo de restrição em uma transição. Por exemplo, um jogador mudará para o estado de parada, se decidir não receber mais cartas.

A Figura 16.2 apresenta a notação que você usará para modelar os estados do jogo vinte-e-um.

16

**FIGURA 16.2**  
*A notação do diagrama de estado.*



No modelo, as transições são simbolizadas através de setas. As transições podem ocorrer como resultado de um evento ou de uma condição (ou uma combinação dos dois). O evento e a condição devem aparecer na seta de transição para que seja óbvio o motivo da ocorrência da transição.

Você também notará que uma transição pode voltar para o estado corrente. Tal transição é conhecida como *autotransição*.

Finalmente, se o objeto executa certa ação quando está no estado, a ação é registrada como uma atividade dentro do símbolo de estado. É completamente válido para um estado não ter uma atividade.

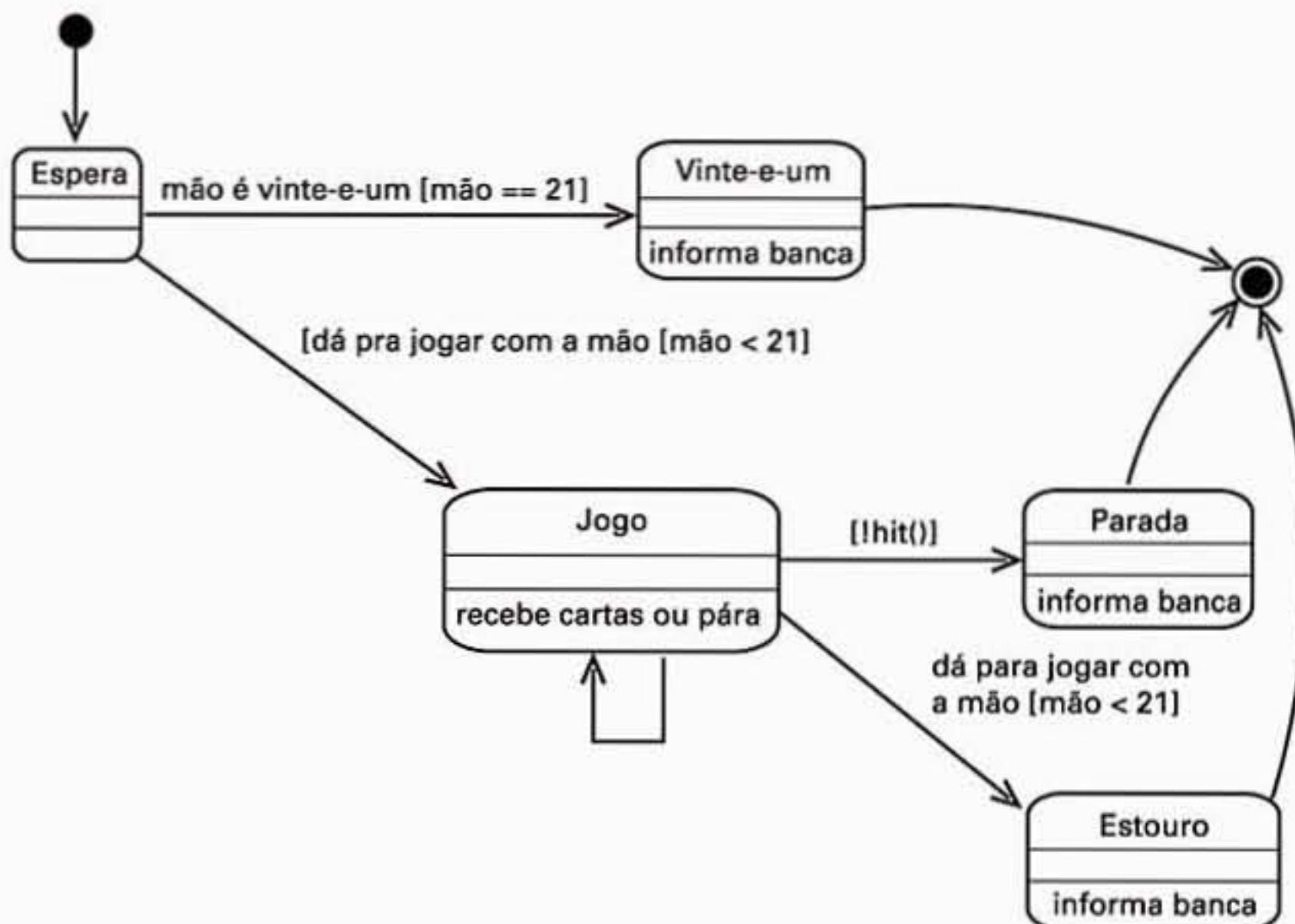
## Modelando os estados do jogador

A Figura 16.3 apresenta o diagrama de estado do jogador.

O jogador tem cinco estados principais: Espera, Vinte-e-um, Jogo, Parado e Estouro. O jogador começa no estado Espera. Após a distribuição de cartas, o jogador muda para o estado Vinte-e-um ou Jogo, dependendo da mão de cartas recebida.

Quando é sua vez de jogar, o jogador faz isso (a atividade do estado Jogo). Ao jogar, o jogador decide receber mais cartas ou parar. Se o jogador decidir parar, ele mudará para o estado Parado. Se o jogador receber mais cartas, mudará para o estado Estouro, se sua mão causar estouro, ou voltará para o estado Jogo, se der para jogar com a mão. Isso continua até que o jogador estoure ou pare.

**FIGURA 16.3**  
O diagrama de estado do jogador.



## Modelando os estados da banca

A Figura 16.4 apresenta o diagrama de estado da banca.

A banca tem seis estados principais: Distribuição, Espera, Vinte-e-um, Jogo, Parado e Estouro. A banca começa no estado Distribuição e distribui cartas para cada jogador e para si mesma. Após a distribuição, a banca muda para o estado Vinte-e-um ou Espera, dependendo da mão recebida.

Enquanto está no estado Espera, a banca espera que todos os jogadores terminem sua vez. Quando todos os jogadores tiverem terminado, a banca mudará para o estado Jogo e começará sua vez.

Assim como o jogador, a banca decide se vai receber mais cartas ou parar, no estado Jogo. Ao contrário dos jogadores, entretanto, a banca está restrita a receber mais cartas quando sua mão for menor que 17 e a parar quando sua mão for maior ou igual a 17.

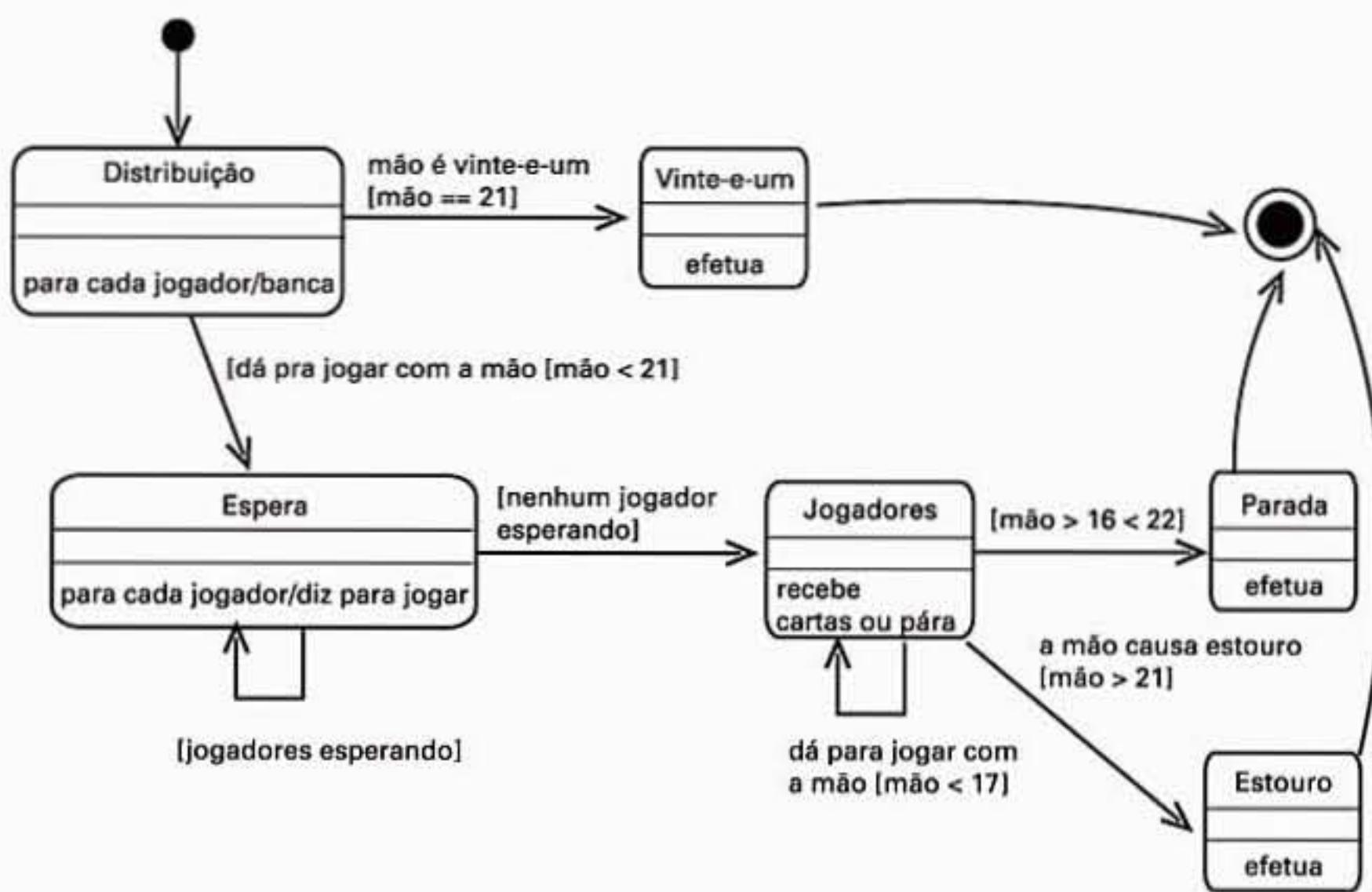
Assim como o jogador, se a banca receber mais cartas e estourar, ela mudará automaticamente para o estado Estouro. Se a banca decidir parar, mudará para o estado Parado.

Dignas de nota são as atividades nos estados Vinte-e-um, Estouro e Parado. Enquanto está nesses estados, a banca efetua o jogo e termina de jogar.

## O modelo do jogo vinte-e-um atualizado

Quando os estados do jogo estiverem modelados e completamente entendidos, você precisará decidir como vai encaixá-los no projeto. Comece colocando cada estado do jogo em sua própria classe. Em seguida, você precisará descobrir o que gera os eventos.

**FIGURA 16.4**  
O diagrama de estado da banca.



Esse uso do termo *estado* se encaixa na definição original apresentada anteriormente. Aqui, você apenas baseia um objeto em torno de cada estado que o objeto Player (Jogador) deve ter em determinado momento. Isso o libera de ter muitas variáveis internas diferentes. Em vez disso, o objeto estado encapsula perfeitamente todos esses diferentes valores dentro de um objeto, que tem estado e comportamento.

Você perceberá rapidamente que todos os eventos giram em torno do estado da mão. Assim, você provavelmente deve deixar que o objeto Hand (Mão) gere e envie esses eventos à medida que objetos Card (Carta) sejam adicionados no objeto Hand (Mão). Você também precisará estabelecer um mecanismo para que os estados recebam eventos de Hand (Mão).

Os estados em si são muito simples e têm três responsabilidades. Os estados são responsáveis por:

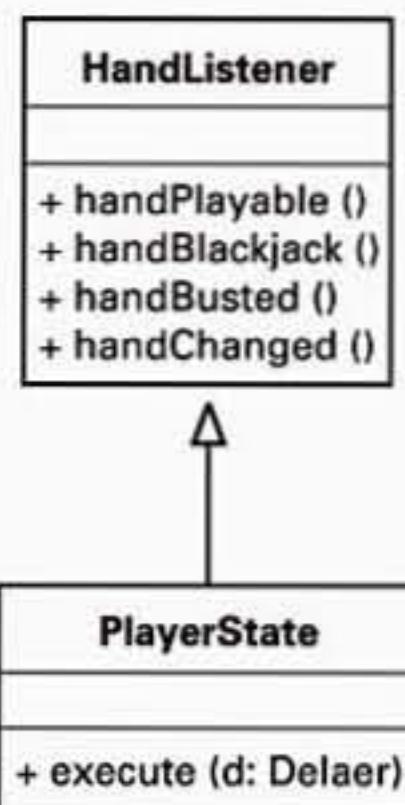
- Executar todas as atividades.
- Receber e responder aos eventos.
- Saber para qual estado ir, em resposta a um evento ou condição.

A Figura 16.5 ilustra o diagrama de classe State.

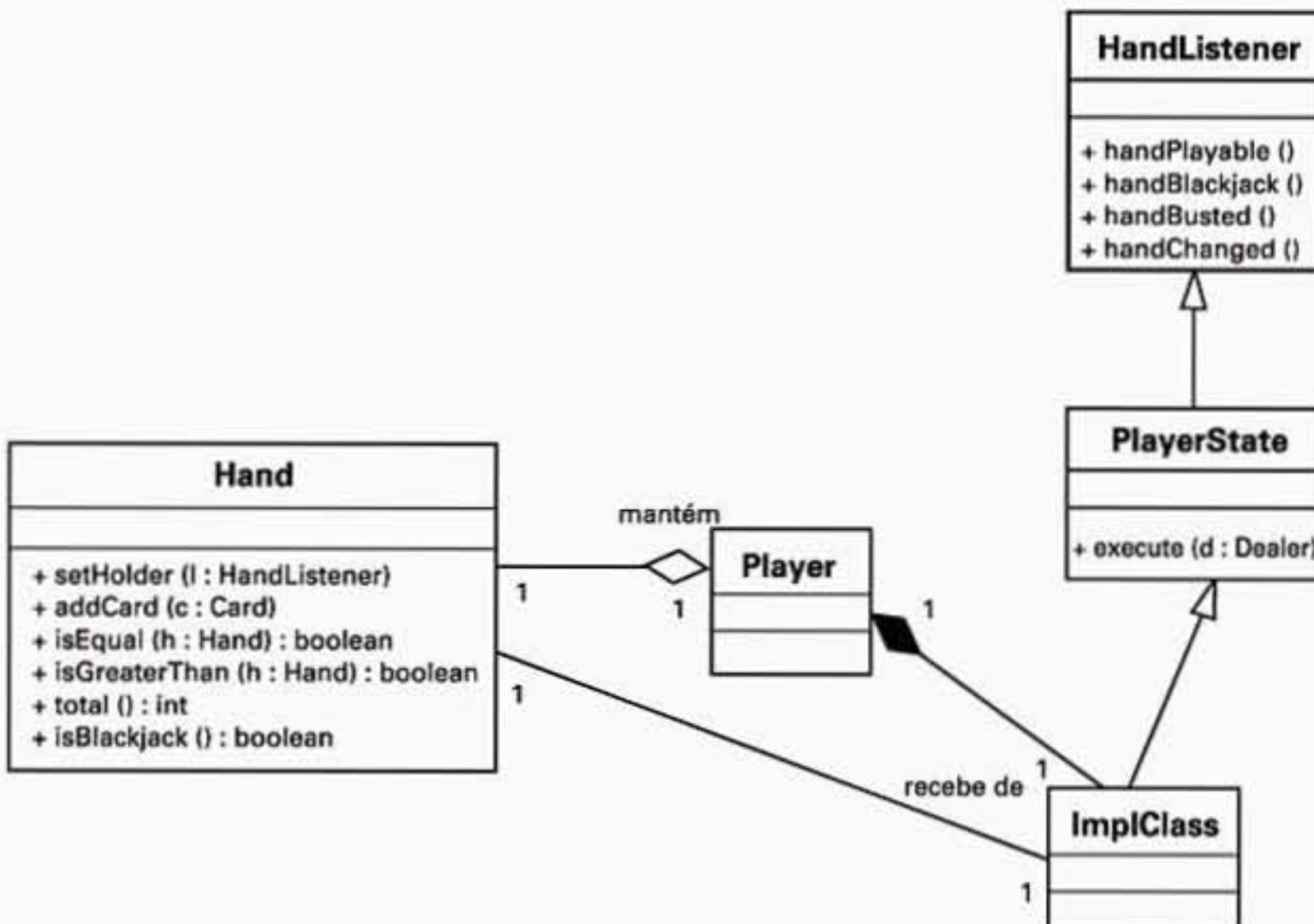
Você notará que cada evento tem um método correspondente no State (Estado). O objeto Hand (Mão) chamará um desses métodos, dependendo do estado que Hand (Mão) gostaria de reportar. O objeto State (Estado) também acrescenta um método `execute()`. Esse método é chamado quando State (Estado) deve executar suas ações.

A Figura 16.6 modela os relacionamentos entre os objetos Player (Jogador), Hand (Mão) e State (Estado).

**FIGURA 16.5**  
O diagrama de classe de State (Estado).



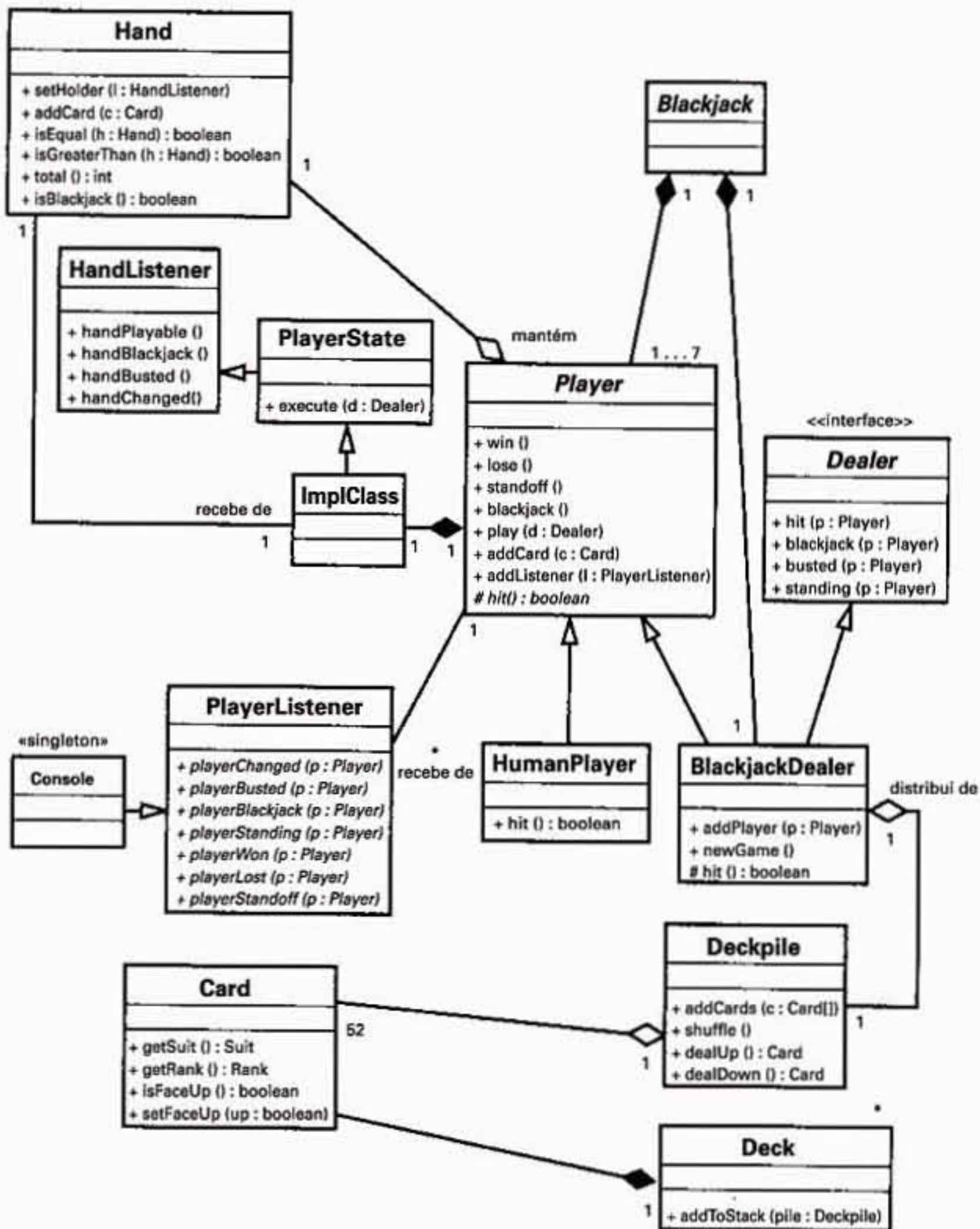
**FIGURA 16.6**  
O diagrama de classe da estrutura State (Estado).



O objeto **Player** (Jogador) mantém um objeto **State** (Estado). Quando for a vez do objeto **Player** (Jogador) fazer algo, ele simplesmente executará o método `execute()` de **State** (Estado). Então, **State** (Estado) executará todas as atividades, receberá de **Hand** (Mão) e mudará para o próximo **State** (Estado), conforme for apropriado. Uma vez feita a transição, o próximo **State** (Estado) executará todas as atividades, receberá de **Hand** (Mão) e mudará, conforme for apropriado. Esse padrão se repetirá até que o jogo termine.

A Figura 16.7 ilustra o diagrama de classes completo do jogo de cartas vinte-e-um.

**FIGURA 16.7**  
*O diagrama de classes completo de Blackjack.*



16

Embora a adição de uma estrutura State (Estado) seja uma alteração importante ocorrida durante essa iteração, outras interfaces e classes foram atualizadas para poder suportar o relato e a exibição das novas condições do jogo.

## **Implementação das regras**

Para suportar os novos recursos do jogo, são necessárias mudanças nas classes Player (Jogador), Dealer (Banca), BlackjackDealer (Banca 21) e Hand (Mão). Várias novas classes e interfaces também precisam ser acrescentadas. As seções a seguir examinarão cada mudança importante.

## Mudanças na Classe Hand

Para suportar os novos recursos do jogo, a classe Hand (Mão) deve relatar seu estado para um receptor. A Listagem 16.2 apresenta a nova interface HandListener.

**LISTAGEM 16.2 HandListener.java**

---

```
public interface HandListener {  
  
    public void handPlayable();  
  
    public void handBlackjack();  
  
    public void handBusted();  
  
    public void handChanged();  
}
```

---

A Listagem 16.3 apresenta a classe Hand (Mão) atualizada.

**LISTAGEM 16.3 Hand.java**

---

```
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class Hand {  
  
    private ArrayList cards = new ArrayList();  
    private static final int BLACKJACK = 21;  
    private HandListener holder;  
    private int number_aces;  
  
    public Hand() {  
        // configura o portador como um receptor em branco para que ele não  
        // seja nulo, caso  
        // não seja configurado externamente  
        setHolder(  
            new HandListener() {  
                public void handPlayable() {}  
                public void handBlackjack() {}  
                public void handBusted() {}  
                public void handChanged() {}  
            }  
        );  
    }  
  
    public void setHolder( HandListener holder ) {  
        this.holder = holder;  
    }  
  
    public Iterator getCards() {
```

**LISTAGEM 16.3** Hand.java (*continuação*)

```
    return cards.iterator();
}

public void addCard( Card card ) {
    cards.add( card );

    holder.handChanged();

    if( card.getRank() == Rank.ACE ) {
        number_aces++;
    }

    if( bust() ) {
        holder.handBusted();
        return;
    }
    if( blackjack() ) {
        holder.handBlackjack();
        return;
    }
    if ( cards.size() >= 2 ) {
        holder.handPlayable();
        return;
    }
}

public boolean isEqual( Hand hand ) {
    if(hand.total() == this.total() ) {
        return true;
    }
    return false;
}

public boolean isGreaterThan( Hand hand ) {
    return this.total() > hand.total();
}

public boolean blackjack() {
    if( cards.size() == 2 && total() == BLACKJACK ) {
        return true;
    }
    return false;
}

public void reset() {
```

**LISTAGEM 16.3 Hand.java (continuação)**

---

```
    cards.clear();
    number_aces = 0;
}

public void turnOver() {
    Iterator i = cards.iterator();
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        card.setFaceUp( true );
    }
}

public String toString() {
    Iterator i = cards.iterator();
    String string = "";
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        string = string + " " + card.toString();
    }
    return string;
}

public int total() {
    int total = 0;
    Iterator i = cards.iterator();
    while( i.hasNext() ) {
        Card card =(Card) i.next();
        total += card.getRank().getRank();
    }
    int temp_aces = number_aces;
    while( total > BLACKJACK && temp_aces > 0 ) {
        total = total - 10;
        temp_aces--;
    }
    return total;
}

private boolean bust() {
    if(total() > BLACKJACK ) {
        return true;
    }
    return false;
}
```

---

Agora, as alterações no método `total()` de `Hand` (Mão) tornam possível que um ás tenha o valor 1 ou 11. Do mesmo modo, as alterações no método `addCard()` permitem agora que `Hand` (Mão) informe ao seu receptor das alterações no seu conteúdo, quando elas acontecerem.

Finalmente, a adição dos métodos `isEqual()` e `isGreaterThan()` possibilita a comparação fácil e encapsulada de objetos `Hand` (Mão).

## Mudanças na Classe Player

A maior alteração na hierarquia `Player` (Jogador) gira em torno do acréscimo de `States` (Estados). A Listagem 16.4 apresenta a nova interface `PlayerState`.

### Listagem 16.4 `PlayerState.java`

```
public interface PlayerState extends HandListener {  
    public void execute( Dealer dealer );  
}
```

16

`PlayerState` herda de `HandListener` e acrescenta um método `execute()`. Uma implementação de `PlayerState` implementará `PlayerState`, responderá apropriadamente a qualquer um dos eventos `HandListener` e executará suas atividades dentro de `execute()`.

O objeto `Player` (Jogador) mantém uma referência para seu estado corrente, através da variável `current_state`. O método `play()` foi alterado para simplesmente executar o estado corrente:

```
public void play( Dealer dealer ) {  
    current_state.execute( dealer );  
}
```

Em vez de definir algum comportamento dentro do método `play()`, o objeto `Player` (Jogador) simplesmente delega para seu estado. Desse modo, você pode fornecer comportamento novo simplesmente trocando objetos de diferentes estados. A troca por diferentes estados é uma solução muito mais elegante do que a troca através de uma lista de lógica condicional.

As listagens 16.5 a 16.9 apresentam as implementações padrão de `Player` e `PlayerState`. Esses estados implementam diretamente os modelos de estado da seção anterior.

### Listagem 16.5 O estado de espera padrão

```
private class Waiting implements PlayerState {  
    public void handChanged() {  
        notifyChanged();  
    }  
    public void handPlayable() {  
        setCurrentState( getPlayingState() );  
        // transição  
    }  
}
```

**LISTAGEM 16.5** O estado de espera padrão (*continuação*)

```
public void handBlackjack() {  
    setCurrentState( getBlackjackState() );  
    notifyBlackjack();  
    // transição  
}  
public void handBusted() {  
    // impossível no estado de espera  
}  
public void execute( Dealer dealer ) {  
    // não faz nada enquanto espera  
}  
}
```

---

**LISTAGEM 16.6** O estado de estouro padrão

```
private class Busted implements PlayerState {  
    public void handChanged() {  
        // impossível no estado de estouro  
    }  
    public void handPlayable() {  
        // impossível no estado de estouro  
    }  
    public void handBlackjack() {  
        // impossível no estado de estouro  
    }  
    public void handBusted() {  
        // impossível no estado de estouro  
    }  
    public void execute( Dealer dealer ) {  
        dealer.busted( Player.this );  
        // termina  
    }  
}
```

---

**LISTAGEM 16.7** O estado vinte-e-um padrão

```
private class Blackjack implements PlayerState {  
    public void handChanged() {  
        //impossível no estado de vinte-e-um  
    }  
    public void handPlayable() {  
        //impossível no estado de vinte-e-um  
    }  
}
```

**LISTAGEM 16.7** O estado vinte-e-um padrão (*continuação*)

```
public void handBlackjack() {  
    // impossível no estado de vinte-e-um  
}  
public void handBusted() {  
    // impossível no estado de vinte-e-um  
}  
public void execute( Dealer dealer ) {  
    dealer.blackjack( Player.this );  
    // termina  
}  
}
```

16

**LISTAGEM 16.8** O estado de parada padrão

```
private class Standing implements PlayerState {  
    public void handChanged() {  
        // impossível no estado de parada  
    }  
    public void handPlayable() {  
        // impossível no estado de parada  
    }  
    public void handBlackjack() {  
        // impossível no estado de parada  
    }  
    public void handBusted() {  
        // impossível no estado de parada  
    }  
    public void execute( Dealer dealer ) {  
        dealer.standing( Player.this );  
        // termina  
    }  
}
```

**LISTAGEM 16.9** O estado de jogo padrão

```
private class Playing implements PlayerState {  
    public void handChanged() {  
        notifyChanged();  
    }  
    public void handPlayable() {  
        // pode ignorar no estado de jogo  
    }  
    public void handBlackjack() {
```

**LISTAGEM 16.9** O estado de jogo padrão (*continuação*)

```
// impossível no estado de jogo
}
public void handBusted() {
    setCurrentState( getBustedState() );
    notifyBusted();
}
public void execute( Dealer dealer ){
    if( hit() ) {
        dealer.hit( Player.this );
    } else {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    current_state.execute( dealer );
    // transição
}
}
```

---

Todos esses estados são implementados como classes internas de Player (Jogador) pois, basicamente, elas são extensões da classe Player (Jogador). Como classes internas, esses estados têm total acesso a todos os métodos e variáveis da classe Player (Jogador). As classes internas permitem que você encapsule eficientemente a lógica de estado dentro de sua própria classe, sem ter de danificar o encapsulamento da classe Player (Jogador).

Subclasses podem fornecer sua própria implementação de estado, sobrepondo os seguintes métodos em Player (Jogador):

```
protected PlayerState getBustedState() {
    return new Busted();
}
protected PlayerState getStandingState() {
    return new Standing();
}
protected PlayerState getPlayingState() {
    return new Playing();
}
protected PlayerState getWaitingState() {
    return new Waiting();
}
protected PlayerState getBlackjackState() {
    return new Blackjack();
}
protected PlayerState getInitialState() {
    return new WaitingState();
}
```

Desde que os estados usem esses métodos para recuperar os outros estados, as subclasses podem introduzir seus próprios estados personalizados. `getInitialState()` é usado pela classe base `Player` (Jogador) para configurar o estado inicial de `Player` (Jogador). Se uma subclasse começar em outro estado, ela também precisará sobrepor esse método.

Finalmente, vários métodos de notificação foram adicionados na classe `Player` (Jogador). Os estados usam esses métodos para informar o receptor das alterações. Esses métodos correspondem aos novos métodos encontrados na interface `PlayerListener`. Novos métodos foram adicionados no receptor para suportar a nova funcionalidade do jogo. A Listagem 16.10 apresenta a interface `PlayerListener` atualizada.

#### **LISTAGEM 16.10** PlayerListener.java

```
public interface PlayerListener {  
  
    public void playerChanged( Player player );  
  
    public void playerBusted( Player player );  
  
    public void playerBlackjack( Player player );  
  
    public void playerStanding( Player player );  
  
    public void playerWon( Player player );  
  
    public void playerLost( Player player );  
  
    public void playerStandoff( Player player );  
  
}
```

16

Como `Console` é um `PlayerListener`, os métodos a seguir foram acrescentados em `Console`:

```
public void playerChanged( Player player ) {  
    printMessage( player.toString() );  
}  
  
public void playerBusted( Player player ) {  
    printMessage( player.toString() + " BUSTED!!" );  
}  
  
public void playerBlackjack( Player player ) {  
    printMessage( player.toString() + " BLACKJACK!!" );  
}
```

```
public void playerStanding( Player player ) {
    printMessage( player.toString() + " STANDING " );
}

public void playerWon( Player player ) {
    printMessage( player.toString() + " WINNER!!" );
}

public void playerLost( Player player ) {
    printMessage( player.toString() + " LOSER!!" );
}

public void playerStandoff( Player player ) {
    printMessage( player.toString() + " STANDOFF " );
}
```

Essas alterações permitem que `Console` apresente todos os principais eventos do jogo.

Alguns métodos novos também foram acrescentados em `Player` (Jogador):

```
public void win() {
    notifyWin();
}

public void lose() {
    notifyLose();
}

public void standoff() {
    notifyStandoff();
}

public void blackjack() {
    notifyBlackjack();
}
```

Esses métodos permitem que `Dealer` (Banca) informe a `Player` (Jogador) se ganhou, perdeu, empatou ou tinha vinte-e-um.

## Mudanças em `Dealer` e `BlackjackDealer`

`Dealer` (Banca) e `BlackjackDealer` (Banca 21) precisam ser atualizados para se encaixar na nova estrutura de estado. A Listagem 16.11 apresenta a interface `Dealer` (Banca) atualizada.

### **LISTAGEM 16.11** Dealer.java

---

```
public interface Dealer {
    // usado pelo jogador para interagir com a banca
```

**LISTAGEM 16.11** Dealer.java (*continuação*)

```
public void hit( Player player );

// usado pelo jogador para se comunicar com a banca
public void blackjack( Player player );
public void busted( Player player );
public void standing( Player player );
}
```

Player (Jogador) usa esses novos métodos para relatar o estado para Dealer (Banca). Assim, por exemplo, quando Player (Jogador) tiver vinte-e-um, chamará o método `blackjack()` de Dealer (Banca). Então, Dealer (Banca) pode passar a vez para o próximo jogador. Esses métodos são parecidos com o método `passTurn()` anterior. Eles apenas são mais específicos.

Dealer (Banca) usa chamadas para esses métodos, para filtrar os objetos Player (Jogador) em recipientes, baseado em seu estado. Isso torna o cumprimento do jogo muito mais fácil para Dealer (Banca).

Por exemplo, aqui está uma implementação de `busted()` de `BlackjackDealer`:

```
public void busted( Player player ) {
    busted_players.add( player );
    play( this );
}
```

Os outros métodos funcionam de maneira semelhante.

`BlackjackDealer` (Banca 21) adiciona um estado `DealerDealing`. Ela também personaliza muitos dos estados padrão de Player (Jogador). As listagens 16.12 a 16.16 apresentam esses estados modificados.

**LISTAGEM 16.12** O estado de estoura da banca personalizado

```
private class DealerBusted implements PlayerState {
    public void handChanged() {
        // impossível no estado de estouro
    }
    public void handPlayable() {
        // impossível no estado de estouro
    }
    public void handBlackjack() {
        // impossível no estado de estouro
    }
    public void handBusted() {
        // impossível no estado de estouro
    }
}
```

**LISTAGEM 16.12** O estado de estoura da banca personalizado (*continuação*)

```
public void execute( Dealer dealer ) {
    Iterator i = standing_players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        player.win();
    }
    i = blackjack_players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        player.win();
    }
    i = busted_players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        player.lose();
    }
}
```

**LISTAGEM 16.13** O estado de banca com vinte-e-um personalizado

```
private class DealerBlackjack implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // impossível no estado de vinte-e-um
    }
    public void handBlackjack() {
        // impossível no estado de vinte-e-um
    }
    public void handBusted() {
        // impossível no estado de vinte-e-um
    }
    public void execute( Dealer dealer ) {
        exposeHand();
        Iterator i = players.iterator();
        while( i.hasNext() ) {
            Player player = (Player)i.next();
            if( player.getHand().blackjack() ) {
                player.standoff();
            } else {
                player.lose();
            }
        }
    }
}
```

**LISTAGEM 16.13** O estado de banca com vinte-e-um personalizado (*continuação*)

```
    }
}
}
```

**LISTAGEM 16.14** O estado de banca parada personalizado

```
private class DealerStanding implements PlayerState {
    public void handChanged() {
        // impossível no estado de parada
    }
    public void handPlayable() {
        // impossível no estado de parada
    }
    public void handBlackjack() {
        // impossível no estado de parada
    }
    public void handBusted() {
        // impossível no estado de parada
    }
    public void execute( Dealer dealer ) {
        Iterator i = standing_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            if( player.getHand().isEqual( getHand() ) ) {
                player.standoff();
            } else if( player.getHand().isGreaterThan( getHand() ) ) {
                player.win();
            } else {
                player.lose();
            }
        }
        i = blackjack_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.win();
        }
        i = busted_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.lose();
        }
    }
}
```

**LISTAGEM 16.15** O estado da banca esperando personalizado

```
private class DealerWaiting implements PlayerState {  
    public void handChanged() {  
        // impossível no estado de espera  
    }  
    public void handPlayable() {  
        // impossível no estado de espera  
    }  
    public void handBlackjack() {  
        // impossível no estado de espera  
    }  
    public void handBusted() {  
        // impossível no estado de espera  
    }  
    public void execute( Dealer dealer ) {  
        if( !waiting_players.isEmpty() ) {  
            Player player = (Player) waiting_players.get( 0 );  
            waiting_players.remove( player );  
            player.play( dealer );  
        } else {  
            setCurrentState( getPlayingState() );  
            exposeHand();  
            getCurrentState().execute( dealer );  
            // faz a transição e executa  
        }  
    }  
}
```

**LISTAGEM 16.16** O estado da banca distribuindo personalizado

```
private class DealerDealing implements PlayerState {  
    public void handChanged() {  
        notifyChanged();  
    }  
    public void handPlayable() {  
        setCurrentState( getWaitingState() );  
        // transição  
    }  
    public void handBlackjack() {  
        setCurrentState( getBlackjackState() );  
        notifyBlackjack();  
        // transição  
    }  
    public void handBusted() {  
        // impossível no estado de distribuição  
    }  
}
```

**LISTAGEM 16.16** O estado da banca distribuindo personalizado (*continuação*)

```
    }
    public void execute( Dealer dealer ) {
        deal();
        getCurrentState().execute( dealer );
        // faz a transição e executa
    }
}
```

Você pode notar que `BlackjackDealer` (Banca 21) não define seu próprio estado de jogo. Em vez disso, ele usa o estado de jogo padrão de `Player` (Jogador); entretanto, para usar seus estados personalizados, `BlackjackDealer` (Banca 21) precisa sobrepor os receptores de estado encontrados na classe base `Player` (Jogador):

```
protected PlayerState getBlackjackState() {
    return new DealerBlackjack();
}
protected PlayerState getBustedState() {
    return new DealerBusted();
}
protected PlayerState getStandingState() {
    return new DealerStanding();
}
protected PlayerState getWaitingState() {
    return new DealerWaiting();
}
```

16

## Teste

Assim como no código do Capítulo 15, “Aprendendo a combinar teoria e processo”, uma bateria completa de testes está disponível para download no endereço [www.samspublishing.com](http://www.samspublishing.com), junto com o código-fonte deste capítulo. Esses testes consistem em um conjunto de testes de unidade e objetos falsificados, que testam completamente o sistema do jogo vinte-e-um.

O teste é uma parte importante do processo de desenvolvimento; entretanto, o estudo do código de teste é deixado como exercício para o leitor.

## Resumo

Hoje, você concluiu a segunda iteração do jogo vinte-e-um. Fazendo esse exercício, você viu pela primeira vez como poderia usar o processo iterativo para conseguir paulatinamente uma solução completa.

Cada iteração anterior atua como base ou fundamento para a seguinte. Em vez de começar uma nova análise ou projeto, hoje você começou construindo os casos de uso e projeto descobertos ontem.

Amanhã, você complementará melhor essa base, quando adicionar recursos de aposta no programa do jogo vinte-e-um.

## Perguntas e respostas

**P Se os estados são tão importantes, por que você quis esperar até essa iteração para incluí-los?**

**R** A iteração inicial era simples. A iteração inicial usava um jogo vinte-e-um básico que não detectava mãos de vinte-e-um natural, ganhadores ou perdedores (embora detectasse estouros). Não havia motivos para atacar o problema com uma solução complexa. Os requisitos dessa iteração justificam uma solução mais complicada, pois ela acrescenta detecção de vinte-e-um, assim como cumprimento do jogo.

**P Você poderia ter implementado os estados fora de Player (Jogador) e BlackjackDealer (Banca 21) ou eles devem ser classes internas?**

**R** Você pode implementar os estados fora da classe. Mas se você os definir fora de Player (Jogador), talvez precise adicionar alguns métodos novos para que os estados possam recuperar todos os dados de que precisam.

Gostaríamos de desaconselhar tal estratégia, por três motivos:

Primeiro, mover a definição de estado para fora da classe não resolve muito. Na verdade, isso causa trabalho extra, devido aos métodos que você precisará adicionar.

Segundo, a adição de métodos extras para que os estados possam recuperar dados estraga o encapsulamento.

Finalmente, mover os estados para fora da classe Player (Jogador) não modela o relacionamento estado/jogador muito bem. Basicamente, os estados são uma extensão de Player (Jogador). Desse modo, os estados atuam como os cérebros de Player (Jogador). É melhor que os cérebros fiquem dentro do corpo.

É importante notar que a implementação de estados como classes internas funciona perfeitamente na linguagem Java. Outras linguagens podem exigir uma estratégia ligeiramente diferente.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Quando as condicionais são perigosas?
2. Liste duas maneiras de remover condicionais.
3. A versão de Hand (Mão) apresentada hoje é melhor encapsulada do que a de ontem. Como a nova versão de Hand (Mão) se encapsula?
4. Que padrão Hand (Mão) e HandListener implementam?
5. Procure na Web mais informações sobre o padrão State.

## Exercícios

16

1. Faça download do código-fonte da iteração de hoje. Quando você tiver o código, compile-o, faça-o funcionar executando o jogo vinte-e-um e, depois, tente entender como ele funciona. Ter um entendimento total do código exigirá algum tempo e paciência.
2. O código a seguir aparece na definição da classe Player (Jogador):

```
protected void notifyChanged() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerChanged( this );
    }
}

protected void notifyBusted() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerBusted( this );
    }
}

protected void notifyBlackjack() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerBlackjack( this );
    }
}

protected void notifyStanding() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
```

```
    PlayerListener pl = (PlayerListener) i.next();
    pl.playerStanding( this );
}
}

protected void notifyStandoff(){
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerStandoff( this );
    }
}

protected void notifyWin() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerWon( this );
    }
}

protected void notifyLose() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerLost( this );
    }
}
```

Os métodos funcionam. Funcionalmente, não há nada de errado com eles; entretanto, cada método executa exatamente a mesma ação, até o ponto em que uma chamada é feita em `PlayerListener`.

Como você poderia usar objetos de modo que precisasse escrever apenas um método de notificação? Projete e implemente uma solução.

# SEMANA 3

DIA **17**

## Iteração 3 do jogo vinte-e-um: adicionando apostas

No capítulo de ontem, você viu como pegar uma implementação bastante primitiva do jogo vinte-e-um e fazer sua iteração para obter um jogo mais amadurecido. Hoje, você vai complementar o mesmo jogo vinte-e-um, adicionando suporte para apostas simples.

A lição de hoje dará a você mais experiência com o processo iterativo, assim como com AOO e POO (Projeto Orientado a Objeto). No final da lição de hoje, você também deverá começar a se sentir mais à vontade com a arquitetura baseada em estado apresentada ontem. Na verdade, um dos exercícios de hoje pedirá para que você adicione um novo estado no sistema.

Hoje você aprenderá:

- Como estender a arquitetura de estado do jogo vinte-e-um para adicionar funcionalidade
- Sobre as vantagens que uma verdadeira estratégia OO pode trazer para um sistema, trabalhando em um sistema baseado em OO

### Aposta no jogo vinte-e-um

A iteração de ontem produziu um jogo vinte-e-um bastante completo. Quase todos os recursos não relacionados com apostas agora fazem parte do jogo. Hoje, você vai acrescentar alguns dos recursos de apostas ausentes.

Assim como nas outras lições desta semana, seguiremos o processo delineado no Capítulo 9, “Introdução à AOO (Análise Orientada a Objetos)”. Vamos começar explorando os casos de uso de aposta.

## Análise da aposta

Para entender completamente a aposta, você precisará finalizar os casos de uso *Fazer apostas* e *Dobrar*, identificados durante a análise inicial do jogo vinte-e-um. Você também precisará rever os outros casos de uso para ter certeza de que eles não precisam de atualização. Uma vez que tiver terminado os casos de uso, você precisará atualizar o modelo de domínio.

### Análise do caso de uso de aposta no jogo vinte-e-um

Vamos começar com o novo caso de uso *Jogador faz aposta*:

Os jogadores começam o jogo com US\$1000 no pote. Antes que quaisquer cartas sejam distribuídas, cada jogador deve fazer uma aposta. Começando com o primeiro jogador, cada um aposta um valor de US\$10, US\$50 ou US\$100.

- Jogador faz aposta
  1. Jogador faz uma aposta de US\$10, US\$50 ou US\$100.
  2. Passa para o próximo jogador e repete até que todos os jogadores tenham feito uma aposta.
- Condições prévias
  - Novo jogo.
- Condições posteriores
  - Jogador fez aposta.

O jogo vinte-e-um real, como todo jogo, tem suas próprias regras sobre aposta. Essas regras incluirão uma aposta mínima, uma aposta máxima e o incremento da aposta. Neste jogo vinte-e-um, um jogador pode apostar US\$10, US\$50 ou US\$100. Por simplicidade, este jogo oferecerá ao jogador uma linha de crédito ilimitada. Cada jogador começará com US\$1000. Quando o jogador esvaziar seu pote, seu saldo se tornará negativo; entretanto, o jogador pode jogar, desde que queira.

O outro novo caso de uso de aposta é *Jogador dobra*:

O jogador decide que não está satisfeito com sua mão inicial. Em vez de simplesmente receber mais cartas, o jogador decide dobrar. Isso duplica a aposta do jogador e acrescenta uma carta na mão. A vez do jogador termina e o jogo passa para o jogador/banca seguinte.

- Jogador dobra
  1. O jogador decide que não está satisfeito com sua mão inicial.

2. O jogador quer dobrar.
  3. A aposta do jogador é duplicada.
  4. A banca acrescenta outra carta em sua mão.
- Condições prévias
    - Essa é a mão inicial do jogador e ainda não recebeu mais cartas nem parou.
    - O jogador não tem vinte-e-um.
    - A banca não tem vinte-e-um.
  - Condições posteriores
    - A mão do jogador tem três cartas.
    - A vez do jogador termina.
  - Alternativa: Jogador estoura

A nova carta faz a mão do jogador estourar (perde).
  - Alternativa: a mão do jogador é maior que 21, mas o jogador tem um ás.

A nova carta faz a mão do jogador ser maior que 21. O jogador tem um ás. O valor do ás muda de 11 para 1, fazendo com que a mão do jogador seja menor ou igual a 21.

Os únicos casos de uso previamente existentes, afetados pela adição da aposta, são o caso de uso *Distribuir cartas* e o caso de uso *Banca efetua jogo*. Os outros casos de uso permanecem inalterados:

Começando com o primeiro jogador, a banca distribui uma carta aberta para cada jogador, terminando consigo mesma. Então, a banca repete esse processo, mas distribui sua própria carta fechada. A distribuição termina e o jogo começa quando a banca tiver distribuído para cada jogador, incluindo ela mesma, duas cartas.

- Distribuir cartas
  1. A banca distribui uma carta aberta para cada jogador, incluindo ela mesma.
  2. A banca distribui uma segunda carta aberta para todos os jogadores, menos para ela.
  3. A banca distribui uma carta fechada para si mesma.
- Condições prévias
  - Todos os jogadores fizeram suas apostas.
- Condições posteriores
  - Todos os jogadores e a banca têm uma mão com duas cartas.
- Alternativa: a banca tem vinte-e-um

Se a banca tiver um 21 natural, o jogo passa para o cumprimento. Os jogadores não terão sua vez.

Agora, a distribuição não começará até que cada jogador tenha feito uma aposta. Vamos ver como a aposta muda o cumprimento do jogo:

Após todo o jogo ser feito, a banca verifica cada mão e determina, para cada jogador, se ele ganhou ou perdeu, ou se o jogo deu empate.

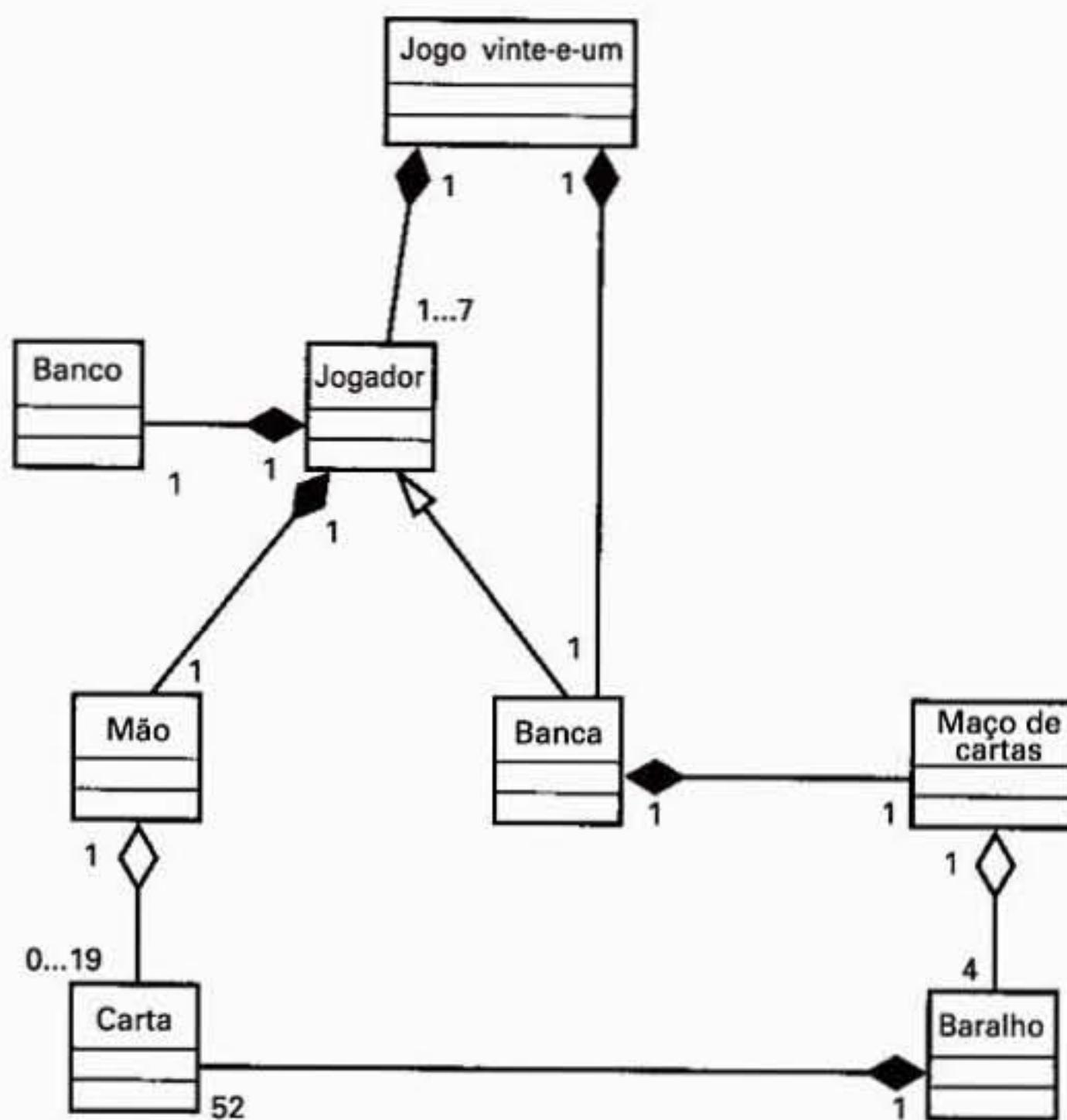
- A banca efetua o jogo
  1. A banca verifica a mão do primeiro jogador e a compara com a sua própria.
  2. A mão do jogador é maior que a da banca, mas não estourou; o jogador ganha.
  3. O valor da aposta é adicionado ao pote do jogador.
  4. A banca repete essa comparação para todos os jogadores.
- Condições prévias
  - Cada jogador teve sua vez.
  - A banca teve sua vez.
- Condições posteriores
  - Resultado final do jogador foi determinado.
- Alternativa: o jogador perde  
A banca verifica a mão do jogador e a compara com a sua própria. A mão do jogador é menor que a da banca. O jogador perde. A aposta é retirada do pote do jogador.
- Alternativa: empate  
A banca verifica a mão do jogador e a compara com a sua própria. A mão do jogador é igual à da banca. O jogo deu empate. Nada é acrescentado ou subtraído do pote do jogador.
- Alternativa: a banca estoura  
Se a banca estourou, todo jogador que estiver parado e com vinte-e-um ganha. Todos os outros perdem. Os ganhadores recebem o valor apostado.
- Alternativa: os jogadores ganham com vinte-e-um  
Se o jogador tiver vinte-e-um e a banca não, o jogador ganhará e receberá na proporção de 3:2 (por exemplo, se o jogador apostasse US\$100, receberia US\$150).

Isso conclui as alterações nos casos de uso. Todos esses casos de uso são relativamente simples. Os diagramas de interação provavelmente seriam complicados. Vamos ver como esses casos de uso atualizados mudaram o modelo de domínio.

## Atualizando o modelo de domínio do jogo vinte-e-um

A análise da aposta exige que você atualize o modelo de domínio, mas apenas ligeiramente. Você precisará adicionar mais um objeto de domínio: Bank (Banco). Todo objeto Player (Jogador) no jogo tem seu próprio objeto Bank (Banco) pessoal. A Figura 17.1 ilustra o modelo de domínio atualizado.

**FIGURA 17.1**  
O modelo de domínio  
do jogo vinte-e-um.



17

## Projeto da aposta

Você deve começar o projeto projetando a nova classe Bank (Banco). Quando Bank (Banco) estiver pronta, você precisará descobrir como apostar no jogo. Para os propósitos da lição de hoje, o caso de uso *Jogador dobra* é deixado como exercício no final da lição.

### Projetando a classe Bank

Ao começar a projetar Bank (Banco), você deve primeiro identificar suas responsabilidades. A Figura 17.2 ilustra o cartão CRC resultante para a classe Bank (Banco).

A boa OO exige a divisão correta das responsabilidades. Desse modo, Bank (Banco) é responsável por controlar todas as atividades de aposta. Bank (Banco) mantém todos os detalhes da aposta internamente e dá acesso à aposta e ao saldo através de uma interface bem definida. A Figura 17.3 ilustra o diagrama da classe Bank (Banco), assim como o relacionamento de Bank (Banco) com Player (Jogador).

### O projeto da aposta

Como se vê, a aposta deve se encaixar bem na arquitetura de estado que vimos ontem. O jogador e a banca precisarão de mais um estado para suportar aposta básica. A banca precisará de um estado *CollectingBets* (Coletar Apostas) e o jogador precisará de um estado *Aposta*. As figuras 17.4 e 17.5 ilustram os novos diagramas de estado para a banca e para o jogador, respectivamente.

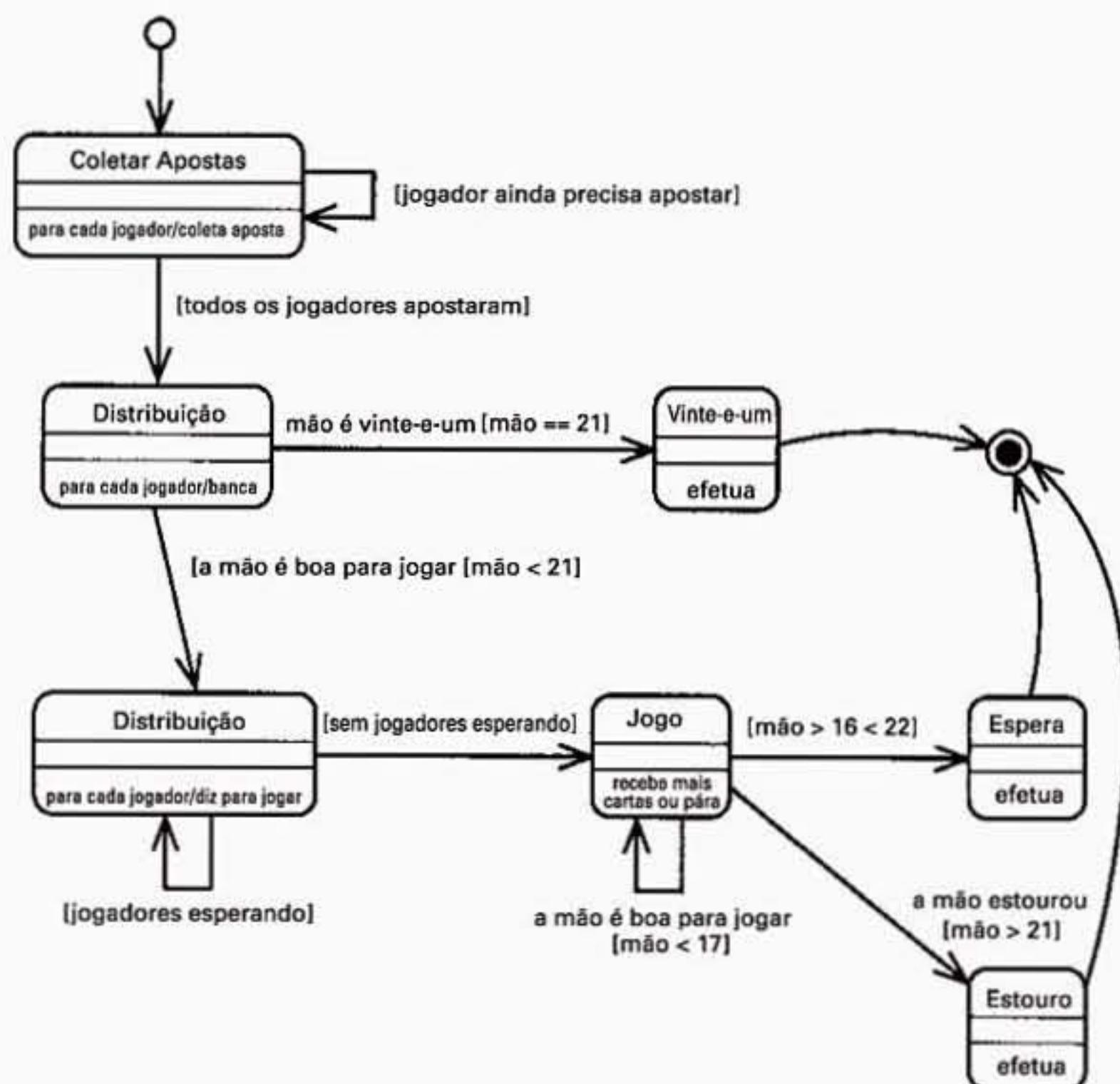
**FIGURA 17.2**  
O cartão CRC de Bank (Banco).

<b>Banco</b>	
contém o total de \$ por jogador	
faz uma aposta de US\$100	
faz uma aposta de US\$50	
faz uma aposta de US\$10	
paga o ganhador	
paga vinte-e-um	
determina a perda	
determina o empate	
representa-se como uma string	<b>String</b>

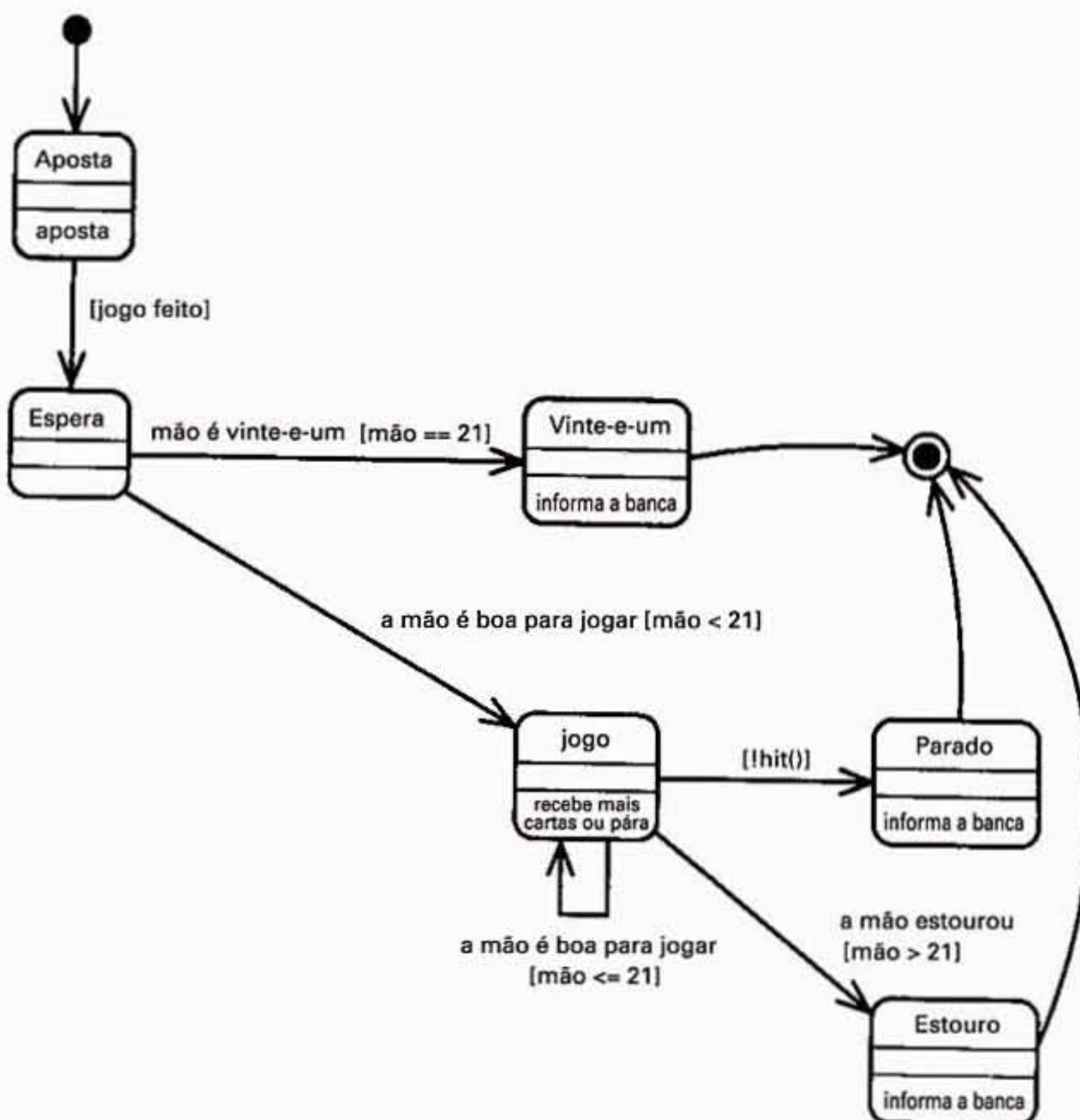
**FIGURA 17.3**  
O diagrama da classe Bank (Banco).

<b>Banco</b>
+ place100Bet(); + place50Bet(); + place10Bet(); + win(); + lose(); + blackjack(); + standoff();

**FIGURA 17.4**  
O diagrama de estado de Dealer (Banca).



**FIGURA 17.5**  
O diagrama de estado de Players (Jogadores).



17

Conforme você pode ver, agora o jogador começa no estado *Aposta*, enquanto a banca começa no estado *CollectingBets* (Coletar Apostas). Quando todas as apostas forem coletadas, a banca passará para o estado *Distribuição*, como antes. Quando terminarem as apostas, os jogadores passarão para o estado *Espera*.

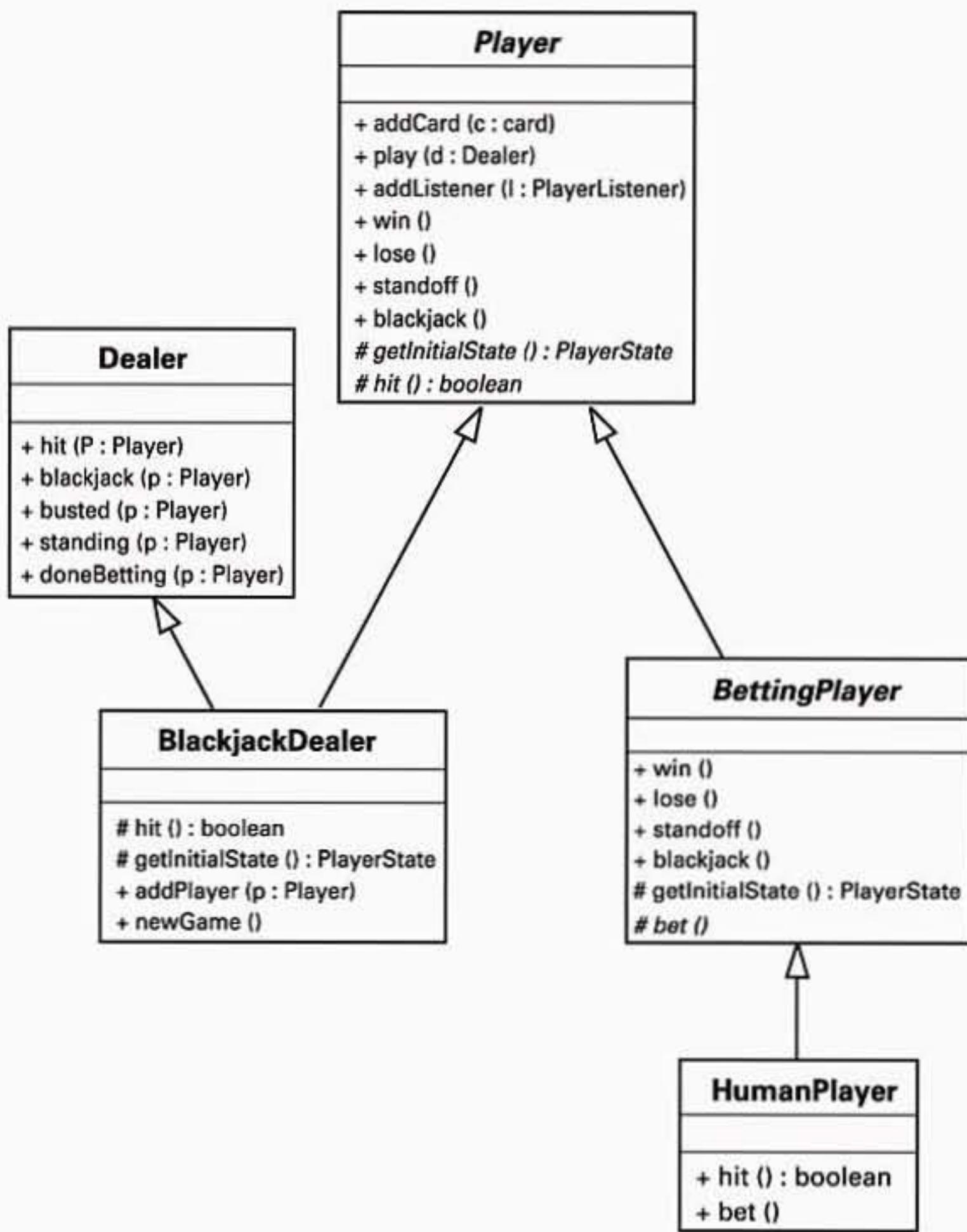
## Refazendo a hierarquia de Player (Jogador)

Neste ponto do projeto, parece que *Player (Jogador)* e *BlackjackDealer (Banca 21)* estão divergindo. Embora *BlackjackDealer (Banca 21)* estenda *Player (Jogador)*, ela não precisa de *Bank (Banco)*. Isso é diferente de *HumanPlayer (Jogador Humano)*, pois a banca não aposta. Se você adicionar suporte para aposta diretamente em *Player (Jogador)*, *BlackjackDealer (Banca 21)* herdará todos os tipos de comportamento inútil, que precisará sobrepor (além de *Player (Jogador)* ficar demasiadamente congestionado).

Este é um bom momento para refazer a hierarquia de herança *Player (Jogador)*, dividindo os elementos comuns em subclasses. Na nova hierarquia, nenhum suporte para apostas deve ser adicionado na classe base *Player (Jogador)*. Em vez disso, uma nova classe, *BettingPlayer*, deve herdar de *Player (Jogador)* e, então, adicionar os estados e métodos necessários para o suporte de apostas.

`BlackjackDealer` (Banca 21) pode continuar a herdar de `Player` (Jogador); entretanto, `HumanPlayer` (Jogador Humano) deve agora herdar de `BettingPlayer`. A Figura 17.6 ilustra a hierarquia resultante.

**FIGURA 17.6**  
A hierarquia de  
*Player* (jogador).



## O modelo do jogo vinte-e-um atualizado

Agora que você terminou o projeto, é uma boa idéia atualizar o diagrama da classe `Blackjack`. A Figura 17.7 ilustra o diagrama da classe.

Agora, você está pronto para passar para a implementação.

## Implementação da aposta

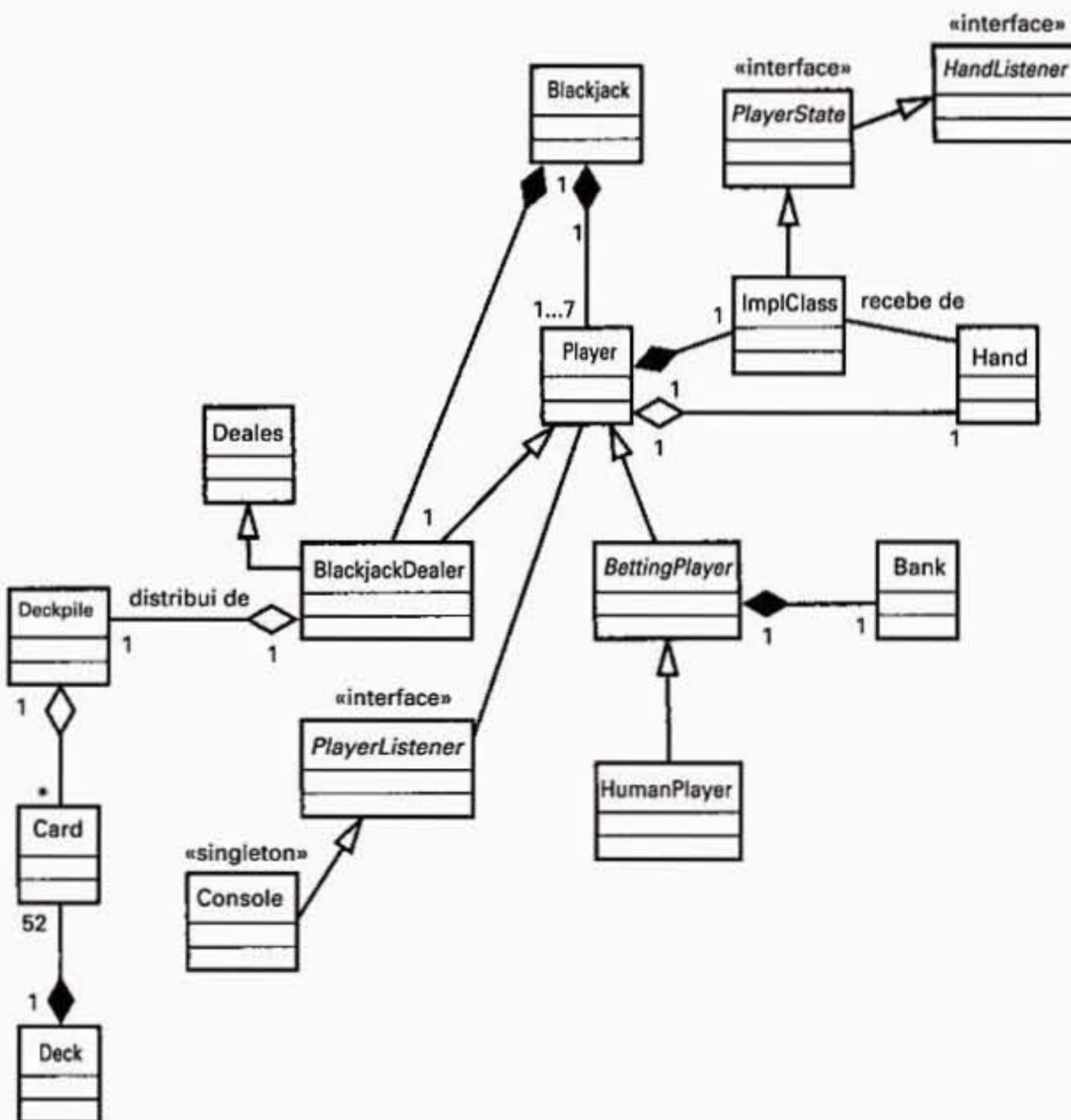
A implementação da aposta exigirá a criação das classes `Bank` e `BettingPlayer`, assim como alterações em `BlackjackDealer`, `Dealer` e `HumanPlayer`. Vamos começar com a classe `Bank`.

## A implementação de Bank

Conforme você descobriu, Bank (Banco) é responsável por conter um total, assim como por gerenciar as apostas. A Listagem 17.1 apresenta uma possível implementação de Bank (Banco).

**FIGURA 17.7**

O diagrama da classe Blackjack.



17

**LISTAGEM 17.1** Bank.java

---

```

public class Bank {

    private int total;
    private int bet;

    public Bank( int amount ) {
        total = amount;
    }

    public void place100Bet() {
        placeBet( 100 );
    }
}
  
```

**LISTAGEM 17.1** Bank.java (*continuação*)

```
public void place50Bet() {
    placeBet( 50 );
}

public void place10Bet() {
    placeBet( 10 );
}

public void win() {
    total += ( 2 * bet );
    bet = 0;
}

public void lose() {
    // já extraído do total
    bet = 0;
}

public void blackjack() {
    total += ( ( 3 * bet ) / 2 ) + bet );
    bet = 0;
}

public void standoff() {
    total += bet;
    bet = 0;
}

public String toString() {
    return ( "$" + total + ".00" );
}

private void placeBet( int amount ) {
    bet = amount;
    total -= amount;
}
```

---

Quando o jogador precisa fazer uma aposta, ele faz isso através da interface Bank. Digno de nota é o modo como Bank oculta completamente os detalhes da aposta. Quando o jogador ganha, perde, atinge vinte-e-um ou empata, ele simplesmente informa a Bank. Bank faz o resto.

## A implementação de BettingPlayer

BettingPlayer precisa herdar de Player, definir um estado *Aposta* (Betting), garantir que seu estado inicial seja configurado como o estado *Aposta* e adicionar suporte para Bank (assim como atualizá-lo corretamente). A Listagem 17.2 apresenta a nova classe BettingPlayer.

### LISTAGEM 17.2 BettingPlayer.java

```
public abstract class BettingPlayer extends Player {  
  
    private Bank bank;  
  
    public BettingPlayer( String name, Hand hand, Bank bank ) {  
        super( name, hand );  
        this.bank = bank;  
    }  
    //*****  
    // comportamento sobreposto  
    public String toString() {  
        return ( getName() + ": " + getHand().toString() + "\n" +  
bank.toString() );  
    }  
  
    public void win() {  
        bank.win();  
        super.win();  
    }  
  
    public void lose() {  
        bank.lose();  
        super.lose();  
    }  
  
    public void standoff() {  
        bank.standoff();  
        super.standoff();  
    }  
  
    public void blackjack() {  
        bank.blackjack();  
        super.blackjack();  
    }  
  
    protected PlayerState getInitialState() {  
        return getBettingState();  
    }
```

**LISTAGEM 17.2 BettingPlayer.java (continuação)**

```
//*****  
// adicionado recentemente para BettingPlayer  
protected final Bank getBank() {  
    return bank;  
}  
  
protected PlayerState getBettingState() {  
    return new Betting();  
}  
  
protected abstract void bet();  
  
private class Betting implements PlayerState {  
    public void handChanged() {  
        // impossível no estado de estouro  
    }  
    public void handPlayable() {  
        // impossível no estado de estouro  
    }  
    public void handBlackjack() {  
        // impossível no estado de estouro  
    }  
    public void handBusted() {  
        // impossível no estado de estouro  
    }  
    public void execute( Dealer dealer ) {  
        bet();  
        setCurrentState( getWaitingState() );  
        dealer.doneBetting( BettingPlayer.this );  
        // termina  
    }  
}
```

---

Você também notará que BettingPlayer acrescenta um novo método abstrato: `protected abstract void bet()`. O método `bet()` é chamado dentro da atividade do estado Betting (Aposta). Cada subclasse deve sobrepor esse método, como achar melhor.

### Mudanças em Dealer e HumanPlayer

A partir do exame do código de BettingPlayer, você provavelmente notou que um novo método foi adicionado em Dealer: `public void doneBetting( Player p )`.

BettingPlayer chama esse método para informar a Dealer que ele terminou a aposta. Desse modo, o objeto Dealer pode saber que o jogador terminou e que o próximo jogador pode começar a apostar.

As alterações em HumanPlayer são muito sem graça. A Listagem 17.3 lista a nova classe HumanPlayer.

### LISTAGEM 17.3 HumanPlayer.java

```
public class HumanPlayer extends BettingPlayer {

    private final static String HIT      = "H";
    private final static String STAND    = "S";
    private final static String PLAY_MSG = "[H]it or [S]tay";
    private final static String BET_MSG  = "Place Bet:[10] [50] or [100]";
    private final static String BET_10    = "10";
    private final static String BET_50    = "50";
    private final static String BET_100   = "100";
    private final static String DEFAULT  = "invalid";

    public HumanPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    protected boolean hit() {
        while( true ) {
            Console.INSTANCE.printMessage( PLAY_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if(response.equalsIgnoreCase( HIT ) ) {
                return true;
            } else if( response.equalsIgnoreCase( STAND ) ) {
                return false;
            }
            // se chegarmos até aqui, faz um laço até obtermos entrada
            // significativa
        }
    }

    protected void bet() {
        while( true ) {
            Console.INSTANCE.printMessage( BET_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equals( BET_10 ) ) {
                getBank().place10Bet();
                return;
            }
        }
    }
}
```

**LISTAGEM 17.3** HumanPlayer.java (*continuação*)

```
    if( response.equals( BET_50 ) ) {
        getBank().place50Bet();
        return;
    }
    if( response.equals( BET_100 ) ) {
        getBank().place100Bet();
        return;
    }
    // se chegarmos até aqui, faz um laço até obtermos entrada
    // significativa
}
}
```

---

Agora, HumanPlayer herda de BettingPlayer, em vez de diretamente de Player. HumanPlayer também fornece uma implementação de bet(). Quando bet() é chamado, ele consulta a linha de comando para ver retorno do usuário.

## Mudanças em BlackjackDealer

BlackjackDealer (Banca 21) implementa o novo método doneBetting(), definido em Dealer. Quando esse método é chamado, BlackjackDealer pega o jogador e o insere em um recipiente de jogadores esperando.

BlackjackDealer também define um novo estado: DealerCollectingBets. Além disso, DealerCollectingBets atua como o novo estado inicial de BlackjackDealer. A Listagem 17.4 apresenta o novo estado.

**LISTAGEM 17.4** O novo estado de DealerCollectingBets

```
private class DealerCollectingBets implements PlayerState {
    public void handChanged() {
        // impossível no estado de aposta
    }
    public void handPlayable() {
        // impossível no estado de aposta
    }
    public void handBlackjack() {
        // impossível no estado de aposta
    }
    public void handBusted() {
        // impossível no estado de aposta
    }
}
```

**LISTAGEM 17.4** O novo estado de DealerCollectingBets} (continuação)

```
public void execute( Dealer dealer ) {
    if( !betting_players.isEmpty() ) {
        Player player = (Player) betting_players.get( 0 );
        betting_players.remove( player );
        player.play( dealer );
    } else {
        setCurrentState( getDealingState() );
        getCurrentState().execute( dealer );
        // faz a transição e executa
    }
}
```

O novo estado chega ao fim e diz a cada jogador para que aposte. Digno de nota é que esse estado não faz laço. Em vez disso, a atividade é executada sempre que um jogador indica que acabou de apostar. Esse comportamento está definido dentro do método doneBetting():

```
public void doneBetting( Player player ) {
    waiting_players.add( player );
    play( this );
}
```

Lembre-se de que uma chamada a `play()` executa o estado corrente.

## Miscelânea de mudanças

A única outra alteração digna de nota é o fato de que o método `getInitialState()` de `Player` agora é declarado como abstrato na classe base `Player`. Tornar o método abstrato funciona como uma forma de documentação que permite a qualquer um, que faça subclasses da classe, saber que deve fornecer sua própria definição de estado inicial.

A prática de tornar um método abstrato para que ele funcione como uma forma de documentação é uma maneira eficiente de estabelecer um protocolo de herança.

## Um pequeno teste: um objeto falsificado

Como sempre, casos de teste estão disponíveis junto com o código-fonte; entretanto, é interessante dar uma olhada em um uso inteligente de objetos falsificados. A Listagem 17.5 apresenta um objeto falsificado `Deckpile` que garante que a banca recebe um vinte-e-um.

**LISTAGEM 17.5** DealerBlackjackPile.java

```
public class DealerBlackjackPile extends Deckpile {
    private Card [] cards;
```

**LISTAGEM 17.5 DealerBlackjackPile.java (continuação)**

```
private int index = -1;

public DealerBlackjackPile() {
    cards = new Card [4];
    cards [0] = new Card( Suit.HEARTS,Rank.TWO );
    cards [1] = new Card( Suit.HEARTS,Rank.ACE );
    cards [2] = new Card( Suit.HEARTS,Rank.THREE );
    cards [3] = new Card( Suit.HEARTS,Rank.KING );
}

public void shuffle() {
    // não faz nada
}

public Card dealUp() {
    index++;
    cards[index].setFaceUp( true );
    return cards [index];
}

public Card dealDown() {
    index++;
    return cards [index];
}

public void reset() {
    // não faz nada
}
}
```

---

Você pode usar esse objeto falsificado para testar se o jogo responde corretamente quando a banca recebe uma mão de vinte-e-um. Esse objeto falsificado realmente prepara o baralho para roubar no jogo.

## Resumo

Hoje, você concluiu a terceira iteração do projeto do jogo vinte-e-um — resta apenas mais uma!

Neste capítulo, você viu como poderia estender a arquitetura de estado para suportar aposta simples no jogo. Você também viu que às vezes é necessário rever e refazer uma hierarquia, quando novos requisitos se apresentam. Refazer pensadamente representa um pouco de trabalho extra antecipado; refazer quando apropriado tende a compensar quando você prosseguir.

Como você refez as hierarquias agora, a base de código será muito mais fácil de entender, estender e manter posteriormente.

Amanhã, você colocará uma UI gráfica no jogo.

## Perguntas e respostas

**P Por que você não modelou o cumprimento como um estado?**

**R** Você poderia ter modelado o cumprimento como um estado; entretanto, um estado de cumprimento teria significado ruído no projeto. As atividades dentro dos estados *Busted*, *Blackjack* e *Standing* podem efetuar o jogo adequadamente. Além disso, esses estados podem efetuar o jogo muito especificamente.

Se a banca fizesse a transição para um estado de cumprimento, ela perderia suas informações de estado anteriores. Se você efetuar dentro de um estado específico, entretanto, a banca poderá fazer a contagem do jogo facilmente, pois saberá em que estado terminou.

17

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Como você pode estabelecer protocolos de herança eficientemente?
2. A lição sobre herança, na Semana 1, disse que as hierarquias de herança são freqüentemente descobertas e não planejadas desde o início. Qual hierarquia você descobriu hoje?
3. Dada a Pergunta 2 do teste, por que você deve esperar para realizar a abstração até ter feito algo algumas vezes?
4. Hoje, você refez a hierarquia Player. Liste duas vantagens que você obteve fazendo as alterações.

## Exercícios

1. Faça download do código-fonte da iteração de hoje. Quando você tiver o código, compile-o, faça-o funcionar executando o jogo vinte-e-um e, depois, tente entender como ele funciona. Ter um entendimento total do código exigirá algum tempo e paciência.
2. Projete e implemente o caso de uso *Jogador dobra*. Baseie sua solução no código-fonte da lição de hoje.

PÁGINA EM BRANCO

# SEMANA 3

DIA **18**

## Iteração 4 do jogo vinte-e-um: adicionando uma GUI

Até aqui nesta semana, você analisou, projetou e construiu um jogo de cartas vinte-e-um. Trabalhando a partir de uma implementação simples e fazendo iterações para obter um aplicativo mais complexo, você adicionou regras e recursos de aposta no jogo. Hoje, você continuará o processo iterativo e fará melhorias na camada de apresentação do jogo.

Hoje você aprenderá como:

- Aplicar análise, projeto e implementação ao escrever interfaces com o usuário
- Aplicar o padrão MVC no jogo vinte-e-um

### Apresentação do jogo vinte-e-um

Até aqui, a única interface para o jogo de cartas vinte-e-um tem sido uma rudimentar interface com o usuário (UI) baseada em linha de comando. Não foi falado muito a respeito dessa UI. Na verdade, muito pouco, se é que houve, análise ou projeto aplicado a UI, apenas foi dito que você usará o padrão MVC. Em vez de fazer a análise e o projeto da UI de linha de comando, a UI mais simples possível foi criada para permitir que você interagisse com o sistema do jogo vinte-e-um.

Durante o desenvolvimento, você freqüentemente verá que precisa desenvolver materiais de suporte, como objetos stubs ou partes de interação do sistema com o usuário, como a UI. Freqüentemente, esses materiais não serão orientados pela análise. Em vez disso, esses itens são orientados por necessidades que se apresentam durante a implementação. No caso do jogo vin-

te-e-um, você precisa absolutamente de uma maneira para interagir com o sistema; entretanto, escrever uma UI gráfica desde o início simplesmente não era prático. Como a UI de linha de comando não se destinava a fazer parte do sistema final, não houve necessidade de realizar análise adicional para ela.

Hoje, você fará uma última otimização na UI de linha de comando original e depois passará para a análise, projeto e implementação de uma UI gráfica (GUI) completa para o jogo vinte-e-um.

## Otimizações da linha de comando

Antes de passarmos para o trabalho na GUI do jogo vinte-e-um, é interessante fazer uma última otimização na UI de linha de comando.

É uma inconveniência ter de reiniciar o jogo cada vez que você quer jogar. A Listagem 18.1 apresenta um novo método principal que permite a você jogar quantos jogos vinte-e-um quiser, sem ter de reiniciar.

---

### LISTAGEM 18.1 Blackjack.java

---

```
public class Blackjack {  
  
    public static void main( String [] args ) {  
        Deckpile cards = new Deckpile();  
        for( int i = 0; i < 4; i ++ ) {  
            cards.shuffle();  
            Deck deck = new Deck();  
            deck.addToStack( cards );  
            cards.shuffle();  
        }  
  
        Hand dealer_hand = new Hand();  
        BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand,  
cards );  
        Bank human_bank = new Bank( 1000 );  
        Hand human_hand = new Hand();  
        Player player = new CommandLinePlayer( "Human", human_hand, human_bank  
);  
        dealer.addListener( Console.INSTANCE );  
        player.addListener( Console.INSTANCE );  
        dealer.addPlayer( player );  
  
        do {  
            dealer.newGame();  
        } while( playAgain() );  
    }  
}
```

**LISTAGEM 18.1** Blackjack.java (*continuação*)

```
        Console.INSTANCE.printMessage( "Thank you for playing!" );  
  
    }  
  
    private static boolean playAgain() {  
        Console.INSTANCE.printMessage( "Would you like to play again? [Y]es  
[N]o" );  
        String response = Console.INSTANCE.readInput( "invalid" );  
        if( response.equalsIgnoreCase( "y" ) ) {  
            return true;  
        }  
        return false;  
    }  
}
```

Adicionar essa funcionalidade no método principal do jogo vinte-e-um tem um valor prático, pois ela nos ajuda a detectar todos os erros que possam estar ocultos no programa, quando você jogar várias vezes. Por exemplo, cada mão precisa ser corretamente reconfigurada, antes do próximo jogo. Levantando todos os erros agora, você não será pego pelo erro posteriormente e não achará que a nova GUI é a culpada.

18

## Análise da GUI do jogo vinte-e-um

Para completar a análise da GUI, você deve realizar a análise de caso de uso, exatamente como fez durante as iterações anteriores. Ao se realizar a análise da GUI, também é fundamental que você se sente com seus clientes, usuários e especialistas na utilização, para projetar o layout da GUI. Na realidade, quanto menos palpite você der como desenvolvedor no layout da GUI, melhor. Todo mundo tem sua própria especialidade. Como desenvolvedor, sua especialidade normalmente é analisar problemas, projetar soluções e implementar essas soluções. Quando você se sentar com seu cliente, vai descobrir como ele quer sua GUI configurada.

Infelizmente, os especialistas em utilização, clientes e usuários não aparecem na forma conveniente de um livro; portanto, você precisará passar sem eles hoje. Em vez disso, trabalharemos em um esboço da tela, antes de passarmos para o projeto.

### Casos de uso da GUI

Ao contrário dos casos de uso do jogo vinte-e-um que você analisou durante as iterações anteriores, os casos de uso da GUI não afetam o domínio do jogo em si. Em vez disso, os casos de uso da GUI ajudam a estabelecer como o usuário vai manipular o jogo vinte-e-um através da UI.

Desse modo, o primeiro caso de uso que você precisa investigar é aquele que inicia um novo jogo:

Quando o programa está iniciando pela primeira vez ou o jogador acabou de jogar um jogo, ele pode optar por jogar um novo jogo.

- Novo jogo com GUI
  1. O jogador clica no botão New Game e um novo jogo começa.
- Condições prévias
  - O jogador deve ter acabado de iniciar o programa ou acabado de terminar um jogo.
- Condições posteriores
  - Novo jogo iniciado.

Conforme você pode ver, esse caso de uso não altera o domínio do jogo vinte-e-um. Ele simplesmente configura as regras básicas da UI. O próximo caso de uso da UI analisa as apostas:

Os jogadores começam o jogo com US\$1000 em seu pote. Antes que qualquer carta seja distribuída, cada jogador deve fazer uma aposta. Começando com o primeiro jogador, cada jogador aposta um valor de US\$10, US\$50 ou US\$100. Se um jogador ficar abaixo de US\$0, ele ainda poderá jogar. O valor em seu pote é refletido como um número negativo.

- Jogador faz aposta com GUI
  1. O jogador seleciona um dos seguintes níveis de aposta: 10, 50 ou 100 e faz a aposta imediatamente.
  2. A aposta passa para o jogador seguinte e se repete até que todos os jogadores tenham feito uma aposta.
- Condições prévias
  - Novo jogo iniciado.
  - Condições posteriores.
  - O jogador fez uma aposta.
- A banca pode começar a dar as cartas

Você ainda precisa de suporte para dar mais cartas e para parar. O próximo caso de uso identifica o ato de dar mais cartas:

O jogador decide que não está satisfeito com sua mão. O jogador ainda não estourou; portanto, ele decide receber mais cartas. Se o jogador não estourar, ele pode optar por receber mais cartas novamente ou parar. Se o jogador estourar, o jogo passa para o jogador seguinte.

- Jogador recebe mais cartas com GUI
  1. O jogador decide que não está satisfeito com sua mão.
  2. O jogador clica no botão Hit, que solicita outra carta da banca.

3. O jogador pode optar por receber mais cartas novamente ou parar, caso o total em sua mão seja menor ou igual a 21.
  - Condições prévias
    - O jogador tem uma mão cujo valor total é menor que 21.
    - O jogador não tem vinte-e-um.
    - A banca não tem vinte-e-um.
  - Condições posteriores
    - Uma nova carta é acrescentada na mão do jogador.
  - Alternativa: Jogador estoura

A nova carta faz com que a mão do jogador seja maior do que 21. O jogador estoura (perde). Começa a vez do jogador seguinte/banca.

- Alternativa: a mão do jogador é maior do que 21, mas ele tem um ás
  - A nova carta faz com que a mão do jogador seja maior que 21. O jogador tem um ás. O valor do ás muda de 11 para 1, fazendo a mão do jogador ser menor ou igual a 21. O jogador pode optar por receber mais cartas novamente ou parar.

Se um jogador não quer receber mais cartas, ele deve parar. O próximo caso de uso analisa o uso da GUI para parar:

O jogador decide que está satisfeito com sua mão e pára.

18

- Jogador pára.
  1. O jogador decide que está contente com sua mão e clica no botão Stand.
- Condições prévias
  - O jogador tem uma mão cujo valor é menor ou igual a 21.
  - O jogador não tem vinte-e-um.
  - A banca não tem vinte-e-um.
- Condições posteriores
  - A vez do jogador termina.

E, por último, mais não menos importante, você precisa considerar a saída do jogo:

O jogador decide que não quer mais jogar e sai.

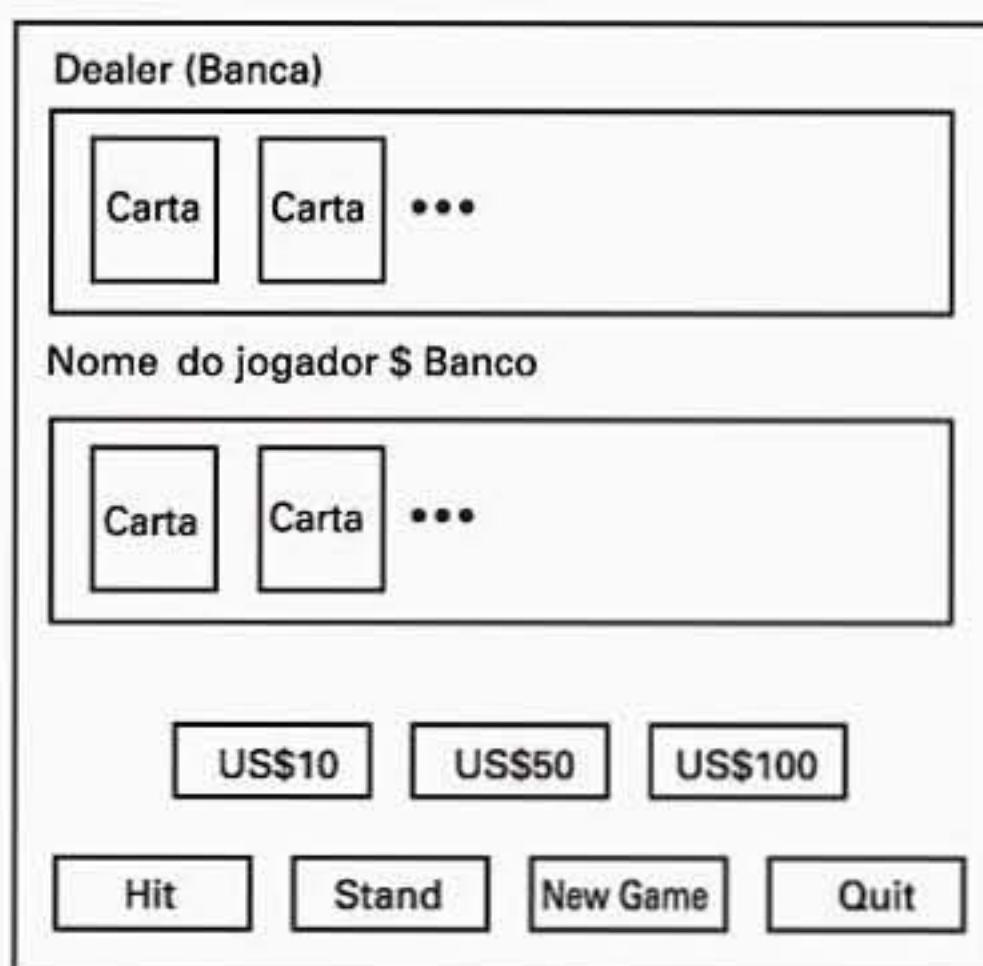
- Jogador sai do jogo com GUI
  1. O jogador clica no botão Quit.
  2. O jogo se fecha.
- Condições prévias
  - O jogo não deve estar em andamento (nenhum jogo foi iniciado ou o jogo foi concluído).

- Condições posteriores
  - Desligamento do jogo.

## Modelos visuais de GUI

Com base nos casos de uso enumerados na seção anterior, você precisará projetar o layout da GUI. A Figura 18.1 apresenta uma possível GUI que preenche todos os requisitos descobertos durante a análise.

**FIGURA 18.1**  
*A GUI do jogo vinte-e-um.*



Existem alguns comportamentos adicionais da GUI com que você pode trabalhar agora. A Figura 18.2 ilustra o status dos botões quando o usuário inicia o jogo pela primeira vez.

**FIGURA 18.2**  
*O status inicial dos botões.*



Todos os botões são visíveis quando o jogo começa; entretanto, apenas New Game e Quit estão ativos. A Figura 18.3 ilustra o status dos botões após um clique em New Game.

**FIGURA 18.3**  
*O status dos botões após clicar em New Game.*



O jogador deve fazer uma aposta após clicar em New Game. Como resultado, apenas os botões de aposta estão ativos. A Figura 18.4 ilustra o status dos botões após ter feito a aposta.

**FIGURA 18.4**  
*O status dos botões  
após fazer uma  
aposta.*



Após fazer uma aposta, um usuário pode receber mais cartas ou parar. Assim, apenas os botões Hit e Stand estão ativos. Todos os outros botões são desativados. Os botões permanecem nesse estado até que o jogador pare ou estoure, no ponto em que o jogo termina os botões voltam para o status ilustrado na Figura 18.2.

Após a conclusão de um jogo, as cartas permanecem na tela até que o usuário clique em New Game. As cartas são então removidas da tela. Durante o jogo, a mão gráfica do jogador é atualizada, quando uma carta é distribuída para ele.

Esse layout de GUI afetará fortemente os modos de visualização que você vai projetar na próxima seção.

## Projeto da GUI do jogo vinte-e-um

18

Projetar as classes que compõem uma GUI não é diferente de projetar qualquer outro tipo de classe. Você deve identificar as classes individuais e suas responsabilidades.

Usando a Figura 18.1 como ponto de partida, você pode gerar uma lista inicial de classes. Você precisará de uma classe para a tela principal, uma classe para ver um jogador e uma classe para visualizar as opções do jogador.

## Cartões CRC da GUI

Uma sessão de cartão CRC pode ou não ser garantida aqui. Tudo depende de seu nível próprio de bem-estar. Para uma GUI maior, você definitivamente desejará passar por um número de sessões, para garantir que tenha feito um bom trabalho na divisão das responsabilidades.

Para nossos propósitos, a GUI do jogo vinte-e-um é suficientemente simples para que se possa pular uma sessão de CRC completa. Em vez disso, listaremos as responsabilidades aqui.

### PlayerView

PlayerView é responsável por visualizar um objeto Player no jogo vinte-e-um. O modo de visualização deve apresentar a mão, o nome e o saldo do pote de Player (se aplicável). PlayerView é simplesmente um veículo de visualização. Desse modo, ele não exige um controlador. Ele precisa simplesmente receber e apresentar seu objeto Player.

## OptionView e OptionViewController

OptionView é responsável por visualizar as opções do jogador humano. OptionView também precisa responder à interação do usuário, de modo que exige um controlador: OptionViewController.

## CardView

CardView é responsável por visualizar os objetos Card individuais. CardView não é interativo; assim, ele não exige um controlador. PlayerView usará CardView para visualizar Hand.

## BlackjackGUI

BlackjackGUI é responsável por agregar e apresentar todos os diversos modos de visualização. Como BlackjackGUI atua como um shell simples, ele não exige um controlador.

## Miscelânea

CardView precisará de um modo para mapear um objeto Card em uma imagem, para exibição. Você pode implementar um longo *if/else* ou *case* para mapear o objeto Card dentro de CardView; entretanto, tal estratégia é horrível (para não mencionar que é lenta).

Em vez de criar uma estrutura condicional, você deve fazer uma subclasse de Deck e Card. Você pode chamar as duas classes resultantes de VDeck e VCard, respectivamente. VCard receberá um argumento construtor extra, o nome de um arquivo bitmap. VDeck construirá objetos VCard.

Como você passa Deckpile para BlackjackDealer, em vez de permitir que BlackjackDealer crie seu próprio maço de cartas, pode passar as cartas visuais de forma transparente para a banca.

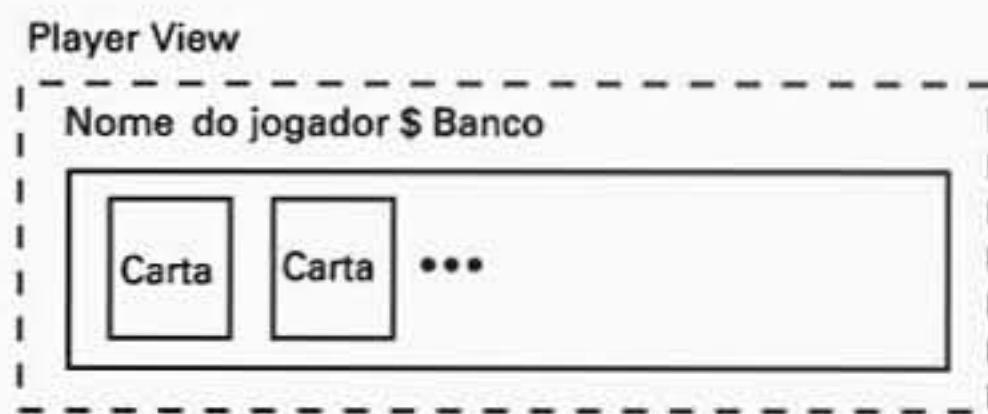
Você também precisará criar um novo jogador humano para a GUI. Esse novo objeto GUIPlayer pode herdar diretamente de BettingPlayer; entretanto, ele precisará fornecer seus próprios estados *Aposta* e *Jogo* personalizados.

Em vez de basear uma decisão no método *hit()* ou *bet()*, o objeto GUIPlayer precisará obter essa informação da GUI. Como resultado, você precisará de métodos que a GUI possa chamar para apostar, receber mais cartas e parar. Quando esses métodos forem chamados, eles colocarão o jogador nos estados corretos e comunicarão qualquer informação para a banca.

Ao todo, você precisará adicionar ou sobrepor os seguintes métodos em GUIPlayer: *place10Bet()*, *place50Bet()*, *place100Bet()*, *takeCard()*, *stand()*, *getBettingState()* e *getPlayingState()*.

## Estrutura da GUI

Às vezes, ao se trabalhar com uma GUI, ajuda esboçar o modo como as partes se encaixarão. Como a própria GUI é visual, seu esboço pode ser realmente um pouco mais poderoso do que os diagramas de classe padrão. A Figura 18.5 visualiza PlayerView.

**FIGURA 18.5***Visualizando PlayerView.*

Você vê que, na Figura 18.5, PlayerView é constituído de vários objetos CardView. PlayerView também desenha uma borda em torno de si mesmo, com o nome e o saldo do pote (se aplicável) de Player, no canto superior esquerdo.

Felizmente, o pacote Java `javax.swing.JPanel` fornece toda a funcionalidade que você precisa para dispor componentes, assim como para desenhar uma borda rotulada.

Continuando, a Figura 18.6 visualiza OptionView.

**FIGURA 18.6***Visualizando OptionView.*

18

OptionView é simplesmente um conjunto de botões. Uma combinação de `javax.swing.JPanel` (para aninhar os botões) e `javax.swing.JButton` deve fornecer tudo que você precisa para implementar esse modo de visualização.

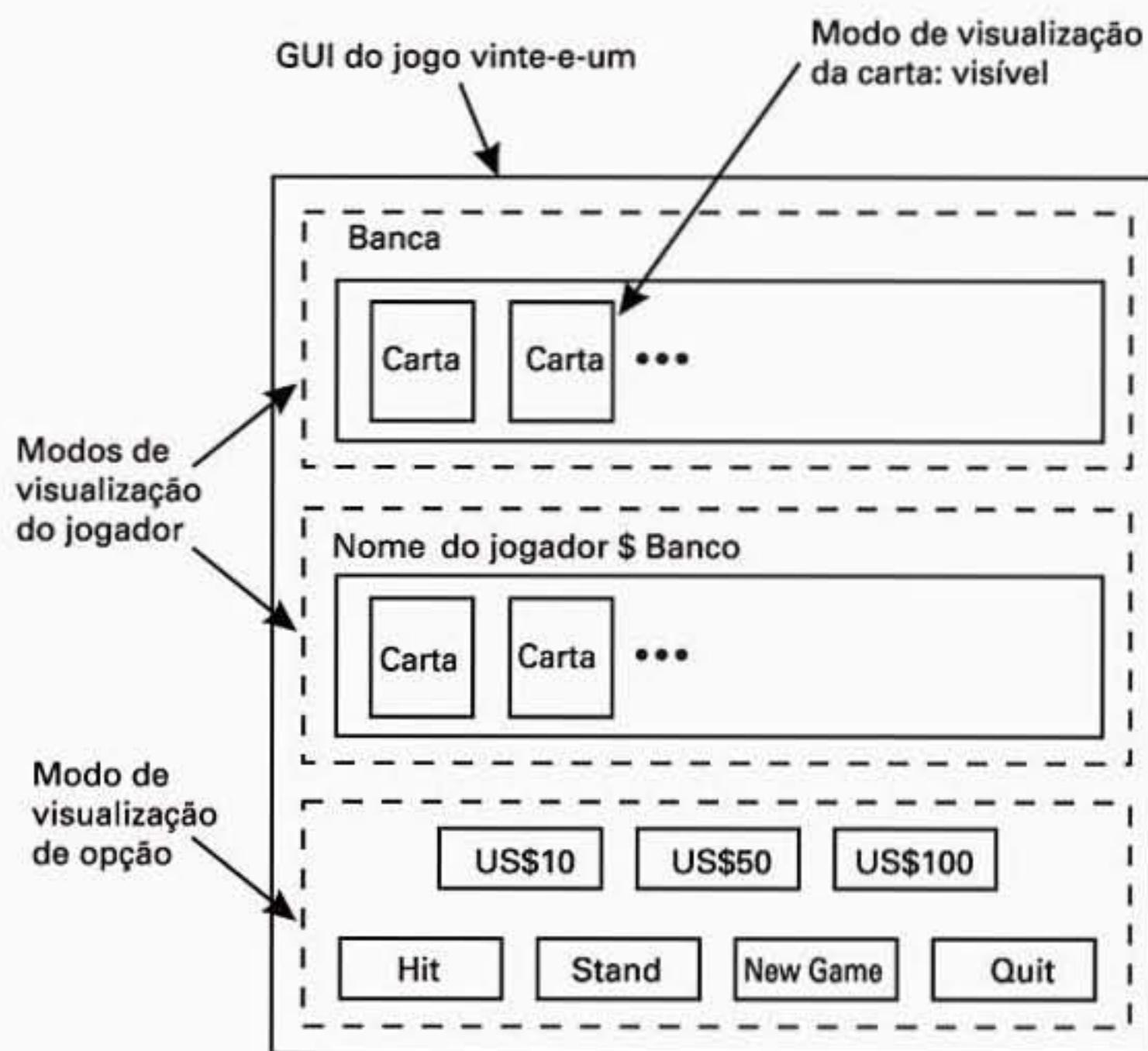
A Figura 18.7 reúne visualmente todas as partes.

As figuras anteriores devem ajudar a visualizar como todos os modos de visualização se encaixam. Entender como as partes se encaixam pode ajudar durante a implementação de uma GUI.

## Refazendo

Agora que existem dois tipos de jogadores humanos — GUI e CLUI — provavelmente faz sentido renomear `HumanPlayer` como `CommandLinePlayer`. Você deve fazer essa alteração hoje.

**FIGURA 18.7**  
*A janela principal dividida por modos de visualização.*



## Diagrama de classes da GUI

Agora que todas as novas classes estão identificadas, você pode modelar a estrutura de classes resultante. Assim como o diagrama de classes do Capítulo 17, “Iteração 3 do jogo vinte-e-um: adicionando aposta”, o modelo apresentado na Figura 18.8 focaliza a estrutura.

## Implementação da GUI do jogo vinte-e-um

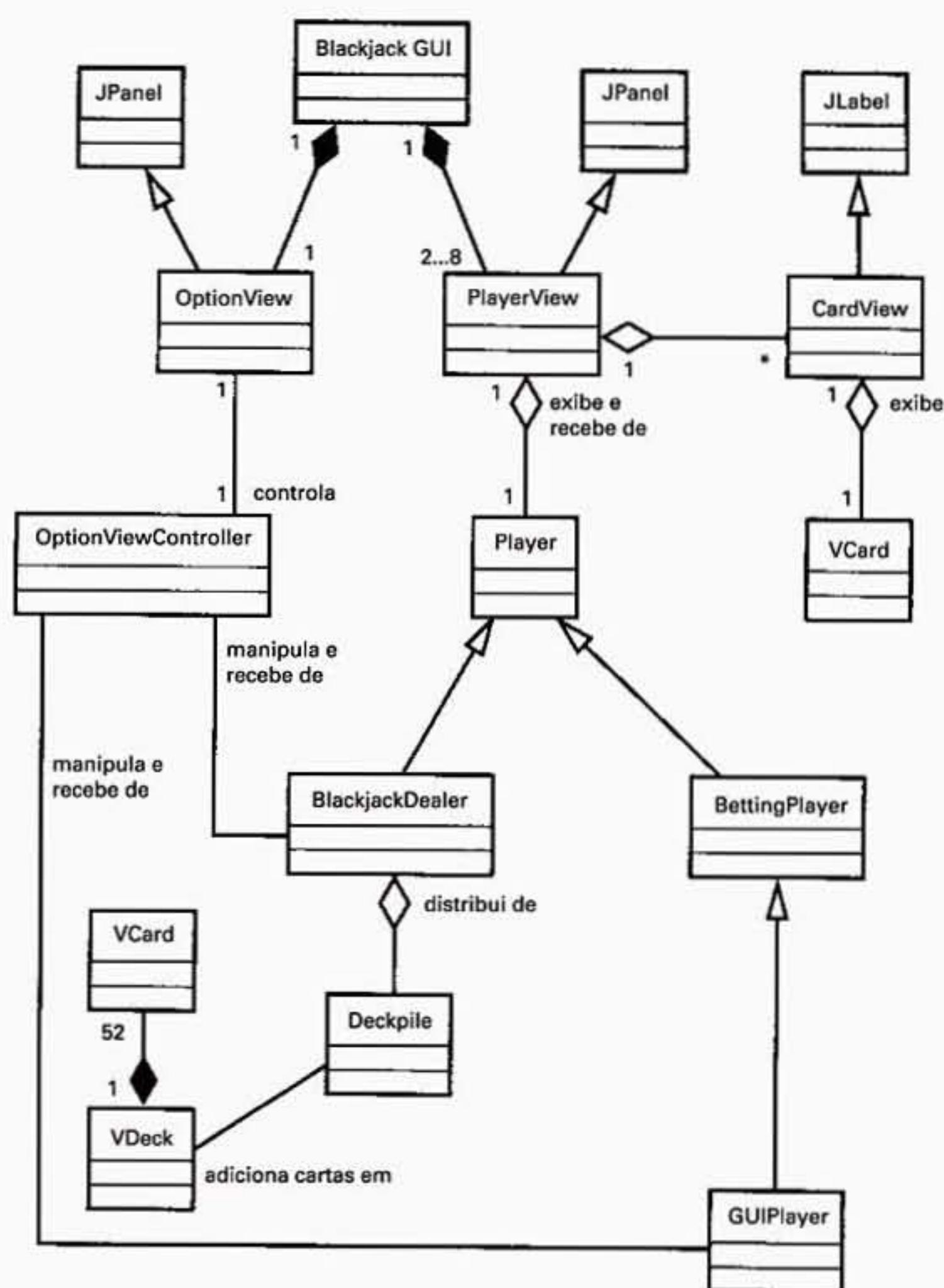
Ao se implementar uma GUI, em geral, é mais fácil trabalhar de baixo para cima. Nesse sentido, você deve implementar na seguinte ordem: VCard, VDeck, CardView, PlayerView, OptionView, OptionViewController, GUIPlayer e BlackjackGUI. Vamos examinar os destaques de cada classe.

## Implementando VCard, VDeck e CardView

VCard tem uma implementação relativamente simples, pois apenas acrescenta mais um atributo à classe Card. A Listagem 18.2 apresenta a nova definição da classe VCard.

**FIGURA 18.8**

A estrutura de classes da GUI.



18

**LISTAGEM 18.2** VCard.java

```

public class VCard extends Card {

    String image;

    public Vcard( Suit suit, Rank rank, String image ) {
        super( suit, rank );
        this.image = image;
    }

    public String getImage() {
        if( isFaceUp() ) {
            return image;
        }
    }
}
  
```

**LISTAGEM 18.2 VCard.java (continuação)**

```
    } else {
        return "/bitmaps/empty_pile.xbm";
    }
}
```

---

A implementação de VDeck é simples. Para criar objetos VCard, em vez de Card, você precisará sobrepor o método buildCards() de Deck. Para sobrepor o método, primeiro você precisará mudar o método para protegido em Deck. Originalmente, o método era privado. A Listagem 18.3 mostra uma listagem parcial da implementação de VDeck.

**LISTAGEM 18.3 VDeck.java**

```
public class VDeck extends Deck {

    protected void buildCards() {

        // Isso é horrível, mas é melhor do que a alternativa laço/if/else
        Card [] deck = new Card [52];
        setDeck( deck );

        deck [0] = new Vcard( Suit.HEARTS, Rank.TWO, "/bitmaps/h2" );
        deck [1] = new Vcard( Suit.HEARTS, Rank.THREE, "/bitmaps/h3" );
        deck [2] = new Vcard( Suit.HEARTS, Rank.FOUR, "/bitmaps/h4" );
        deck [3] = new Vcard( Suit.HEARTS, Rank.FIVE, "/bitmaps/h5" );
        deck [4] = new Vcard( Suit.HEARTS, Rank.SIX, "/bitmaps/h6" );
        deck [5] = new Vcard( Suit.HEARTS, Rank.SEVEN, "/bitmaps/h7" );
        deck [6] = new Vcard( Suit.HEARTS, Rank.EIGHT, "/bitmaps/h8" );
        // restante cortado por brevidade
    }
}
```

---

Para a GUI, usaremos um conjunto de bitmaps que está contido no diretório bitmaps, junto com o download do código-fonte. Os nomes dos bitmaps seguem uma convenção de atribuição de nomes específica; portanto, você também pode implementar buildCards() como um laço. Embora seja horrível, simplesmente codificar os valores é um pouco mais fácil de entender (e manter).

**ALERTA**

Codificar a criação da carta também pode não ser a solução mais fácil de manter. O problema é que cada estratégia que você possa adotar tem uma deficiência. VDeck é um exemplo de uma dessas ocasiões em que você deve fazer uma escolha entre dois males e conviver com ele.

A solução delineada anteriormente é falha, devido aos erros inerentes na digitação de todas as chamadas. Além disso, se o construtor mudar, você precisará atualizar cada chamada.

Como alternativa, você poderia fazer um laço pela representação de List de Ranks. Tal solução o obriga a supor uma ordem específica dos elementos na lista (para que você possa gerar corretamente o nome de arquivo da imagem). Se a ordenação mudar, o laço será desfeito misteriosamente. Qualquer um que mantiver o código terá dificuldade para encontrar a fonte do erro. Evitamos a estratégia do laço porque alterações em uma classe não relacionada poderia danificar VDeck.

CardView exibirá o bitmap VCard. javax.swing.JLabel fornece a funcionalidade necessária para exibir um bitmap. A Listagem 18.4 apresenta a implementação de CardView.

**LISTAGEM 18.4 CardView.java**

```
import javax.swing.*;
import java.awt.*;

public class CardView extends JLabel {

    private ImageIcon icon;

    public CardView( VCard card ) {
        getImage( card.getImage() );
        setIcon( icon );
        setBackground( Color.white );
        setOpaque( true );
    }

    private void getImage( String name ) {
        java.net.URL url = this.getClass().getResource( name );
        icon = new ImageIcon( url );
    }
}
```

**18**

CardView recebe um objeto VCard, extrai o caminho do bitmap, converte o caminho em uma url, cria um ImageIcon e adiciona o ícone em si mesmo. Isso é tudo que você precisa para carregar e exibir um bitmap!

## Implementando PlayerView

PlayerView exibirá qualquer subclasse de Player. Ao contrário de OptionView, que você verá na próxima seção, PlayerView só precisa apresentar o objeto Player; ele não aceita interação do usuário. Como resultado, a implementação é muito simples. A Listagem 18.5 apresenta o método que é chamado quando o objeto Player muda.

---

### **LISTAGEM 18.5** O código atualizado de PlayerView

---

```
public void playerChanged( Player player ) {  
    border.setTitle( player.getName() );  
    cards.removeAll();  
    Hand hand = player.getHand();  
    Iterator i = hand.getCards();  
    while(i.hasNext() ) {  
        VCard vcard = (Vcard) i.next();  
        JLabel card = new CardView( vcard );  
        cards.add( card );  
    }  
    revalidate();  
    repaint();  
}
```

---

Como você pode ver, o método `playerChanged()` extrai os objetos `VCard` de `Player` e cria um `CardView` para cada um. Finalmente, ele adiciona o modo de visualização em si mesmo, para que o objeto `VCard` seja apresentado.

A implementação apresentada aqui não é a mais eficiente, pois ela cria um novo `CardView` para cada objeto `VCard`, sempre que o objeto `Player` muda. Uma implementação mais eficiente poderia usar alguma cache do modo de visualização. Como você está usando objetos, pode mudar a implementação para outra mais eficiente, a qualquer momento. O desempenho parece estar normal, de modo que a sobrecarga da adição do uso de cache simplesmente não vale a pena neste ponto.

PlayerView também precisa apresentar o resultado do jogo de Player. A Listagem 18.6 apresenta dois métodos que são chamados no final do jogo de Player.

---

### **LISTAGEM 18.6** Um exemplo dos métodos PlayerListener de PlayerView

---

```
public void playerBusted( Player player ) {  
    border.setTitle( player.getName() + " BUSTED!!" );  
    cards.repaint();  
}  
  
public void playerBlackjack( Player player ) {
```

**LISTAGEM 18.6** Um exemplo dos métodos PlayerListener de PlayerView (continuação)

```
border.setTitle( player.getName() + " BLACKJACK!" );
cards.repaint();
}
```

Esses métodos configuram a borda do modo de visualização com o resultado do jogo. PlayerListener define mais do que dois métodos, mas assim como os dois listados aqui, a implementação dos métodos de PlayerView segue um padrão semelhante para todos. Examine o código-fonte, se você estiver interessado em ver a lista inteira de métodos de atualização.

## Implementando OptionView e OptionViewController

OptionView herda de JPanel e acrescenta vários botões em si mesmo. OptionView não recebe do modelo. Em vez disso, OptionViewController recebe do modelo e ativa ou desativa os botões em OptionView, conforme for apropriado.

Nenhuma das duas classes é muito interessante do ponto de vista da implementação. Se você estiver interessado nos detalhes específicos, faça download e leia o código.

## Implementando GUIPlayer

GUIPlayer provavelmente é a classe mais interessante dessa iteração. Ao implementar uma GUI, você deve lembrar que toda interação com o usuário é assíncrona — ela pode surgir a qualquer momento.

Escrever um jogador de linha de comando foi muito fácil. Você só teve que sobrepor hit() ou bet() para que ele fosse lido da linha de comando. Como a linha de comando fica bloqueada até receber a entrada do usuário, o jogador foi muito fácil de implementar. Um jogador com GUI é um pouco mais difícil de escrever.

Em vez de chamar um método e bloquear até recebermos a entrada, GUIPlayer precisa esperar até que o usuário decida clicar em um botão. Como resultado, todos os estímulos vêm de *fora* do jogador.

Em resposta a essa realidade, você precisa adicionar vários métodos que a GUI possa chamar para GUIPlayer. A Listagem 18.7 lista os métodos de aposta que você deve adicionar.

**LISTAGEM 18.7** Métodos de aposta de GUIPlayer

```
// esses métodos de aposta serão chamados pelo controlador da GUI
// para cada um: faz a aposta correta, muda o estado, permite que a
// banca saiba que o jogador terminou de apostar
public void place10Bet() {
    getBank().place10Bet();
    setCurrentState( getWaitingState() );
```

**LISTAGEM 18.7** Métodos de aposta de GUIPlayer (*continuação*)

```
    dealer.doneBetting( this );  
}  
  
public void place50Bet() {  
    getBank().place50Bet();  
    setCurrentState( getWaitingState() );  
    dealer.doneBetting( this );  
}  
  
public void place100Bet() {  
    getBank().place100Bet();  
    setCurrentState( getWaitingState() );  
    dealer.doneBetting( this );  
}
```

---

Você notará que esses métodos precisam fazer apostas e configurar o usuário com o estado correto. A Listagem 18.8 lista os métodos para receber mais cartas e para parar.

**LISTAGEM 18.8** Métodos para receber mais cartas e para parar de GUIPlayer

```
// takeCard será chamado pelo controlador da GUI, quando o jogador  
// decidir receber mais cartas  
public void takeCard() {  
    dealer.hit( this );  
}  
// stand será chamado pelo controlador da GUI, quando o jogador optar  
// por parar, quando a parada mudar de estado, deixa o mundo saber, e então  
// diz à banca  
public void stand() {  
    setCurrentState( getStandingState() );  
    notifyStanding();  
    getCurrentState().execute( dealer );  
}
```

---

Assim como os métodos de aposta, os métodos da Listagem 18.8 devem executar sua ação e atualizar o estado. Como o estado não pode simplesmente chamar `hit()` ou `bet()` para jogar ou apostar, você precisará fornecer alguns estados Jogo e Apostas personalizados. A Listagem 18.9 apresenta os métodos `getPlayingState()` e `getBettingState()` sobrepostos.

**LISTAGEM 18.9** Métodos de obtenção de estado sobrepostos de GUIPlayer

```
protected PlayerState getPlayingState() {  
    return new Playing();
```

**LISTAGEM 18.9** Métodos de obtenção de estado sobrepostos de GUIPlayer (cont.)

```
}
```

```
protected PlayerState getBettingState() {
    return new Betting();
}
```

Sobrepondo esses métodos, GUIPlayer pode fornecer seus próprios estados personalizados. A Listagem 18.10 apresenta o estado *Jogo* personalizado de GUIPlayer.



Métodos como `getPlayingState()` e `getBettingState()` são métodos factory.

**LISTAGEM 18.10** Estado de Jogo personalizado de GUIPlayer

```
private class Playing implements PlayerState {

    public void handPlayable() {
        // não faz nada
    }

    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        getCurrentState().execute( dealer );
    }

    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
        getCurrentState().execute( dealer );
    }

    public void handChanged() {
        notifyChanged();
    }

    public void execute( Dealer d ) {
        // não faz nada aqui, as ações virão da GUI, que é
        // externa ao estado, mas quando eventos chegarem, certifica-se de
        // impor a transição de estado imediatamente
    }
}
```

Quando executado, o estado *Jogo* personalizado não faz nada. Em vez disso, GUIPlayer precisa esperar pela interação assíncrona da GUI. Você notará que o estado *Jogo* ainda faz transições em resposta aos eventos de Hand.

A Listagem 18.11 apresenta o estado *Aposta* personalizado. Você notará que esse estado não faz nada. Em vez disso, GUIPlayer deve esperar que o jogador pressione algum botão na GUI. Quando isso acontecer, o botão chamará o método de aposta correto em GUIPlayer.

---

**LISTAGEM 18.11** Estado Aposta personalizado de GUIPlayer

```
private class Betting implements PlayerState {  
    public void handChanged() {  
        // impossível no estado de estouro  
    }  
    public void handPlayable() {  
        // impossível no estado de estouro  
    }  
    public void handBlackjack() {  
        // impossível no estado de estouro  
    }  
    public void handBusted() {  
        // impossível no estado de estouro  
    }  
    public void execute( Dealer d ) {  
        // não faz nada aqui, as ações virão da GUI, que é  
        // externa ao estado, pois nenhum evento vem como parte da  
        // aposta; o estado precisará ser mudado externamente para este estado  
    }  
}
```

---

## Reunindo tudo com BlackjackGUI

BlackjackGUI cria e exibe o sistema de jogo vinte-e-um. A Listagem 18.12 destaca o método *setUp()* de BlackjackGUI.

---

**LISTAGEM 18.12** Método *setUp()* de BlackjackGUI

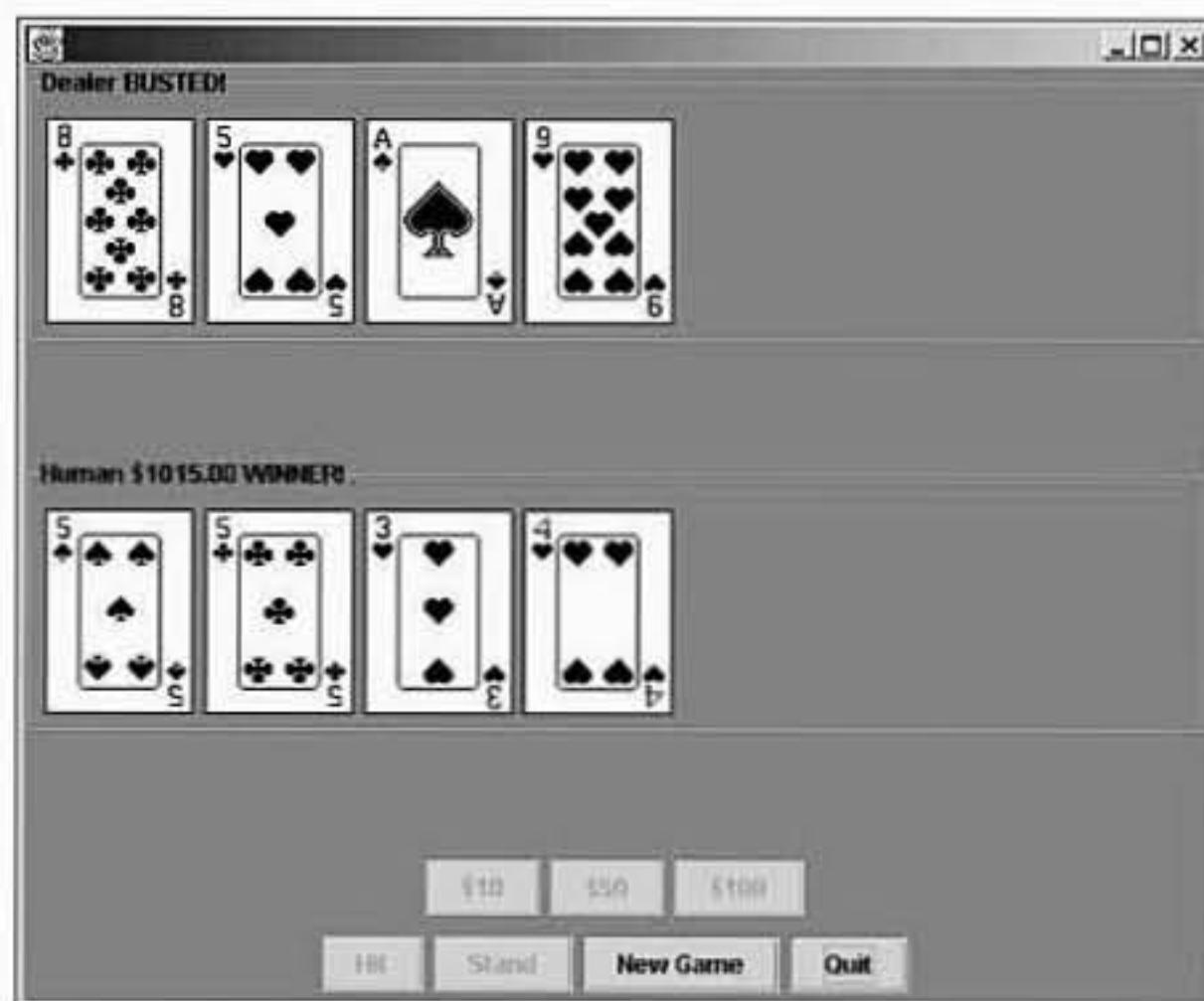
```
private void setUp() {  
    BlackjackDealer dealer = getDealer();  
    PlayerView v1 = getPlayerView( dealer );  
  
    GUIPlayer human = getHuman();  
    PlayerView v2 = getPlayerView( human );  
  
    PlayerView [] views = { v1, v2 };
```

**LISTAGEM 18.12** Método `setUp()` de `BlackjackGUI` (*continuação*)

```
addPlayers( views );  
  
dealer.addPlayer( human );  
  
addOptionView( human, dealer );  
}
```

O método `setUp()` cria cada jogador, os modos de visualização e coloca tudo junto. Os outros métodos principalmente constroem os vários objetos. Se você estiver interessado no código-fonte completo, examine o código. A Figura 18.9 ilustra a tela final do jogo.

**FIGURA 18.9**  
*A GUI do jogo  
vinte-e-um.*



18

## Resumo

Hoje, você viu o padrão MVC aplicado a um programa real. Às vezes, ajuda ver um exemplo estendido, para poder entender completamente um padrão. A lição de hoje também apresentou a questão de que uma GUI não é algo a ser pensado posteriormente. A GUI merece o mesmo nível de análise e projeto que qualquer outra parte de um sistema.

A lição de hoje conclui o jogo vinte-e-um. Amanhã, você verá um projeto e implementação alternativos da GUI do jogo vinte-e-um.

## Perguntas e respostas

**P** Você mencionou anteriormente que não deve simplesmente anexar a GUI no final. Bem, essa foi a última iteração e estamos adicionando uma GUI. Isso não vai contra o que você disse anteriormente?

**R** Não, absolutamente, não!

Quando dissemos ‘anexar’, queríamos dizer adicionar uma GUI sem realizar qualquer projeto. Nós planejamos uma GUI desde o início. No começo, dissemos que poderíamos usar MVC. Esse foi todo o projeto que precisamos realizar, até estarmos finalmente prontos para adicionar a GUI. Uma vez prontos para adicionar a GUI, fizemos mais um projeto, em uma iteração totalmente dedicada à GUI.

Dizer anteriormente que usariamos MVC e adicionar um mecanismo observador era tudo que precisávamos fazer para saber que pudessemos suportar uma GUI.

**P** Onde/quais estão/são as diferentes partes do MVC (encontramos menção de vários modos de visualização e de um controlador)? Desenvolva o que são o modelo e o controlador.

**R** O modelo é o sistema. Neste caso, BlackjackDealer, BettingPlayers etc., constituem a camada do modelo.

O projeto pediu apenas um controlador — OptionViewController — assim, não havia muito a dizer sobre controladores.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Como você usou herança e polimorfismo para introduzir uma carta “visual”?
2. Na lição sobre herança, foi mencionado que, se um método não é necessário para as classes externas e não existem requisitos para uma subclasse usar o método, então você deve defini-lo como privado. Se uma subclasse precisar dele, você pode torná-lo protegido nesse momento, mas não anteriormente. Encontre um exemplo dessa recomendação no projeto do jogo vinte-e-um.

## Exercícios

1. Faça download do código-fonte da iteração de hoje. O código está dividido em dois diretórios separados: `mvc_gui` e `exercise_2`.

`mvc_gui` contém o código da GUI que foi criada durante a iteração de hoje.

`exercise_2` contém os arquivos que você precisará para o Exercício 2, assim como as soluções.

Estude o código em `mvc_gui`. Tente entender como tudo funciona e depois complete o Exercício 2.

2. O Exercício 2 do Capítulo 17 pediu para que você adicionasse o recurso de apostar em dobro no jogo vinte-e-um. Você precisa adicionar esse recurso no jogo novamente. Desta vez, adicione-o na versão gráfica do jogo que você carregou por download para o Exercício 1. O download inclui todos os arquivos que você precisará para começar este exercício.

PÁGINA EM BRANCO

# SEMANA 3

DIA **19**

## Aplicando uma alternativa ao MVC

Ontem, você analisou, projetou e implementou uma GUI para o jogo vinte-e-um. Hoje, você vai usar uma outra estratégia para o projeto e implementação da GUI.

Hoje você aprenderá:

- Sobre uma alternativa para o padrão de projeto MVC
- Como aplicar uma GUI alternativa no jogo vinte-e-um
- Quando basear suas GUIs no MVC e quando não fazer isso

## Uma GUI alternativa do jogo vinte-e-um

Ontem, você criou uma GUI para o jogo vinte-e-um baseada no MVC. O MVC é apenas uma estratégia para o projeto de GUI. Hoje, você vai reprojetar e reimplementar a GUI, usando uma estratégia diferente.

A estratégia que você vai empregar hoje é uma especialização do padrão de projeto PAC (*Presentation Abstraction Control*). Assim como o padrão de projeto MVC, o padrão de projeto PAC divide o projeto da GUI em três camadas separadas:

- A camada de apresentação, que exibe o sistema
- A camada de abstração, que representa o sistema
- A camada de controle, que monta todos os componentes da camada de apresentação

As semelhanças do PAC com o padrão de projeto MVC são ilusórias. Na verdade, o PAC segue uma filosofia totalmente diferente daquela seguida pelo padrão MVC.

## As camadas do PAC

A camada de abstração do PAC é semelhante à camada de modelo no padrão MVC. A camada de abstração abriga a funcionalidade básica do sistema. É essa funcionalidade que a camada de apresentação exibe e manipula. A camada de abstração também é responsável por dar acesso aos objetos do nível de apresentação.

No PAC, as funcionalidades das camadas de visualização e controladora do MVC não são divididas; em vez disso, essas duas entidades são combinadas dentro da apresentação. A camada de apresentação é responsável por exibir e manipular a camada de abstração, assim como por responder à interação do usuário.

Como o controle e o modo de visualização MVC são combinados na camada de apresentação, o controle tem um objetivo totalmente diferente no PAC. No PAC, o controle monta todas as diferentes apresentações. Ele não recebe e responde à interação do usuário, como o controlador MVC.

## A filosofia do PAC

O MVC faz todo o possível para desacoplar completamente cada parte de seu projeto. Quando você usa o padrão MVC, é fácil trocar para novos modos de visualização em seu sistema, a qualquer momento; assim, usar o padrão MVC proporciona a você muita liberdade sobre como exibe seu sistema. Entretanto, o Capítulo 13, “OO e programação de interface com o usuário”, informa que a maior liberdade tem o preço do encapsulamento.

A estratégia do PAC é diferente. O PAC não desacopla as camadas de apresentação e abstração. Em vez disso, as duas camadas são fortemente acopladas. Isso não quer dizer, por exemplo, que Player estenderá JComponent diretamente. Isso quer dizer que a camada de abstração criará e retornará sua apresentação. Assim, Player e sua apresentação ainda são dois objetos separados.

Para obter uma apresentação diferente de uma parte da abstração, você precisará alterar a definição da abstração para que ela retorne um objeto de apresentação diferente. É um pouco mais difícil alterar a apresentação ou fornecer dois ou mais modos de visualização diferentes do mesmo sistema.

A construção da GUI é mais fácil, entretanto. Quando a camada de controle montar a tela, ela solicitará a cada membro da camada de abstração sua apresentação. Tudo que a camada de controle precisa fazer é adicionar essa apresentação na tela principal. Não há modo de visualização e controlador para reunir.

## Quando usar o padrão de projeto PAC

A suposição subjacente do PAC é que você não precisará fornecer vários modos de visualização do sistema. Em vez disso, quando usa o PAC, você precisa certificar-se de que o sistema tenha apenas uma interface bem definida. Se seu sistema vai ter apenas uma interface, o PAC pode fornecer uma alternativa muito elegante ao MVC.

O PAC tem várias vantagens. Como a camada de abstração pode criar sua apresentação, você não precisa destruir o encapsulamento de seu sistema. Em vez disso, você pode definir as classes de apresentação como classes internas. Como uma classe interna, a apresentação pode ter total acesso a sua classe de abstração progenitora. Quando ela precisa visualizar a abstração, pode acessar e exibir diretamente o estado da abstração.

O uso de PAC simplifica a comunicação entre a apresentação e sua abstração. Quando a abstração mudar, ela pode simplesmente chamar um método de atualização na apresentação que tiver criado.

Quando você usa o padrão PAC, pode considerar a apresentação como uma extensão direta da abstração. Atuando na apresentação, você atua diretamente no sistema subjacente. De certa forma, a apresentação age de forma muito parecida com um proxy para o sistema. Tal manipulação direta simplifica muito o projeto global.

## Analisando a GUI PAC do jogo vinte-e-um

Para aplicar o padrão de projeto PAC na GUI do jogo vinte-e-um, você pode simplesmente reutilizar a análise que fez ontem. Nada muda quanto à análise, pois você decide usar o padrão PAC em vez do MVC.

## Projetando a GUI PAC do jogo vinte-e-um

Para o jogo vinte-e-um existe apenas uma interface principal: uma GUI. Você não vai distribuir esse jogo como um aplicativo HTML da Web (embora você pudesse transformá-lo em um applet facilmente) ou como um PDA; assim, é seguro usar o padrão de projeto PAC.

É importante notar que nada o obriga a remover os mecanismos receptores que já construiu no sistema. Ainda é possível ter uma GUI baseada em linha de comando e uma GUI baseada no PAC, simultaneamente. Na verdade, através do uso cuidadoso de subclasses, você pode deixar todas as definições de classe originais intactas. Quando você quiser uma GUI, pode simplesmente instanciar as classes que suportam uma GUI. Quando você quiser um jogo de linha de comando, pode instanciar as classes antigas. Escolher MVC ou PAC não o impede necessariamente de usar o outro.

Assim como no Capítulo 18, “Iteração 4 do jogo vinte-e-um: adicionando uma GUI”, você pode pegar o projeto e implementação completos do Capítulo 17, “Iteração 3 do jogo vinte-e-um: ad-

cionando aposta”, como base para a nova GUI. O que você precisa fazer é descobrir quais dessas classes precisam de seus próprios objetos de apresentação. Quando você tiver essas classes identificadas, precisará projetar a camada de abstração. Quando isso estiver feito, você poderá projetar a camada de controle.

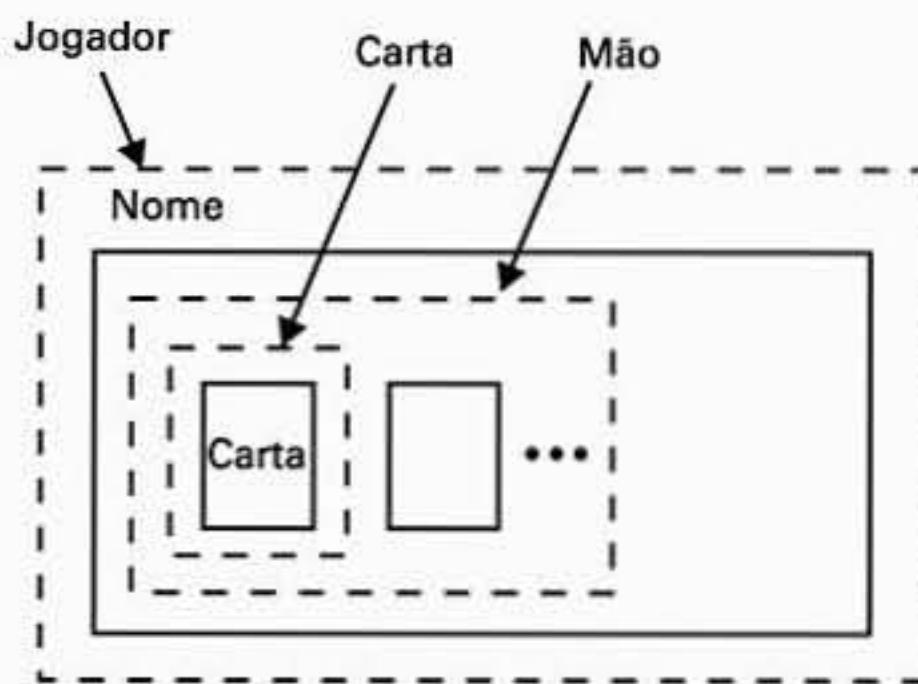
## Identificando os componentes da camada de apresentação

Para identificar os componentes da camada de apresentação, ajuda fazer alguns esboços da GUI. Desta vez, você deve associar partes da tela à classe subjacente, em vez de associá-las a um modo de visualização separado.

A Figura 19.1 isola uma parte da GUI. Dissecando cuidadosamente esse segmento da tela, você pode identificar alguns componentes da apresentação.

**FIGURA 19.1**

*Um segmento da tela.*



Dividindo o segmento da tela em partes, você pode ver que Card, Hand e Player precisarão fornecer objetos de apresentação. A Figura 19.2 dissecava a parte restante da tela.

**FIGURA 19.2**

*Os botões da GUI.*



Todos os botões pertencem ao objeto Player humano; portanto, a classe que representa o jogador humano precisará estender a apresentação de Player e adicionar botões. GUIPlayer deve ter um projeto igual àquele criado para o Capítulo 18. Em vez de repetir esse projeto aqui, volte e leia a seção “Implementando GUIPlayer”, no Capítulo 18, se você precisar de um lembrete. A

única diferença entre a GUIPlayer do Capítulo 18 e esta é que esta também fornecerá uma apresentação de si mesma.

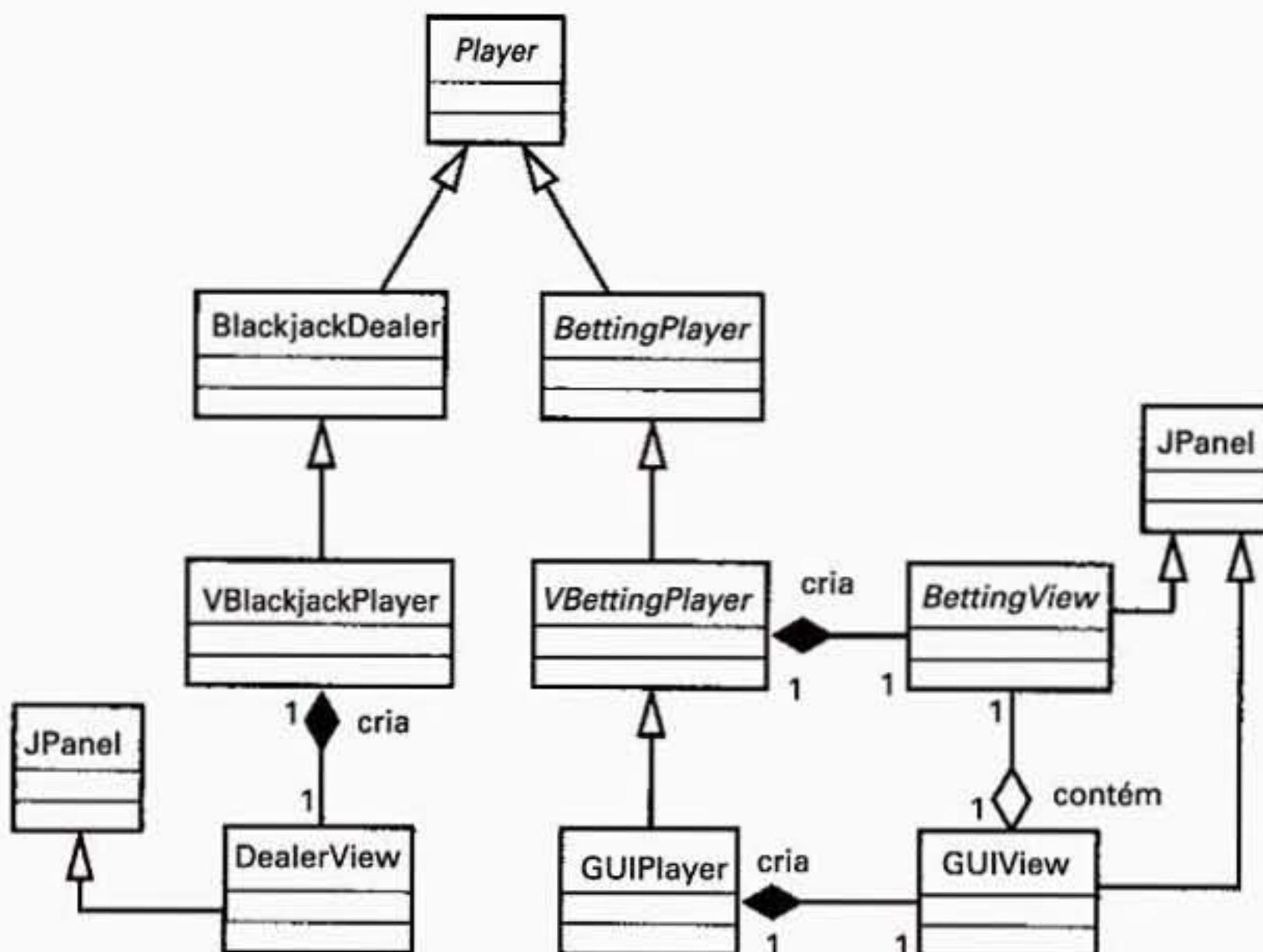
## Projetando os componentes da camada de abstração

Na seção anterior, você identificou Card, Hand e as diversas subclasses de Player necessárias para fornecer uma apresentação delas mesmas.

Para cada uma dessas classes, você precisa criar uma subclasse de abstração. Em particular, você precisa ter uma subclasse de BlackjackDealer, BettingPlayer, Hand e Card. Além disso, você precisa criar uma GUIPlayer, como no Capítulo 18. Entretanto, essa GUIPlayer também precisa fornecer uma apresentação.

A Figura 19.3 ilustra a hierarquia de herança Player resultante.

**FIGURA 19.3**  
A hierarquia de abstração Player (Jogador).

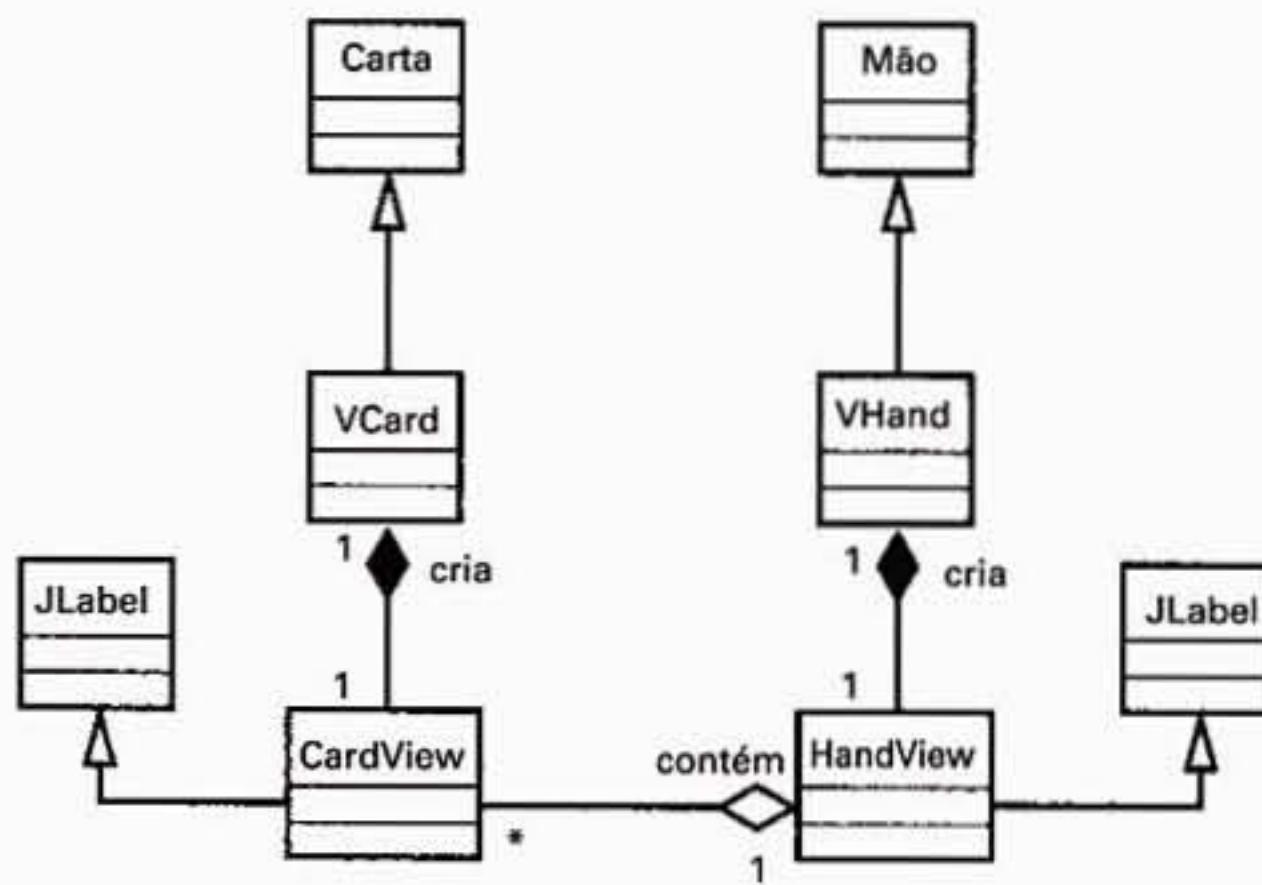


19

Você precisa criar uma subclasse **VBlackjackDealer** de **BlackjackDealer**. Você também precisa criar uma subclasse **VBettingPlayer** de **BettingPlayer**. Essas subclasses adicionarão suporte para criar e retornar objetos de apresentação.

A Figura 19.4 ilustra as hierarquias Hand e Card resultante.

**FIGURA 19.4**  
*A hierarquia de abstração Hand (Mão) e Card (Carta).*



Você precisará criar uma subclasse VCard Card, assim como uma subclasse VHand Hand. Essas subclasses visualizarão Card e Hand, respectivamente. Como no projeto de ontem, você também precisa de um VDeck. O VDeck criará um maço de objetos VCard.

## Projetando a camada de controle

O controle é uma classe relativamente simples. Uma instância do controle recuperará um objeto VBlackjackDealer, assim como os vários jogadores. De cada um desses objetos, o controle solicitará um objeto de apresentação. O controle pegará esse objeto de apresentação e o adicionará na exibição.

Você precisará projetar um mecanismo que o console possa usar para solicitar à camada de abstração a apresentação de seus objetos. A estratégia mais fácil é definir uma interface — vamos chamá-la de `Displayable`. `Displayable` tem um método: `public JComponent view()`, que recupera a apresentação de um objeto. Cada classe de abstração que fornece uma apresentação precisará implementar esse método.

As figuras 19.5 e 19.6 mostram as hierarquias atualizadas. Agora, as classes de abstração percebem a interface `Displayable`.

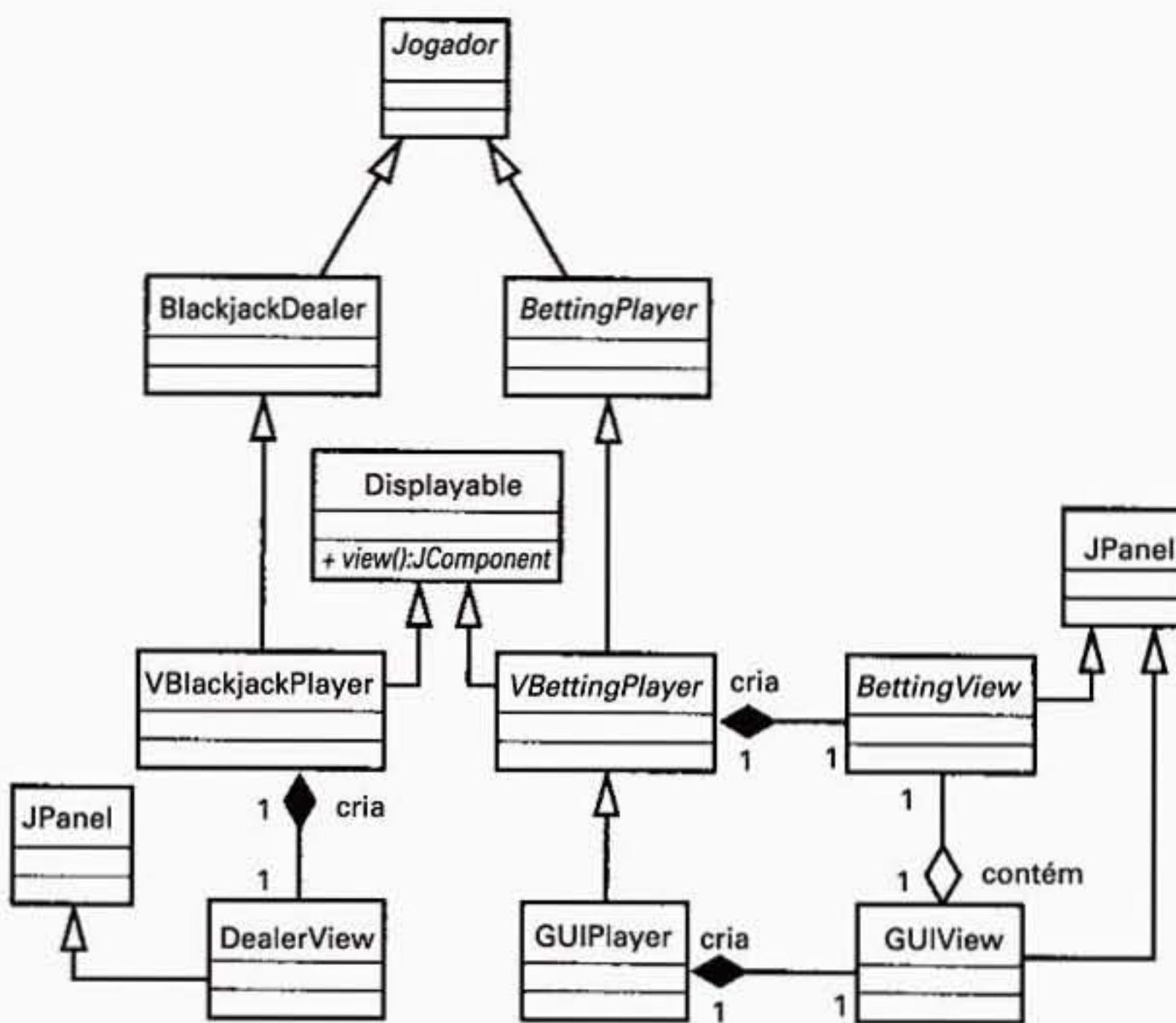
## Usando o padrão Factory para evitar erros comuns

Existe um pequeno problema na hierarquia resultante: nada o impede de criar um objeto Deckpile de objetos Card que não possam ser exibidos e passá-los para a banca. Os relacionamentos com capacidade de substituição permitem tal substituição. Infelizmente, você experimentará erros de tempo de execução, se misturar e combinar as classes GUI e não-GUI incorretamente.

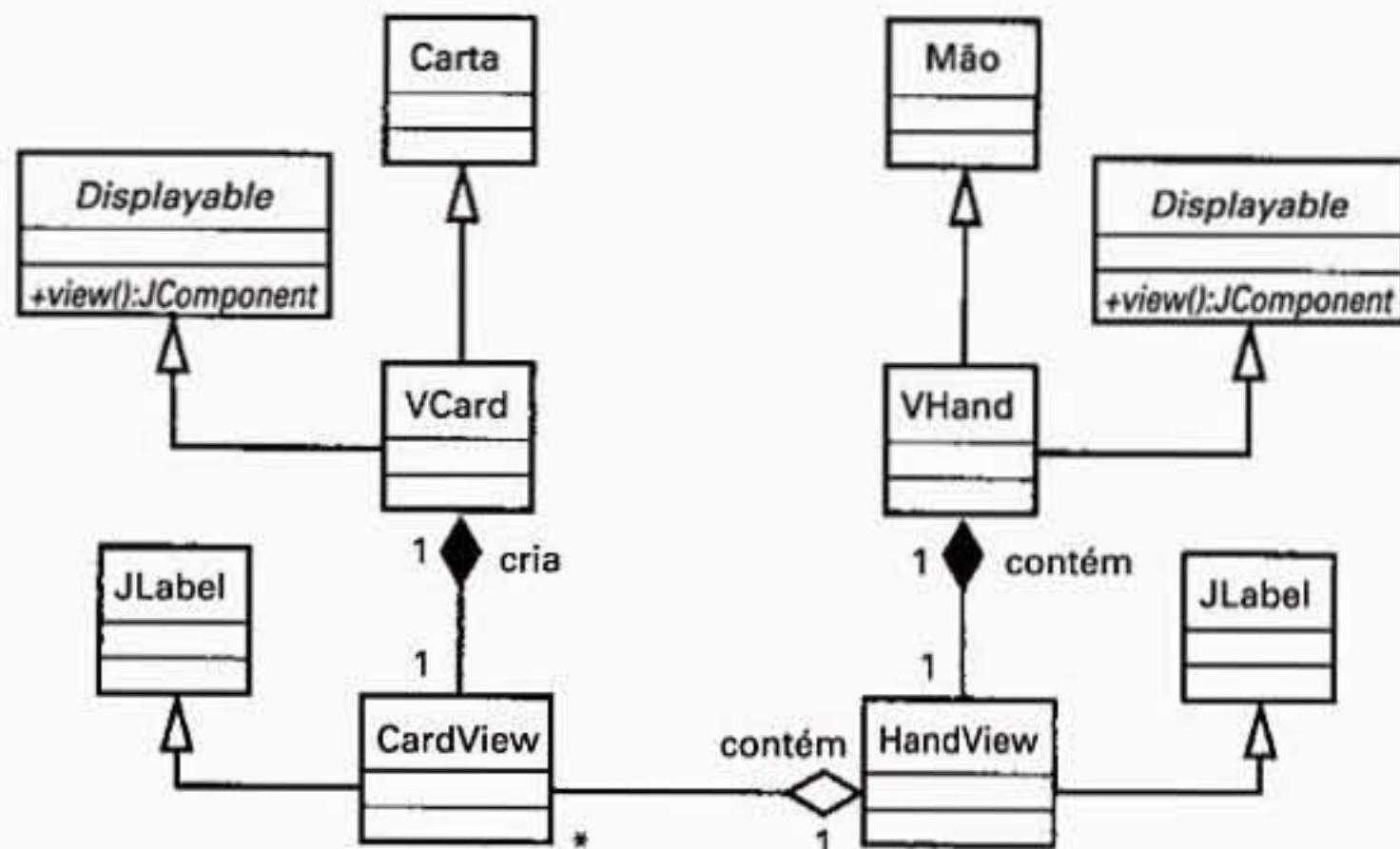
O Capítulo 12, “Padrões avançados de projeto”, apresentou o padrão de projeto `Abstract Factory`. Um motivo para usar esse padrão era para garantir que um conjunto de objetos fossem usados juntos. Isso o impede de usar objetos incompatíveis juntos.

**FIGURA 19.5**

A hierarquia de abstração Player atualizada.

**FIGURA 19.6**

A hierarquia de abstração Hand e Card atualizada.



19

Você pode usar factory para garantir que os objetos corretos sejam usados juntos. Você precisará criar factory que retorne um objeto **VBlackjackDealer** e um objeto **GUIPlayer** que tenham sido instanciados com os tipos corretos de argumentos. Quando o controle for recuperar os jogadores e a banca, ele deverá fazer isso apenas através de factory. Essa camada extra garantirá que todos os objetos sejam instanciados corretamente.

## Implementando a GUI PAC do jogo vinte-e-um

A lição de ontem mostrou que, ao se implementar uma GUI, freqüentemente é mais fácil trabalhar de cima para baixo. Seguindo essa recomendação, você deve implementar na seguinte ordem: VCard, VHand, VBettingPlayer, VBlackjackGUI e GUIPlayer. Vamos examinar cada implementação.

### Implementando VCard e VHand

VCard herda de Card e se representa através da classe interna: CardView. A Listagem 19.1 apresenta a implementação de VCard.

---

#### LISTAGEM 19.1 VCard.java

---

```
public class VCard extends Card implements Displayable {

    private String image;
    private CardView view;

    public Vcard( Suit suit, Rank rank, String image ) {
        super( suit, rank );
        this.image = image;
        view = new CardView( getImage() );
    }

    public void setFaceUp( boolean up ) {
        super.setFaceUp( up );
        view.changed();
    }

    public JComponent view() {
        return view;
    }

    private String getImage() {
        if( isFaceUp() ) {
            return image;
        } else {
            return "/bitmaps/empty_pile.xbm";
        }
    }

    private class CardView extends JLabel {
```

**LISTAGEM 19.1** VCard.java (*continuação*)

```
public CardView( String image ) {
    setImage( image );
    setBackground( Color.white );
    setOpaque( true );
}

public void changed() {
    setImage( getImage() );
}

private void setImage( String image ) {
    java.net.URL url = this.getClass().getResource(image);
    ImageIcon icon = new ImageIcon(url );
    setIcon( icon );
}
}
```

Essa implementação de VCard é muito parecida com aquela apresentada anteriormente, com exceção da classe de apresentação interna. Na criação, VCard cria e contém um modo de apresentação de si mesmo.

Você também notará que o novo atributo agora fica completamente encapsulado dentro de VCard. Para uma entidade externa exibir a imagem, ela precisa solicitar um modo de visualização a VCard. Além disso, quando a carta é virada, VCard diz automaticamente ao seu modo de apresentação para que se atualize, chamando `changed()` no modo de visualização. Ao contrário do padrão MVC, todo o controle é mantido dentro da própria abstração.

VHand é semelhante a VCard. Na criação, VHand cria um modo de apresentação de si mesmo. A Listagem 19.2 apresenta a implementação de VHand.

19

**LISTAGEM 19.2** VHand.java

```
public class VHand extends Hand implements Displayable {

    private HandView view = new HandView();

    public JComponent view() {
        return view;
    }

    // você precisa sobrepor addCard e reconfigurar para que quando a mão mudar, a
    // alteração se propague para o modo de visualização
```

**LISTAGEM 19.2 VHand.java (continuação)**

---

```
public void addCard( Card card ) {
    super.addCard( card );
    view.changed();
}

public void reset() {
    super.reset();
    view.changed();
}

private class HandView extends JPanel {
    public HandView() {
        super( new FlowLayout(FlowLayout.LEFT) );
        setBackground( new Color( 35, 142, 35 ) );
    }
    public void changed() {
        removeAll();
        Iterator i = getCards();
        while( i.hasNext() ) {
            VCard card = (Vcard) i.next();
            add( card.view() );
        }
        revalidate();
    }
}
```

---

Assim como VCard, VHand diz ao seu modo de visualização para que se atualize quando VHand mudar.

## Implementando VBettingPlayer

O conceito por trás de VBettingPlayer é o mesmo de VHand e VCard. A Listagem 19.3 apresenta a implementação de VBettingPlayer.

---

**LISTAGEM 19.3 VBettingPlayer.java**

---

```
public abstract class VBettingPlayer extends BettingPlayer implements
Displayable {

    private BettingView view;

    public VBettingPlayer( String name, VHand hand, Bank bank ) {
```

**LISTAGEM 19.3** VBettingPlayer.java (*continuação*)

```
super( name, hand, bank );
}

public JComponent view() {
    if(view == null ) {
        view = new BettingView( (VHand)getHand() );
        addListener( view );
    }
    return view;
}

private class BettingView extends JPanel implements PlayerListener {

    private TitledBorder border;

    public BettingView( VHand hand ) {
        super( new FlowLayout( FlowLayout.LEFT ) );
        buildGUI( hand.view() );
    }

    public void playerChanged( Player p ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name );
        repaint();
    }

    public void playerBusted( Player p ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " BUSTED!!" );
        repaint();
    }

    // o resto dos métodos PlayerListener foi cortado por brevidade
    // todos eles seguem o mesmo padrão; veja a listagem completa no
    // código-fonte

    private void buildGUI( JComponent hand ) {
        border = new TitledBorder( VBettingPlayer.this.getName() );
        setBorder( border );
        setBackground( new Color( 35, 142, 35 ) );
        border.setTitleColor( Color.black );
        add( hand );
    }
}
```

VBettingPlayer cria seu modo de visualização e o configura como um receptor. Quando o jogador muda, o modo de visualização sabe atualizar-se automaticamente. De interesse é o método `buildGUI()`. O método `buildGUI()` configura o modo de visualização.

Você notará que, em vez de pegar cada carta na mão e construir um modo de visualização, BettingView simplesmente pega o modo de visualização de VHand e o insere em si mesmo. VHand gerenciará a exibição das cartas. Tudo que BettingView tem de fazer é inserir o modo de visualização em si mesmo e manter o status do título atualizado.

## Implementando VBlackjackDealer

VBlackjackDealer funciona exatamente como VBettingPlayer. A Listagem 19.4 apresenta a implementação de VBlackjackDealer.

### LISTAGEM 19.4 VBlackjackDealer.java

---

```
public class VBlackjackDealer extends BlackjackDealer implements Displayable {  
  
    private DealerView view;  
  
    public VBlackjackDealer( String name, VHand hand, Deckpile cards ) {  
        super( name, hand, cards );  
    }  
  
    public JComponent view() {  
        if( view == null ) {  
            view = new DealerView( (Vhand) getHand() );  
            addListener( view );  
        }  
        return view;  
    }  
  
    private TitledBorder border;  
  
    public DealerView( VHand hand ) {  
        super( new FlowLayout( FlowLayout.LEFT ) );  
        String name = VBlackjackDealer.this.getName();  
        border = new TitledBorder( name );  
        setBorder( border );  
        setBackground( new Color(35, 142, 35) );  
        border.setTitleColor( Color.black );  
  
        add( hand.view() );  
        repaint();  
    }  
  
    public void playerChanged( Player p ) {
```

**LISTAGEM 19.4 VBlackjackDealer.java (continuação)**

```
String name = VBlackjackDealer.this.getName();
border.setTitle( name );
repaint();
}

public void playerBusted( Player p ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " BUSTED!!" );
    repaint();
}

// o resto dos métodos PlayerListener foi cortado por brevidade
// todos eles seguem o mesmo padrão; veja a listagem completa no
// código-fonte
}
}
```

DealerView recebe as alterações em VBlackjackDealer. À medida que essas alterações ocorrem, o modo de visualização atualiza seu título. VHand cuida de manter o modo de visualização da carta atualizado.

## Implementando GUIPlayer

Todos os aspectos não relativos a GUI da classe GUIPlayer são iguais àqueles apresentados ontem. Assim como as outras classes da camada de abstração, GUIPlayer define uma classe de apresentação interna.

Essa classe combina as classes OptionView e OptionViewController do Capítulo 18. O código de apresentação não é tão diferente das classes de modo de visualização e controle originais. Para ver a listagem completa, faça download do código-fonte do dia de hoje, no endereço [www.sams-publishing.com](http://www.sams-publishing.com).

19

## Reunindo tudo com o Controle

Antes de criar o controle, você precisa criar uma factory de jogador. A Listagem 19.5 apresenta a implementação de VPlayerFactory.

**LISTAGEM 19.5 VPlayerFactory.java**

```
public class VPlayerFactory {

    private VBlackjackDealer dealer;
    private GUIPlayer human;
    private Deckpile pile;
```

**LISTAGEM 19.5 VPlayerFactory.java (continuação)**

```
public VBlackjackDealer getDealer() {
    // cria e retorna apenas um
    if( dealer == null ) {
        VHand dealer_hand = getHand();
        Deckpile cards = getCards();
        dealer = new VBlackjackDealer( "Dealer", dealer_hand, cards );
    }
    return dealer;
}

public GUIPlayer getHuman() {
    // cria e retorna apenas um
    if( human == null ) {
        VHand human_hand = getHand();
        Bank bank = new Bank( 1000 );
        human = new GUIPlayer( "Human", human_hand, bank, getDealer() );
    }
    return human;
}

public Deckpile getCards() {
    // cria e retorna apenas um
    if( pile == null ) {
        pile = new Deckpile();
        for( int i = 0; i < 4; i ++ ) {
            pile.shuffle();
            Deck deck = new VDeck();
            deck.addToStack( pile );
            pile.shuffle();
        }
    }
    return pile;
}

private VHand getHand() {
    return new VHand();
}
```

---

VPlayerFactory garante que VBlackjackDealer e GUIPlayer sejam instanciados corretamente. A Listagem 19.6 apresenta o método `setup()` de BlackjackGUI: o controle.

**LISTAGEM 19.6** O método `setUp()` do controle BlackjackGUI

```
public class BlackjackGUI extends JFrame {  
  
    // CORTE!! parte do código omitido por brevidade  
  
    private JPanel players = new JPanel( new GridLayout( 0, 1 ) );  
  
    private void setUp() {  
        VBlackjackDealer dealer = factory.getDealer();  
  
        GUIPlayer human = factory.getHuman();  
  
        dealer.addPlayer( human );  
  
        players.add( dealer.view() );  
        players.add( human.view() );  
        getContentPane().add( players, BorderLayout.CENTER );  
    }  
}
```

`setUp()` simplesmente recupera cada jogador, adiciona seus modos de visualização em si mesmo e conecta a banca aos jogadores. Compare isso com a Listagem 19.7, o método `setUp()` de ontem.

**LISTAGEM 19.7** O método `setUp()` de MVC BlackjackGUI

```
private void setUp() {  
    BlackjackDealer dealer = getDealer();  
    PlayerView v1 = getPlayerView( dealer );  
  
    GUIPlayer human = getHuman();  
    PlayerView v2 = getPlayerView( human );  
  
    PlayerView [] views ={ v1, v2 };  
    addPlayers( views );  
  
    dealer.addPlayer( human );  
  
    addOptionView( human, dealer );  
}
```

19

Parece que simplesmente solicitar um modo de visualização da abstração é muito mais simples do que criar e juntar os vários modos de visualização da versão MVC.

## Resumo

Hoje, você viu uma alternativa ao MVC. Se seu sistema é relativamente estável e tem uma UI bem definida, a estratégia PAC pode oferecer uma solução mais elegante do que o MVC.

Mesmo que você precise suportar várias interfaces, a lição de hoje mostra como é possível usar herança para separar a camada de abstração da funcionalidade básica do sistema. Assim, para suportar várias interfaces, você precisa apenas criar uma subclasse para cada tipo de apresentação.

## Perguntas e respostas

**P Se o PAC oferece uma escolha melhor, por que fizemos a implementação MVC?**

**R** O PAC simplesmente oferece uma alternativa. Uma não é necessariamente melhor que a outra. Trata-se apenas de uma decisão de projeto.

O fato é que você encontrará o MVC em muitas implementações. Você pode encontrar o PAC, mas é muito menos provável. Assim, abordar o MVC primeiro é mais prático. Pessoalmente, tendemos a privilegiar o PAC.

Tenha as duas opções em mente. O que você nunca quer fazer é codificar sua lógica corporativa (classes como `BettingPlayer`) diretamente como um componente da GUI. Por exemplo, `BettingPlayer` *nunca* deve estender `JComponent` (ou algum outro componente da GUI) diretamente. O MVC e o PAC fornecem um mecanismo que evita misturar seu modelo e a GUI. Os padrões apenas adotam estratégias diferentes. O que você escolhe depende de seu projeto e de sua equipe de projeto.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Quais são as três camadas do padrão de projeto PAC?
2. Dê uma breve descrição de cada uma das camadas do PAC.
3. Como você pode usar herança para desacoplar a GUI das classes de sistema subjacentes?
4. Antes de usar o PAC, quais características seu projeto deve exibir?
5. Mesmo que você usasse PAC por toda esta lição, como ainda poderia fornecer uma UI de linha de comando para o sistema?
6. Como o padrão factory foi usado neste capítulo?

## Exercícios

1. Faça download do código-fonte da iteração de hoje. Quando você tiver o código, compile-o, faça-o funcionar executando BlackjackGUI e, depois, tente entender como ele funciona. Ter um entendimento total do código exigirá algum tempo e paciência.
2. O Exercício 2 do Capítulo 17 pediu para que você acrescentasse aposta em dobro no jogo vinte-e-um. Você precisa adicionar aposta em dobro no jogo novamente. Desta vez, adicione-a na versão gráfica do jogo que você carregou por download para o Exercício 1.

PÁGINA EM BRANCO

# SEMANA 3

DIA **20**

## Divertindo-se com o jogo vinte-e-um

Nos últimos dias, você trabalhou em um projeto de POO bastante intenso. Hoje, você vai dar o troco e se divertir com o jogo vinte-e-um, usando polimorfismo para adicionar vários jogadores não-humanos ao jogo. Você também verá como a OO se presta para fazer simuladores.

Hoje você aprenderá como pode:

- Usar polimorfismo para adicionar jogadores no jogo vinte-e-um
- Usar OO para criar simuladores

## Divertindo-se com o polimorfismo

O jogo vinte-e-um possibilita que até sete jogadores joguem em determinado momento. Até agora, o jogo que você criou incluía apenas um jogador humano e a banca. Felizmente, o polimorfismo permite que você adicione jogadores não-humanos no jogo.

### Criando um jogador

Para criar um novo jogador não-humano, basta criar uma nova classe que herde de `BettingPlayer`. Tudo que sua nova classe precisa fazer é implementar os dois métodos abstratos a seguir:

- `public boolean hit()`
- `public void bet()`

O comportamento que você fornecer para esses dois métodos determinará como o jogador jogará na sua vez. Você não precisará alterar quaisquer estados ou sobrepor quaisquer outros métodos. Os estados padrão sabem como usar os métodos que você implementa.

Quando você tiver acabado de definir a nova classe de jogador, pode alterar BlackjackGUI para criar o jogador e adicioná-lo no jogo.

## O jogador seguro

Vamos criar um novo jogador: SafePlayer. SafePlayer nunca recebe mais cartas e sempre aposta a menor quantia permitida. A Listagem 20.1 apresenta a definição de SafePlayer.

---

### **LISTAGEM 20.1** SafePlayer.java

---

```
public class SafePlayer extends BettingPlayer {

    public SafePlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit() {
        return false;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

---

Você notará que `hit()` sempre retorna falso; SafePlayer nunca receberá mais cartas. Do mesmo modo, SafePlayer sempre chama `place10Bet()`.

## Adicionando SafePlayer na GUI

Adicionar SafePlayer no jogo é relativamente simples. Primeiro, adicione o seguinte método em BlackjackGUI:

---

### **LISTAGEM 20.2** O método `getSafePlayer()`

---

```
private Player getSafePlayer() {
    // retorna o quanto for solicitado
    Hand safe_hand = new Hand();
    Bank safe_bank = new Bank( 1000 );
```

**LISTAGEM 20.2** O método `getSafePlayer()` (*continuação*)

```
return new SafePlayer( "Safe", safe_hand, safe_bank );  
}
```

`getSafePlayer()` é um método factory que instancia objetos `SafePlayer`. Quando você tiver o método, pode atualizar `setUp()` para que ele adicione o novo jogador no jogo. A Listagem 20.3 apresenta o método `setUp()` atualizado.

**LISTAGEM 20.3** O método `setUp()` atualizado

```
private void setUp() {  
    BlackjackDealer dealer = getDealer();  
    PlayerView v1 = getPlayerView( dealer );  
  
    GUIPlayer human = getHuman();  
    PlayerView v2 = getPlayerView( human );  
  
    Player safe = getSafePlayer();  
    PlayerView v3 = getPlayerView( safe );  
  
    PlayerView [] views = { v1, v2, v3 };  
    addPlayers( views );  
  
    dealer.addPlayer( human );  
    dealer.addPlayer( safe );  
  
    addOptionView( human, dealer );  
}
```

Você também desejará alterar o método principal da GUI, para que ele torne a janela um pouco maior, a fim de que o novo jogador possa caber. A Figura 20.1 ilustra o novo jogador, conforme ele aparece na GUI.

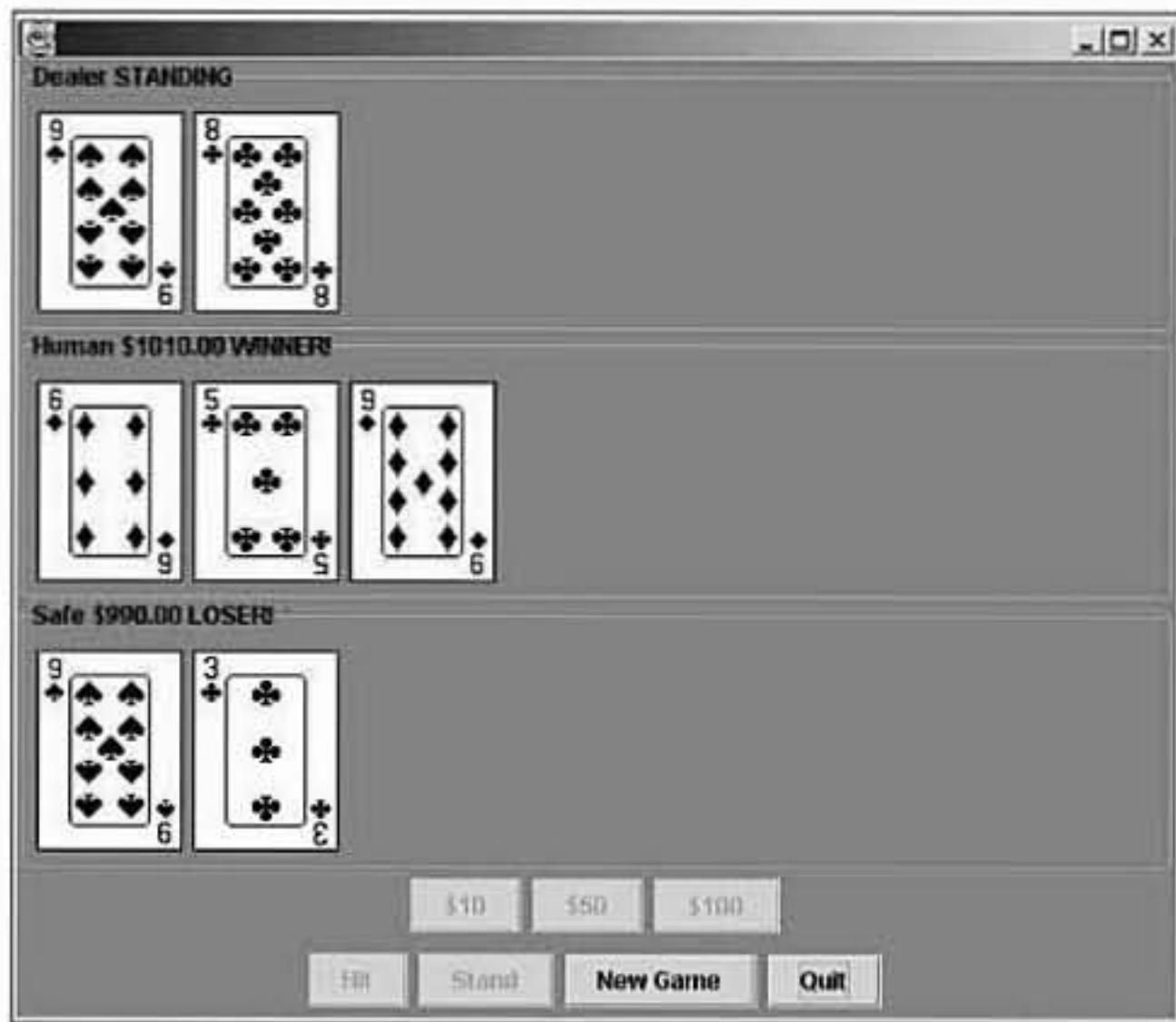
Aqui, o jogador é adicionado como um segundo jogador que joga após o ser humano. Nada o impede de deixar que o jogador não-humano jogue primeiro.

20

## Aperfeiçoamento

Você pode adicionar qualquer número e tipo de objeto `BettingPlayer` no jogo. Quando o número de opções de jogador ficar maior, talvez você queira fornecer ao usuário uma caixa de diálogo que permita configurar a mistura de jogadores. No mínimo, você desejará tornar o método `setUp()` protegido, para que subclasses possam sobrepor o método. Quando protegido, você pode escrever subclasses que criem jogos com diferentes misturas de jogadores.

**FIGURA 20.1**  
*A GUI contendo três jogadores.*



## POO e simulações

Conforme mencionado em uma lição anterior, quando você escreve um sistema de POO, na verdade está escrevendo um simulador vivo de algum problema real. Nesta semana, você escreveu um sistema que simula o jogo vinte-e-um.

Até aqui, o jogo tem sido centrado em um jogador humano. Você quer uma pessoa com a qual possa jogar o jogo, embora o sistema não se preocupe se há um ser humano jogando ou não. Para o sistema, todos os jogadores são objetos `Player`. O sistema cuida apenas que existam objetos `Player`.

Criando diferentes tipos de jogadores e adicionando-os ao sistema, você pode ter um jogo vinte-e-um que jogue sozinho, sem interação humana: um simulador do jogo vinte-e-um.

Um simulador do jogo vinte-e-um pode ser útil por vários motivos. Talvez você quisesse criar jogadores que utilizassem diferentes estratégias para ver quais delas funcionam melhor.

Talvez você esteja fazendo pesquisa de IA e queira escrever uma rede neural que aprenda a jogar perfeitamente um jogo vinte-e-um. Você pode usar um simulador do jogo vinte-e-um para atingir todos esses objetivos.

No capítulo e nos exercícios de hoje, você vai criar vários jogadores para saber se pode ganhar da banca com o passar do tempo.

## Os jogadores do jogo vinte-e-um

Você já viu um objeto `SafePlayer`. Será interessante ver como ele se desempenha com o passar do tempo. Além de `SafePlayer`, vamos definir:

- **FlipPlayer:** um jogador que alterna entre receber mais cartas e parar.
- **OneHitPlayer:** um jogador que sempre recebe mais cartas a cada vez.
- **SmartPlayer:** um jogador que pára com qualquer mão maior do que 11.

**NOTA**

Você pode notar a falta de casos de uso para esses jogadores. Como exercício, talvez você queira trabalhar nos casos de uso para os jogadores. Entretanto, um princípio deste livro tem sido realizar apenas a quantidade de análise que faça sentido e valorize seu entendimento do problema. Em nossa opinião, os casos de uso não ajudariam no seu entendimento neste caso e pareceria mais com a criação de documentação simplesmente.

## Implementando FlipPlayer

A implementação de `FlipPlayer` é um pouco mais complicada do que a de `SafePlayer`. A Listagem 20.4 apresenta a implementação de `FlipPlayer`.

### LISTAGEM 20.4 `FlipPlayer.java`

```
public class FlipPlayer extends BettingPlayer {

    private boolean hit = false;
    private boolean should_hit_once = false;

    public FlipPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit() {
        if( should_hit_once &&!hit ) {
            hit = true;
            return true;
        }
        return false;
    }

    public void reset() {
        super.reset();
        hit = false;
        should_hit_once = !should_hit_once;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

20

FlipPlayer precisa manter dois flags booleanos. Um flag informa ao jogador se ele deve receber mais cartas toda vez e o outro controla se o jogador recebeu mais cartas durante essa rodada. Os flags garantem que o jogador só receba mais cartas uma vez a cada outro jogo.

Para permitir que toda a lógica booleana funcione, você precisará sobrepor o método `reset()`, para alternar o estado booleano `should_hit_once`. Sobrepor `reset()` dessa maneira garante que o jogador só receba mais cartas a cada outro jogo.

## Implementando OneHitPlayer

OneHitPlayer é semelhante na implementação a FlipPlayer. Ao contrário de FlipPlayer, OneHitPlayer receberá mais cartas a cada jogo, mas apenas uma carta. A Listagem 20.5 apresenta a implementação de OneHitPlayer.

### **LISTAGEM 20.5** OneHitPlayer.java

---

```
public class OneHitPlayer extends BettingPlayer {

    private boolean has_hit = false;

    public OneHitPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit() {
        if( !has_hit ) {
            has_hit = true;
            return true;
        }
        return false;
    }

    public void reset() {
        super.reset();
        has_hit = false;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

---

Novamente, você precisa sobrepor `reset()` para que ele limpe o flag `has_hit`. Esse flag garante que o jogador só receba mais uma carta na rodada.

## Implementando SmartPlayer

SmartPlayer tem uma implementação muito simples. A Listagem 20.6 apresenta a implementação.

### LISTAGEM 20.6 SmartPlayer.java

```
public class SmartPlayer extends BettingPlayer {  
  
    public SmartPlayer( String name, Hand hand, Bank bank ) {  
        super( name, hand, bank );  
    }  
  
    public boolean hit() {  
        if( getHand().total() > 11 ) {  
            return false;  
        }  
        return true;  
    }  
  
    public void bet() {  
        getBank().place10Bet();  
    }  
}
```

SmartPlayer verifica o total de seu objeto Hand. Se o total for maior que onze, o jogador pára. Se for menor, o jogador recebe mais cartas.

## Configurando o simulador

Para transformar o jogo em um simulador, você pode simplesmente alterar o método principal encontrado na classe Blackjack. Aqui, copiaremos e renomearemos a classe como BlackjackSim. A Listagem 20.7 apresenta o novo simulador.

### LISTAGEM 20.7 BlackjackSim.java

```
public class BlackjackSim {  
  
    public static void main( String [] args ) {  
  
        Console.INSTANCE.printMessage( "How many times should the simulator  
play?" );  
        String response = Console.INSTANCE.readInput( "invalid" );  
        int loops = Integer.parseInt( response );  
  
        Deckpile cards = new Deckpile();
```

**LISTAGEM 20.7** BlackjackSim.java (*continuação*)

```
for( int i = 0; i < 4; i ++ ) {
    cards.shuffle();
    Deck deck = new Deck();
    deck.addToStack( cards );
    cards.shuffle();
}

// cria uma banca
Hand dealer_hand = new Hand();
BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand,
cards );

// cria um OneHitPlayer
Bank one_bank = new Bank( 1000 );
Hand one_hand = new Hand();
Player oplayer = new OneHitPlayer( "OneHit", one_hand, one_bank );

// cria um SmartPlayer
Bank smart_bank = new Bank( 1000 );
Hand smart_hand = new Hand();
Player smplayer = new SmartPlayer( "Smart", smart_hand, smart_bank );

// cria um SafePlayer
Bank safe_bank = new Bank( 1000 );
Hand safe_hand = new Hand();
Player splayer = new SafePlayer( "Safe", safe_hand, safe_bank );

// cria um FlipPlayer
Bank flip_bank = new Bank( 1000 );
Hand flip_hand = new Hand();
Player fplayer = new FlipPlayer( "Flip", flip_hand, flip_bank );

// reúne todos os jogadores
dealer.addListener( Console.INSTANCE );
oplayer.addListener( Console.INSTANCE );
dealer.addPlayer( oplayer );
splayer.addListener( Console.INSTANCE );
dealer.addPlayer( splayer );

smplayer.addListener( Console.INSTANCE );
dealer.addPlayer( smplayer );
fplayer.addListener( Console.INSTANCE );
dealer.addPlayer( fplayer );

int counter = 0;
```

**LISTAGEM 20.7** BlackjackSim.java (*continuação*)

```
    while( counter < loops ) {  
        dealer.newGame();  
        counter++;  
    }  
}
```

O simulador primeiro o consultará quanto ao número de vezes a jogar. Quando ele tiver essa informação, criará um objeto `BlackjackDealer`, um `OneHitPlayer`, um `SmartPlayer` e um `FlipPlayer`. Então, ele conectará esses jogadores no Console e os adicionará na banca.

Quando tiver terminado a configuração, ele jogará o jogo pelo número de vezes que você tiver especificado.

## Os resultados

Após executar o simulador algumas vezes, com 1000 jogos por execução, torna-se fácil ver como os jogadores se comportam. Aqui estão os resultados obtidos, do maior para o menor:

- `SmartPlayer`
- `SafePlayer`
- `FlipPlayer`
- `OneHitPlayer`

Parece que, desses quatro jogadores, permanecer com uma mão que é maior que 11 é a melhor estratégia. Na verdade, `SmartPlayer` foi o único jogador a ficar com dinheiro após os 1000 jogos.

`SafePlayer` vem em um segundo lugar próximo, mas acabou perdendo dinheiro. E, faça o que fizer, não receba mais cartas. Esses jogadores perderam mais dinheiro do que tinham em seu pote inicial.



Nenhum desses jogadores acabou com mais dinheiro do que começou. Os jogadores só diferiram na velocidade com que perderam seu dinheiro.

20

## Resumo

Você aprendeu uma lição importante hoje: não siga nenhuma das estratégias de hoje ao jogar vinte-e-um. Você perderá todo seu dinheiro!

Você também viu pela primeira vez como o polimorfismo permite escrever software à prova do futuro. Você pode introduzir tipos de jogador no sistema, sem ter de alterar o sistema básico. Esses tipos de jogador nem mesmo eram considerados, quando você construiu o sistema inicial.

## Perguntas e respostas

**P Há um motivo pelo qual você não tenha usado a GUI como base de seu simulador?**

**R** Poderíamos ter usado a GUI, em vez do console. Poderia ter sido interessante ver os jogos cintilarem. Normalmente, um simulador não tem UI. Em vez disso, o simulador mostrará algumas estatísticas no final.

Também existem limites práticos para uma GUI. As GUIs demoram algum tempo para atualizar. Jogar 1000 jogos visuais poderia demorar um pouco mais do que jogá-los na linha de comando.

Várias versões de Swing também sofrem de estouro de memória. Esses estouros poderiam surgir e atrapalhá-lo, caso você executasse 10.000 jogos visuais.

Para propósitos de teste, entretanto, você poderia considerar o uso da GUI como base para o simulador.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Como o polimorfismo permitiu jogar o jogo sem um jogador humano?
2. Qual estratégia de aposta você nunca simulará?

## Exercícios

1. Faça download do código-fonte da iteração de hoje, a partir do endereço [www.samspublishing.com](http://www.samspublishing.com). O código está dividido em quatro diretórios separados: `gui`, `simulation`, `exercise_2` e `exercise_3`.

`gui` contém uma GUI que tem um jogador humano e um `SafePlayerU`.

`simulation` contém o código de simulação. `simulation` tem os seguintes jogadores: `OneHitPlayer`, `FlipPlayer`, `SafePlayer` e `SmartPlayer`.

`exercise_2` e `exercise_3` contêm os arquivos que você precisará para os exercícios 2 e 3, assim como as soluções.

Estude o código de gui e simulator. Tente entender como tudo funciona. Quando você tiver feito isso, complete os exercícios 2 e 3.

2. O download contém todos os arquivos iniciais que você precisará para completar este exercício. Os arquivos iniciais fazem alterações no método `hit` declarado pela primeira vez em `Player`.

As alterações adicionam `Dealer` como um parâmetro. O método `getUpCard` foi adicionado em `Dealer` para que você possa obter a carta aberta de `Dealer`. Em um jogo vinte-e-um real, os jogadores podem ver a carta aberta da banca. Você pode usar essa informação para tornar alguns movimentos mais inteligentes.

Para este exercício, escreva um ou dois jogadores novos, que baseiem sua decisão de receber mais cartas em seu próprio total, assim como na carta aberta da banca. Aqui estão duas sugestões, mas sinta-se livre para implementar seus próprios jogadores, adicioná-los no simulador e ver como eles reagem:

### **KnowledgeablePlayer**

`KnowledgeablePlayer` deve basear a decisão de se vai ou não receber mais cartas de acordo com as seguintes regras:

Independentemente de tudo, se o total da mão for maior que 15, pára.

Se o total da mão for 11 ou menor, recebe mais cartas.

Se a mão for 15 ou menos e maior que 11, baseia a decisão de receber mais cartas na carta da banca. Se a carta da banca for maior que 7, recebe mais cartas; caso contrário, você deve parar.

### **OptimalPlayer**

`OptimalPlayer` está tão próximo da perfeição, que você pode não conseguir diferenciar entre mãos fáceis e difíceis. (A diferenciação é deixada como exercício para o leitor). `OptimalPlayer` deve basear a decisão de se vai ou não receber mais cartas nas seguintes regras:

Se o total da mão for maior ou igual a 17, pára.

Se o total da mão for 11 ou menos, recebe mais cartas.

Se o total da mão for 16, baseia a decisão de receber mais cartas ou parar na carta aberta da banca. Se a carta aberta for 7, 8 ou 9, recebe mais cartas; caso contrário, pára.

Se o total da mão for 13, 14 ou 15, baseia a decisão de receber mais cartas ou parar na carta aberta da banca. Se a carta aberta for 2, 3, 4, 5 ou 6, pára; caso contrário, recebe mais cartas.

Se o total da mão for 12, baseia a decisão de receber mais cartas ou parar na carta aberta da banca. Se a carta aberta for 4, 5 ou 6, pára; caso contrário, recebe mais cartas.

3. O download contém todos os arquivos iniciais necessários para você completar este exercício. Os arquivos iniciais adicionam suporte para dobrar a aposta nas classes. Além disso, o novo método `doubleDown` agora aceita um objeto `Dealer` como argumento.

As alterações adicionam `Dealer` como um parâmetro em `doubleDown`. O método `getUpCard` foi adicionado em `Dealer` para que você possa obter a carta aberta de `Dealer`. Em um jogo vinte-e-um real, o jogador pode ver a carta aberta da banca. Você pode usar essa informação para tornar alguns movimentos mais inteligentes.

Para este exercício, escreva um ou dois jogadores novos que baseiem sua decisão de receber mais cartas ou dobrar em seu próprio total, assim como na carta aberta da banca. Aqui estão duas sugestões, mas sinta-se livre para implementar seus próprios jogadores, adicioná-los no simulador e ver como eles se comportam:

#### **KnowledgeablePlayer**

Para receber mais cartas, siga as regras delineadas no Exercício 2.

Para dobrar, siga estas regras:

Se o total da mão for 10 ou 11, dobra. Em todos os outros casos, não dobra.

#### **OptimalPlayer**

Para receber mais cartas, siga as regras delineadas no Exercício 2.

Para dobrar, siga estas regras:

Se o total da mão for 11, sempre dobra.

Se o total da mão for 10, baseia a decisão de dobrar na carta aberta da banca. Se a carta aberta for 10 ou um ás, dobra; caso contrário, não dobra.

Se o total da mão for 9, baseia a decisão de dobrar na carta aberta da banca. Se a carta aberta for 2, 3, 4, 5 ou 6, dobra; caso contrário, não dobra.

Em todos os outros casos, não dobra.

# SEMANA 3

DIA **21**

## O último quilômetro

Parabéns! Você chegou à última lição deste livro. Você percorreu um longo caminho. Agora, você deve ter a base necessária para ter êxito ao continuar seus estudos de POO.

Hoje, eliminaremos algumas arestas e depois o enviaremos por seu caminho!

Hoje você aprenderá sobre:

- Refazer o projeto do jogo vinte-e-um para reutilizar em outros sistemas
- As vantagens que a POO trouxe para o jogo vinte-e-um
- Realidades do setor que podem impedir soluções totais OO

## Amarrando as pontas

Você abordou muitos assuntos durante as três últimas semanas. Você começou considerando a teoria básica da OO e chegou em um projeto inteiramente baseado em POO. Agora, você já deve ter uma boa idéia do que POO realmente significa.

Antes de concluirmos o livro, entretanto, restam três problemas para abordar:

- Refazer o projeto do jogo vinte-e-um para reutilizar em outros sistemas
- Um levantamento das vantagens que a POO trouxe para o sistema de jogo vinte-e-um
- Uma palavra sobre as realidades do setor e POO

## Refazendo o projeto do jogo vinte-e-um para reutilização em outros sistemas

Existe um pequeno problema relacionado ao projeto do jogo vinte-e-um que precisamos examinar. Esse problema não têm impacto sobre o jogo vinte-e-um, mas poderia ter impacto negativo em outro sistema OO, se ele reutilizar o projeto incorretamente.

No Dia 15, você viu duas soluções alternativas. Em uma solução, a banca faz um laço por seus jogadores dizendo a cada um para que jogue depois que o jogador anterior terminar. Então, você viu uma outra estratégia mais baseada em objetos para o laço do jogo.

Em vez da banca percorrer os jogadores um por um e dizer a eles o que fazer, a banca OO iniciava cada jogador e esperava que ele dissesse quando tinha acabado de jogar.

Isso ofereceu uma solução mais limpa, pois você deixava por conta do jogador dizer à banca quando tivesse terminado — somente então é que a banca continuava. Também se viu que esse projeto era absolutamente necessário para a GUI funcionar; caso contrário, um jogador humano retornaria instantaneamente e, se a banca estivesse fazendo o laço, diria ao próximo jogador para prosseguir. O jogador humano nunca teria sua vez na estratégia procedural!

### O problema do projeto

Existe um pequeno problema na estratégia delineada nas lições. Vamos acompanhar as chamadas de método em um jogo onde existe um jogador e uma banca. Os dois jogadores param durante sua vez.

A Listagem 21.1 representa um acompanhamento de todas as chamadas de método no jogo. A pilha termina quando um método retorna.

---

#### LISTAGEM 21.1 Um acompanhamento da pilha de método

---

```
BlackjackSim.main
  BlackjackDealer.newGame
    Player.play
      BlackjackDealer$DealerCollectingBets.execute
        Player.play
          BettingPlayer$Betting.execute
            BlackjackDealer.doneBetting
              Player.play
                BlackjackDealer$DealerCollectingBets.execute
                  BlackjackDealer$DealerDealing.execute
                    BlackjackDealer$DealerWaiting.execute
                      Player.play
                        Player$Playing.execute
                          Player$Standing.execute
                            BlackjackDealer.standing
```

**LISTAGEM 21.1** Um acompanhamento da pilha de método (*continuação*)

```
Player.play
BlackjackDealer$DealerWaiting.execute
Player$Playing.execute
BlackjackDealer$DealerStanding
```

O problema é sutil. Nenhum dos métodos retorna até que a banca termine sua vez! Os métodos chamam-se uns aos outros recursivamente. Assim, por exemplo, o método `notifyChanged` da Listagem 21.2 não será executado até que o jogo corrente termine.

**LISTAGEM 21.2** Um método que não será chamado antes que o jogo corrente termine

```
public void execute( Dealer dealer ) {
    if( hit( dealer ) ) {
        dealer.hit( Player.this );
    } else {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    current_state.execute( dealer );
    // transição

    // não será chamado até que a pilha se desenrole!!!!
    notifyChanged();
}
```

No jogo vinte-e-um, isso não é problema, pois existe um limite de sete jogadores e a pilha de chamada de métodos se desenrola após cada jogo. Você também pode codificar cuidadosamente no caso de problemas como o demonstrado na Listagem 21.2.

Entretanto, imagine um simulador com centenas ou milhares de objetos que seguem o projeto do jogo vinte-e-um. Se esses objetos chamarem um ao outro recursivamente, mesmo que a pilha finalmente se desenrole, você poderia acabar ficando sem memória. De qualquer modo, cada chamada de método alocará mais memória. Se você seguir esse projeto inalterado, seu sistema não funcionará ou exigirá muito mais memória do que o absolutamente necessário.

## Costurando uma solução com linhas de execução (threads)

Felizmente, existe uma solução: as linhas de execução (threads). Embora uma discussão completa sobre o uso de linhas de execução esteja bem fora dos objetivos deste livro, você vai ver como pode usá-las para resolver o problema da chamada de métodos rapidamente.

**NOTA**

Quase todo sistema não-trivial compartilhará duas características:

- Eles usarão linhas de execução
- Eles terão lógica de estado

O sistema do jogo vinte-e-um compartilha essas duas características.

Uma linha de execução é simplesmente um caminho de execução através de seu programa. Até agora, o sistema do jogo vinte-e-um tem uma única linha de execução. A Figura 21.1 ajuda a visualizar essa linha de execução única.

**FIGURA 21.1**

*O sistema do jogo vinte-e-um com uma única linha de execução.*

Sistema do jogo vinte-e-um com uma única linha de execução



Como o jogo vinte-e-um usa apenas uma linha de execução (ele tem somente uma linha de execução), essa linha de execução faz tudo.

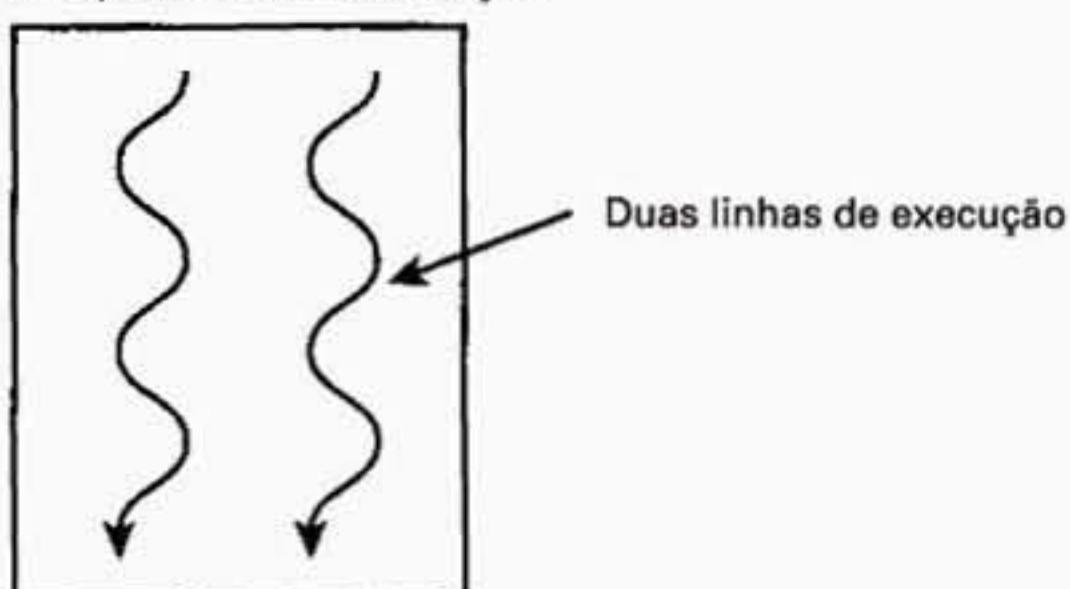
O uso de linhas de execução permite que você crie várias linhas de execução através de seu programa. Criando várias linhas de execução, seu programa pode fazer muitas coisas diferentes, simultaneamente. A Figura 21.2 ajuda a visualizar duas linhas de execução percorrendo o sistema do jogo vinte-e-um.

Usar linhas de execução no sistema do jogo vinte-e-um pode permitir que um método retorne imediatamente. Vamos ver um exemplo simples de linha de execução da linguagem Java. A Listagem 21.3 apresenta uma linha de execução simples que imprime “Hello World!”.

**FIGURA 21.2**

*O sistema do jogo vinte-e-um com múltiplas linhas de execução.*

Sistema do jogo vinte-e-um com múltiplas linhas de execução



**LISTAGEM 21.3** "Hello World!" com múltiplas linhas de execução

```
public class HelloWorld {  
  
    public void sayHello() {  
        System.out.println( "Hello World!" );  
    }  
  
    public static void main( String [] args ) {  
        final HelloWorld hw = new HelloWorld();  
  
        Runnable runnable = new Runnable() {  
            public void run() {  
                hw.sayHello();  
            }  
        };  
  
        Thread thread = new Thread( runnable );  
        thread.start();  
  
        System.out.println( "All Done!" );  
    }  
}
```

HelloWorld é ela própria uma classe simples que tem um método: sayHello. sayHello imprime uma mensagem na linha de comando.

É no método main que as coisas ficam interessantes. Primeiro, esse método instancia HelloWorld. Então, ele cria uma classe Runnable anônima. Os objetos Runnable têm um método: run. O método run diz à linha de execução o que fazer quando for iniciada. Neste caso, ele dirá à instância de HelloWorld para que imprima sua mensagem.

Após criar Runnable, o método principal instancia um Thread Java. Quando você cria um objeto Thread, precisa passá-lo para um objeto Runnable. O método run de Runnable diz ao objeto Thread o quê fazer quando você disser a Thread para que comece (start). Após iniciar o objeto Thread, o método main imprime sua própria mensagem.

Você pode ficar surpreso quando ver o método principal ser executado. A Figura 21.3 apresenta a saída de HelloWorld.

Ao executar HelloWorld, você verá que "All Done" é impresso antes de "Hello World!" a chamada de Thread.start não bloqueia como outras chamadas de método. Como start inicia uma nova linha de execução, ele retorna automaticamente. Após chamar start, você tem duas linhas de execução no programa HelloWorld. Apenas acontece que o método main imprime sua mensagem antes que a nova linha de execução tenha uma chance de chamar sayHello.

**FIGURA 21.3**  
*A saída de HelloWorld.*



Você pode usar o fato de que start não bloqueia para corrigir a deficiência do projeto no jogo vinte-e-um. A Listagem 21.4 apresenta um novo estado Waiting para BlackjackDealer, que inicia cada jogador em sua própria linha de execução.

#### **LISTAGEM 21.4** DealerWaiting com linha de execução

---

```
private class DealerWaiting implements PlayerState {
    public void handChanged() {
        // impossível no estado de espera
    }
    public void handPlayable() {
        // impossível no estado de espera
    }
    public void handBlackjack() {
        // impossível no estado de espera
    }
    public void handBusted() {
        // impossível no estado de espera
    }
    public void execute( final Dealer dealer ) {
        if( !waiting_players.isEmpty() ) {
            final Player player = (Player) waiting_players.get( 0 );
            waiting_players.remove( player );
            Runnable runnable = new Runnable() {
                public void run() {
                    player.play( dealer );
                }
            };
            Thread thread = new Thread( runnable );
            thread.start();
        } else {
            setCurrentState( getPlayingState() );
            exposeHand();
            getCurrentState().execute( dealer );
            // faz a transição e executa
        }
    }
}
```

**LISTAGEM 21.4** DealerWaiting com linha de execução (*continuação*)

```
}
```

```
}
```

```
}
```

Iniciando cada jogador em sua própria linha de execução, o método de estado execute de BlackjackDealer pode retornar imediatamente, desenrolando assim a pilha. Isso interpõe algumas dificuldades, se você fizer um laço com as chamadas de newGame, pois agora newGame retornará antes que o jogo tenha realmente terminado. Se você fizer o laço, iniciará outro jogo antes que o último tenha terminado e, então, enfrentará todos os tipos de problemas desagradáveis. Você pode resolver esse problema dizendo a BlackjackDealer quantas vezes deve fazer o laço. No final de cada jogo, ele pode verificar se precisa jogar novamente.

**NOTA**

O uso de linhas de execução é apenas uma maneira de resolver o problema da recursividade. Apresentamos uma solução com linhas de execução aqui para que você as visse um pouco.

O problema do laço/linha de execução levanta algumas preocupações. Você também poderia criar um objeto GameTable que iniciasse e parasse as linhas de execução. Então, Dealer poderia receber o estado da tabela e distribuir cartas, receber mais cartas, efetuar o jogo, etc., com base no estado. Entretanto, tal estratégia é um pouco mais complicada do que simplesmente usar linhas de execução para os jogadores quando eles começam.

Você também poderia se desfazer da recursividade através de iteração pelos jogadores.

A boa nova é que, se você não fizer o laço, como na GUI, pode usar linha de execução facilmente, simplesmente mudando o estado de DealerWaiting de BlackjackDealer! O código-fonte carregado por download contém versões com linhas de execução da GUI.

**ALERTA**

É fácil usar linhas de execução no jogo vinte-e-um, pois apenas uma linha de execução de jogador é executada em dado momento. Você não tem muitas linhas de execução de jogador sendo executadas concomitantemente.

O uso de linhas de execução se torna complicado quando muitas linhas de execução são executadas concomitantemente e compartilham os mesmos dados!

## Identificando as vantagens que a POO trouxe para o sistema do jogo vinte-e-um

A primeira semana mostrou alguns dos objetivos e vantagens da POO. Para recapitular, a POO tenta produzir software que é:

21

1. Natural
2. Confiável

3. Reutilizável
4. Manutenível
5. Extensível
6. Oportuno

A POO trouxe cada uma dessas vantagens para o sistema do jogo vinte-e-um. O sistema do jogo vinte-e-um atinge cada um dos objetivos da POO:

- Natural: o sistema do jogo vinte-e-um modela naturalmente um jogo vinte-e-um.  
O sistema do jogo vinte-e-um existe nos termos de um jogo vinte-e-um real. O sistema do jogo vinte-e-um é constituído de objetos Player, um BlackjackDealer, objetos Card, Deck e um DeckPile. Como você vê, o jogo vinte-e-um é uma simulação viva do domínio do jogo vinte-e-um.
- Confiável: o sistema do jogo vinte-e-um é confiável.  
Através de uma combinação de testes cuidadosos e encapsulamento, você criou um sistema do jogo vinte-e-um confiável. Como você isolou conhecimentos e responsabilidades e os colocou no lugar onde pertencem, pode fazer melhorias no sistema sem se preocupar com um impacto negativo em partes não relacionadas do sistema.
- Reutilizável: o sistema do jogo vinte-e-um é reutilizável.  
Como esse foi o primeiro jogo de cartas que você escreveu, não houve muita ênfase na escrita de uma estrutura de jogo de cartas abstrata. Em vez disso, você escreveu um jogo vinte-e-um. Como resultado, o jogo não é completamente reutilizável; entretanto, classes como Card, Deck e Deckpile podem ser reutilizadas em praticamente qualquer jogo de cartas.  
Além disso, muitas das idéias do projeto são reutilizáveis em muitos problemas. À medida que você escrever mais jogos de cartas, poderá abstrair mais e criar uma estrutura totalmente reutilizável.
- Manutenível: o sistema do jogo vinte-e-um é manutenível.  
Encapsulando conhecimentos e responsabilidades no lugar onde eles pertencem, fica simples fazer alterações em uma parte do sistema sem ter um impacto negativo em outras partes não relacionadas do sistema.  
Tais divisões tornam possível fazer melhorias no sistema a qualquer momento. Você também viu pela primeira vez como a herança e o polimorfismo tornam possível adicionar novos jogadores no sistema, a qualquer momento.
- Extensível: o sistema do jogo vinte-e-um é extensível.  
Você viu pela primeira vez como pode adicionar novos jogadores no sistema. Além disso, através de uma herança cuidadosa, você pode introduzir novos tipos de cartas (como cartas visuais) e mãos. O processo iterativo mostrou como um sistema de POO pode ser extensível.

- Oportuno: o sistema do jogo vinte-e-um é oportuno.  
Você produziu um jogo vinte-e-um completo em quatro iterações — o tempo de uma semana. Veja como ele é oportuno!

## Realidades do setor e POO

As lições deste livro tiveram como pressuposto o fato de que você estava iniciando seus projetos de POO desde o princípio. Quando você começa desde o princípio, não precisa integrar sistemas de backend legados, não-OO. Você não precisa reutilizar bibliotecas procedurais. Você pode começar do começo e tudo que usar pode ser OO.

Você verá que um projeto de POO independente é raro. Na maioria das vezes, você precisará interagir com componentes não-OO. Pegue o caso dos bancos de dados relacionais. Os bancos de dados relacionais não são particularmente orientados a objetos e os bancos de dados orientados a objetos ainda são raramente usados fora de alguns nichos do setor.

A própria linguagem Java não é totalmente orientada a objetos. A confiança em primitivas não-OO faz você realizar alguma codificação não-OO, de tempos em tempos.

Ao se deparar com essas realidades, é melhor trabalhar para empacotar esses aspectos não-OO em um empacotador orientado a objetos. Por exemplo, ao tratar com bancos de dados relacionais, ajuda escrever uma camada de persistência de objetos. Em vez de ir diretamente a um banco de dados e reconstituir seus objetos através de várias consultas SQL, a camada de persistência pode fazer esse trabalho para você.

Realmente não é possível abordar aqui cada tipo de sistema não-OO que você vai encontrar. Mas teríamos sido negligentes se não apontássemos essas realidades antes de mandá-lo aplicar POO.

Vai demorar muito tempo, antes que todo sistema legado seja convertido para uma arquitetura baseada em objetos (se é que isso vai acontecer). Você deve estar preparado para essa eventualidade e pronto para tratar com ela da forma mais elegante possível.

## Resumo

Você terminou! Em três semanas curtas, este livro forneceu a você uma base sólida em POO. O resto fica por sua conta. Agora você tem conhecimento suficiente para começar a aplicar POO em seus projetos diários. Boa sorte!

## Perguntas e respostas

21

- P Por que você esperou até agora para nos dizer a respeito de linhas de execução?**
- R** O problema do projeto não afeta o jogo vinte-e-um. Apresentar os possíveis problemas antecipadamente teria confundido a questão.

É importante que você perceba as deficiências do projeto, assim como uma possível solução.

O uso de linhas de execução também é um assunto avançado. Usar linhas de execução no jogo vinte-e-um foi muito fácil. Mas usá-las em outros aplicativos pode não se mostrar tão simples.

**P O que pode tornar o uso de linhas de execução difícil?**

**R** Se você tiver múltiplas linhas de execução compartilhando dados, uma linha de execução poderá alterar os dados e danificar outra linha de execução. Tais problemas de concorrência são extremamente difíceis de projetar, implementar e depurar.

## Workshop

As perguntas e respostas do teste são fornecidas para seu melhor entendimento. Veja as respostas no Apêndice A, “Respostas”.

## Teste

1. Como as linhas de execução tratam do problema da chamada de método recursiva?

## Exercícios

1. Faça download do código-fonte da iteração de hoje. O código está dividido em quatro diretórios separados: `threaded_hello_world`, `threaded_mvc_gui`, `threaded_pac_gui` e `threaded_simulador`. Estude o código e certifique-se de entender como ele funciona.
2. Seu estudo de POO não deve terminar com este livro. Gere uma lista de assuntos sobre os quais você gostaria de aprender mais. Classifique esses assuntos em ordem de importância, pesquise a Web para encontrar materiais e comece a estudar!

# SEMANA 3

## Em revisão

Agora, você concluiu a terceira e última semana deste livro e nos últimos sete dias, aprendeu a desenvolver seu próprio jogo vinte-e-um OO.

O Dia 15 apresentou as regras básicas do jogo vinte-e-um. Você desenvolveu uma lista de casos de uso em potencial e selecionou alguns deles para desenvolver na primeira iteração do jogo. Você seguiu o processo de projeto, da análise, de implementação e teste, e no final do dia tinha uma versão funcional do jogo vinte-e-um que distribuía cartas e o deixava jogar.

No Dia 16, você completou a segunda iteração do jogo. Você adicionou mais funcionalidade, como a capacidade de determinar os resultados do jogo. Ao fazer isso, você aprendeu a respeito dos estados e como usá-los para melhorar seu projeto.

O Dia 17 mostrou a você como completar uma outra iteração do jogo vinte-e-um — a aposta. Ao fazer isso, você viu como poderia estender a arquitetura de estado para suportar apostas simples no jogo. Você também viu que, às vezes, é necessário rever e refazer uma hierarquia, quando novos requisitos se apresentam. Embora o fato de refazer apresente um pouco de trabalho extra antecipado, refazer quando apropriado tende a valer a pena, quando você prossegue.

No Dia 18, você concluiu o jogo vinte-e-um, adicionando uma GUI. Para isso, você reviu o padrão MVC discutido em um capítulo anterior.

O Dia 19 forneceu uma GUI alternativa, usando o padrão PAC, àquela desenvolvida no Dia 18. Isso ajudou a refinar seu entendimento de quais padrões são apropriados para cenários específicos.

Durante o Dia 20, você reviu os conceitos do polimorfismo que permitem escrever software à prova do futuro. Você se divertiu um pouco, quando adicionou vários jogadores não-humanos ao sistema e transformou o jogo vinte-e-um em um simulador. Mexendo com estratégias de vários jogadores, você aprendeu o que não deve fazer ao participar de um jogo. Você também aprendeu que pode introduzir tipos de jogador no sistema sem ter de alterar o sistema básico. Esses tipos de jogador não foram nem mesmo considerados quando você construiu o sistema inicial.

Finalmente, no Dia 21, você aprendeu sobre linhas de execução. O capítulo também abordou todas as arestas do projeto e apresentou uma discussão sobre a OO pura, em oposição ao que você provavelmente verá no mundo real.

As lições desta semana aprofundaram seu entendimento de OO e, além disso, você acabou com um divertido e demorado jogo vinte-e-um OO para provar isso.

Após concluir este livro, você tem a base necessária em OO para começar a desenvolver software OO. Tudo de que você precisa agora é prática e experiência. Boa sorte.

Para mais recursos, o Apêndice D, “Bibliografia selecionada”, fornece um ponto de partida para mais informações sobre OO.

# Apêndices

- A Respostas
- B Resumo do Java
- C Referência da UML
- D Bibliografia selecionada
- E Listagens do código do jogo vinte-e-um

A

B

C

D

E

PÁGINA EM BRANCO

# **APÊNDICE A**

## **Respostas**

### **Dia 1 Respostas do teste**

#### **Respostas do teste**

1. Como uma disciplina de software, a programação procedural decompõe um programa em dados e procedimentos para manipular esses dados. A programação procedural tem uma natureza seqüencial. Listas de chamadas procedurais que são executadas seqüencialmente orientam o fluxo de um programa procedural. Um programa procedural termina após chamar sua última procedure.
2. A programação procedural fornece a um programa uma estrutura global: dados e procedimentos. Os procedimentos também o ajudam a ver como deve programar uma tarefa. Em vez de escrever um único bloco de processamento grande, você divide os procedimentos em subprocedimentos. Os procedimentos fornecem um nível de reutilização. Você pode criar bibliotecas de procedimentos reutilizáveis.
3. A programação modular acopla dados e procedimentos fortemente, para manipular esses dados em unidades conhecidas como módulos. Os módulos ocultam o funcionamento e a representação de dados internos de um programa. Entretanto, a maioria das linguagens modulares ainda permite que você use esses módulos em um ambiente procedural.

4. A programação modular oculta a implementação e, assim, protege os dados de manipulação inconsistente ou imprópria. Os módulos também fornecem uma estrutura de nível mais alto para um programa. Em vez de pensar em termos de dados e procedimentos, os módulos permitem que você pense em um nível comportamental conceitual.
5. A programação procedural e a programação modular têm suporte limitado para a reutilização. Embora você possa reutilizar procedimentos, eles são altamente dependentes de seus dados. A natureza global dos dados no mundo procedural torna a reutilização difícil. Os procedimentos podem ter dependências difíceis de quantificar.

Os módulos em si são prontamente reutilizáveis. É possível pegar um módulo e usá-lo em qualquer um de seus programas. Entretanto, os módulos limitam a reutilização. Seu programa só pode usar os módulos diretamente. Você não pode usar um módulo existente como base para um novo módulo.

6. POO é uma disciplina de software que modela o programa em termos de objetos do mundo real. A POO divide o programa em vários objetos inter-relacionados. Ela complementa a programação modular, suportando encapsulamento, assim como eliminando deficiências da reutilização através da herança e deficiências de tipos através do polimorfismo.
7. As seis vantagens da POO são programas que são:

Naturais

Confiáveis

Reutilizáveis

Manuteníveis

Extensíveis

Oportunos

8. A POO é natural. Em vez de modelar os problemas em termos de dados ou procedimentos, a POO permite que você modele seus programas nos termos do problema. Tal estratégia o libera para pensar nos termos do problema e focalizar o que está tentando fazer. Isso retira o foco dos detalhes da implementação.
9. A classe define todos os atributos e comportamentos comuns de um grupo de objetos. Você usa essa definição de classe para criar instâncias desses objetos.

Um objeto é uma instância de uma classe. Seus programas manipulam esses objetos.

Um objeto executa comportamentos. Você também pode chamar os comportamentos de um objeto de interface pública dele. Outros objetos podem exercer qualquer comportamento na interface de um objeto.

10. Os objetos se comunicam através do envio de mensagens de uns para os outros. Chamar um método é sinônimo de fazer uma chamada de método ou procedimento.
11. Um construtor é um método que define como se cria uma instância de objeto. O uso do construtor instanciará um objeto e o tornará disponível para seu programa.
12. Um acessor é um comportamento que dá acesso aos dados internos de um objeto.
13. Um mutante é um comportamento que pode alterar o estado interno de um objeto.
14. `this` é uma referência que cada instância tem para si mesma. A referência `this` dá à instância acesso às suas variáveis e comportamentos internos.

**A**

## Dia 2 Respostas do teste e dos exercícios

### Respostas do teste

1. O encapsulamento é natural. O encapsulamento permite que você modele o software em termos do problema e não nos termos da implementação.

O encapsulamento leva a um software confiável. O encapsulamento oculta o funcionamento interno de um componente de software e garante que ele seja acessado corretamente. O encapsulamento permite que você isole e valide a responsabilidade. Quando você ver um componente agir corretamente, pode reutilizá-lo com confiança.

O encapsulamento proporciona a você um software reutilizável. Como cada componente de software é independente, você pode reutilizar o componente em muitas situações diferentes.

O encapsulamento leva a um código manutenível, pois cada componente é independente. Uma alteração em um componente não danificará outro componente. Assim, a manutenção e os aprimoramentos são simplificados.

O encapsulamento torna seu software modular. As alterações em uma parte de um programa não danificarão o código em outra parte. A modularidade permite que você faça correções de erro e melhorias de funcionalidade sem danificar o restante de seu código.

O encapsulamento leva a um desenvolvimento de código oportuno, pois remove acoplamento de código desnecessário. Freqüentemente, dependências ocultas levam a erros que são difíceis de encontrar e corrigir.

2. Abstração é o processo de simplificar um problema difícil. Quando você começa a resolver um problema, não se sobrecarrega com cada detalhe que envolve o domínio. Em vez disso, você o simplifica, tratando dos detalhes pertinentes à formulação de uma solução.

A área de trabalho gráfica de seu computador é um exemplo de abstração. A área de trabalho oculta completamente os detalhes do sistema de arquivo.

3. Uma implementação define como um componente fornece um serviço. A implementação define os detalhes internos do componente.
4. Uma interface define o quê você pode fazer com um componente. A interface oculta completamente a implementação subjacente.
5. Uma interface descreve o que um componente de software faz; a implementação diz como o componente faz isso.
6. Sem uma divisão clara, as responsabilidades se misturam. Responsabilidades misturadas levam a dois problemas relacionados.

Primeiro, o código que poderia ser centralizado se torna descentralizado. A responsabilidade descentralizada deve ser repetida ou reimplementada em cada lugar onde ela é necessária. Lembre-se do exemplo `BadItem` apresentado anteriormente.

É fácil ver que cada usuário precisaria reimplementar o código para calcular o total ajustado de um item. Sempre que você reescreve a lógica, abre a possibilidade de erros. Você também abre seu código para um uso incorreto, pois a responsabilidade de manter o estado interno não está mais dentro do componente. Em vez disso, você coloca essa responsabilidade nas mãos de outros.

7. Um tipo é um elemento da linguagem que representa alguma unidade de cálculo ou comportamento. Se as linhas de código são frases, os tipos são as palavras. Normalmente, os tipos são tratados como unidades independentes e atômicas.
8. Um TAD é um conjunto de dados e um conjunto de operações sobre esses dados. Os TADs nos permitem definir novos tipos de linguagem, ocultando dados internos e o estado por trás de uma interface bem definida. Essa interface apresenta o TAD como uma única unidade atômica.
9. Existem várias maneiras de obter ocultamento de implementação e código fracamente acoplado. A resposta fácil é usar encapsulamento. Entretanto, um encapsulamento eficaz não é acidente. Aqui estão algumas dicas para o encapsulamento eficaz:
  - Acesse seu TAD apenas através de uma interface de métodos; nunca permita que estruturas internas se tornem parte da interface pública.
  - Não forneça acesso às estruturas de dados internas; abstraia todo o acesso.
  - Não forneça acesso inadvertido às estruturas de dados internas, retornando acidentalmente ponteiros ou referências.
  - Nunca faça suposições a respeito dos outros tipos que você usa. A não ser que um comportamento apareça na interface ou na documentação, não conte com ele.
  - Cuidado ao escrever dois tipos intimamente relacionados. Não se permita programar acidentalmente com base em suposições e dependências.
10. Você precisa conhecer algumas armadilhas da abstração.

Não caia na paralisia da abstração. Resolva os problemas que você encontrar primeiro. Resolver problemas é sua principal tarefa. Veja a abstração como um bônus e não como o objetivo final. Caso contrário, você vai se deparar com a possibilidade de prazos finais perdidos e abstração incorreta. Existem momentos para abstrair e momentos em que a abstração não é apropriada.

A abstração pode ser perigosa. Mesmo que você tenha abstraído algum elemento, isso pode não funcionar em todos os casos. É muito difícil escrever uma classe que satisfaça as necessidades de todos os usuários.

Não coloque em uma classe mais detalhes do que são necessários para resolver o problema.

Não queira resolver todos os problemas. Resolva o problema que está a sua frente e depois procure maneiras de abstrair o que você fez.

A

## Respostas dos exercícios

1. Um possível TAD de pilha:

```
public interface Stack {  
    public void push( Object obj );  
    public Object pop();  
    public boolean isEmpty();  
    public Object peek();  
}
```

2. A pilha é melhor implementada como uma lista encadeada isoladamente, com um ponteiro de frente. Quando coloca ou retira um elemento, você usa o ponteiro de frente para encontrar o primeiro elemento.
3. Examinando a resposta do Exercício 1 e a implementação do Exercício 2, você vê que a interface era adequada. A interface nos proporciona as vantagens que qualquer interface bem definida oferece. Aqui está uma lista breve das vantagens:
  - A interface define a pilha como um tipo. Estudando a interface, você sabe exatamente o que a pilha fará.
  - A interface oculta completamente a representação interna da pilha.
  - A interface define claramente as responsabilidades da pilha.

## Dia 3 Respostas do teste e dos exercícios

### Respostas do teste

1. Account tem dois mutantes: depositFunds() e withdrawFunds().

Account tem um acessor: getBalance().

2. Existem dois tipos de construtores: aqueles que têm argumentos e aqueles que não têm (`noargs`).

`Account`, do Laboratório 2, tem os dois tipos de construtores.

3. (Opcional) `Public` é aceitável no caso de `Boolean`, pois as variáveis são constantes. Ter acesso público às constantes não estraga o encapsulamento, pois isso não está expondo a implementação para uso externo.

Além disso, o uso de valores `Boolean` constantes para verdadeiro e falso economiza memória. Não há necessidade de instanciar suas próprias cópias de `Boolean`. Você pode simplesmente compartilhar essas constantes de instâncias globais.

4. As instâncias de `Card` são imutáveis. Seria mais eficiente definir 52 constantes `Card` — uma para cada carta. Não há necessidade de instanciar várias representações da mesma carta, mesmo quando existem várias instâncias de `Deck`. É perfeitamente seguro compartilhar as mesmas instâncias de `Card` entre os maços de carta.
5. Ao projetar suas classes, você deve perguntar-se o que torna ‘isso’ uma classe? Especificamente, lembre-se da discussão sobre como as classes classificam objetos relacionados. Como as cartas são relacionadas? Todas as cartas contêm um número, um naipe e como exibi-las.

O número ou o naipe não torna uma carta um tipo diferente de carta. Ela ainda é uma carta de pôquer. As cartas de pôquer simplesmente poderiam ter valores diferentes. Assim como um mamífero marrom ainda é um mamífero, um 10 de copas ainda é apenas uma carta.

Às vezes, pode ser extremamente difícil decidir o que deve e o que não deve ser uma classe. Contudo, existe uma regra geral que você pode seguir.

Se você ver que o comportamento de um objeto muda fundamentalmente quando o valor de um atributo muda, as chances são de que você deva criar classes separadas: uma para cada valor possível desse atributo. Claramente, o valor da carta não muda seu comportamento de nenhuma maneira fundamental.

6. A divisão correta de responsabilidades torna o projeto de `Deck`, `Dealer` e `Card` mais modular. Em vez de uma única classe grande, as cartas de pôquer se dividem perfeitamente em três classes. Cada classe é responsável por fazer seu trabalho e ocultar essa implementação das outras classes. Como resultado, essas classes podem mudar sua implementação facilmente, a qualquer momento, sem prejudicar qualquer um de seus usuários.

Com classes separadas, você também tem a vantagem de que pode reutilizar a classe `Card` separada das classes `Deck` e `Dealer`.

## Respostas dos exercícios

1. Aqui está uma possível solução para o Exercício 1:

```
public class DoubleKey {  
  
    private Object key1, key2;  
  
    // um construtor noargs  
    public DoubleKey() {  
        key1 = "key1";  
        key2 = "key2";  
    }  
  
    // um construtor com argumentos  
    // deve procurar e manipular caso nulo  
    public DoubleKey( Object key1, Object key2 ) {  
        this.key1 = key1;  
        this.key2 = key2;  
    }  
  
    // acessor  
    public Object getKey1() {  
        return key1;  
    }  
  
    // mutante  
    // deve procurar e manipular caso nulo  
    public void setKey1( Object key1 ) {  
        this.key1 = key1;  
    }  
  
    // acessor  
    public Object getKey2() {  
        return key2;  
    }  
  
    // mutante  
    // deve procurar e manipular caso nulo  
    public void setKey2( Object key2 ) {  
        this.key2 = key2;  
    }  
  
    // os dois métodos a seguir são exigidos para  
    // trabalhar corretamente como um keyif passado para HashMap ou  
    // Hashtable  
    public boolean equals( Object obj ) {
```

A

```
if( this == obj ) {
    return true;
}

if( this.getClass()== obj.getClass() ) {
    DoubleKey dk = ( DoubleKey )obj;
    if( dk.getKey1().equals( getKey1() ) &&
        dk.getKey2().equals( getKey2() ) ) {
        return true;
    }
}
return false;
}
public int hashCode() {
    return key1.hashCode() + key2.hashCode();
}
}
```

2. Aqui está uma possível solução para o Exercício 2:

```
public class Deck {

    private java.util.LinkedList deck;

    public Deck() {
        buildCards();
    }

    public String display() {
        int num_cards = deck.size();
        String display = "";
        int counter = 0;
        for( int i = 0; i < num_cards; i ++ ) {
            Card card = (Card ) deck.get( i );
            display = display + card.display()+ " ";
            counter++;
            if( counter == 13 ) {
                counter = 0;
                display = display + "\n";
            }
        }
        return display;
    }

    public Card get( int index ) {
        if( index < deck.size() ) {
            return (Card) deck.get( index );
        }
    }
}
```

```
    return null;
}

public void replace( int index, Card card ) {
    deck.set( index, card );
}

public int size() {
    return deck.size();
}

public Card removeFromFront() {
    if( deck.size() > 0 ) {
        Card card = (Card) deck.removeFirst();
        return card;
    }
    return null;
}

public void returnToBack( Card card ) {
    deck.add( card );
}

private void buildCards() {
    deck =new java.util.LinkedList();

    deck.add( new Card( Card.CLUBS, Card.TWO ) );
    deck.add( new Card( Card.CLUBS, Card.THREE ) );
    deck.add( new Card( Card.CLUBS, Card.FOUR ) );
    deck.add( new Card( Card.CLUBS, Card.FIVE ) );
    // definição completa cortada por brevidade
    // veja a listagem completa no código-fonte
}
```

A

## Dia 4 Respostas do teste e dos exercícios

### Respostas do teste

1. A reutilização simples não fornece nenhum mecanismo para reutilização além da instanciação. Para reutilizar código diretamente, você precisa recortar e colar o código que deseja reutilizar. Tal prática resulta em várias bases de código que diferem apenas em um pequeno número de maneiras. Um código que não possui herança é estático. Ele não pode ser estendido. Além disso, um código estático é limitado quanto ao tipo. Um código

estático não pode compartilhar tipo. Assim, você perde as vantagens da capacidade de conexão de tipo.

2. A herança é um mecanismo interno para a reutilização segura e a extensão de definições de classes preexistentes. A herança permite que você estabeleça relacionamentos ‘é um’ entre as classes.

3. As três formas de herança são:

Herança para reutilização de implementação

Herança por diferença

Herança para substituição de tipo

4. A implementação da herança pode cegar um desenvolvedor. A reutilização da implementação nunca deve ser o único objetivo da herança. A substituição de tipo sempre deve ser sua primeira prioridade. A herança para reutilização cega leva a hierarquias de classes que simplesmente não fazem sentido.

5. A programação por diferença é uma das formas de herança. Isso significa que, quando você herda, programa apenas os recursos que diferenciam a nova classe da antiga. Tal prática leva a classes menores e incrementais. As classes menores são mais fáceis de gerenciar.

6. Os três tipos de métodos e atributos são:

Sobrepostos

Novos

Recursivos

Um atributo ou método sobreposto é um atributo ou método declarado na progenitora (ou ancestral) e reimplementado na filha. A filha altera o comportamento do método ou a definição do atributo.

Um método ou atributo novo é um método ou atributo que aparece na filha, mas não na progenitora ou ancestral.

Um atributo ou método recursivo é definido na progenitora ou na ancestral, mas não é redefinido pela filha. A filha simplesmente herda o método ou atributo. Quando é feita uma chamada para esse método ou atributo na filha, a chamada sobe na hierarquia até que seja encontrado alguém que saiba como manipulá-la.

7. A programação pela diferença fornece as menores classes que definem um número menor de comportamentos. Classes menores devem conter menos erros, ser mais fáceis de depurar, mais fáceis de manter e mais fáceis de entender.

A programação pela diferença permite que você programe de forma incremental. Como resultado, um projeto pode evoluir com o passar do tempo.

8. `AllPermission`, `BasicPermission` e `UnresolvedPermission` são todas filhas de `Permission`. `SecurityPermission` é descendente de `Permission`.

`Permission` é a classe-raiz. `AllPermission`, `UnresolvedPermission` e `SecurityPermission` são todas classes folhas, pois elas não têm filhas.

A

Sim, `Permission` é ancestral de `SecurityPermission`.

9. Herança por substituição de tipo é o processo de definir relacionamentos com capacidade de substituição. A capacidade de substituição permite que você substitua uma descendente por uma ancestral, desde que não precise usar quaisquer novos métodos definidos pela descendente.
10. A herança pode destruir o encapsulamento, dando a uma subclasse acesso inadvertido à representação interna de uma superclasse.

A destruição inadvertida do encapsulamento é uma armadilha que pode apanhá-lo sorrateiramente. A herança fornece silenciosamente a uma classe filha acesso mais liberal à progenitora. Como resultado, se os passos corretos não forem dados, uma filha poderia ter acesso direto à implementação da progenitora. O acesso direto à implementação é tão perigoso entre progenitora e filha quanto entre dois objetos. Muitas das mesmas armadilhas ainda se aplicam.

Evite a destruição do encapsulamento tornando a implementação interna privada. Faça implementação interna privada apenas dos métodos absolutamente necessários para uma subclasse.

Na maioria das vezes, as filhas devem exercitar apenas a interface pública da progenitora.

## Respostas dos exercícios

1. Toda subclasse terá acesso direto à representação interna de `Point`. Tal acesso irrestrito destrói o encapsulamento e abre a classe para os problemas tratados na pergunta 10.

Remediar a situação é tão fácil quanto tornar `x` e `y` privados. Note que essa classe `Point` é modelada de acordo com `java.awt.Point`.

## Dia 5 Respostas do teste

### Respostas do teste

1. Em `CheckingAccount`, `public double withdrawFunds( double amount )` é um exemplo de método redefinido. `CheckingAccount` sobreporá `withdrawFunds()` para que ele possa controlar o número de transações.

Em `BankAccount`, `public double getBalance()` é um exemplo de método recursivo. O método aparece na progenitora, mas nenhuma das subclasses o redefine.

Entretanto, as subclasses o chamam.

Finalmente, o método `public double getInterestRate()` de `SavingsAccount` é um exemplo de método novo. O método aparece apenas na classe `SavingsAccount` e não na progenitora.

2. Você usaria uma classe abstrata para herança planejada. Uma classe abstrata fornece às suas subclasses um indício do que ela precisará redefinir. As classes abstratas garantem que suas subclasses usem a classe base corretamente.
3. O Laboratório 3 mostra ‘tem um’. `Deck` tem objetos `Card`. `DoubleKey`, do Laboratório 2, também tem duas `String`s.
4. Os laboratórios preservaram o encapsulamento ocultando todos os membros de dados. Se você examinar as soluções, verá que todos os dados são privados. Por exemplo, a classe `BankAccount` declara o saldo (*balance*) como privado. Em vez disso, cada classe dá acesso à representação dos dados através de uma interface bem definida.
5. `SavingsAccount` é um exemplo de especialização. Ela se especializa em relação à sua progenitora, `BankAccount`, adicionando métodos para configurar, consultar e aplicar juros na conta.
6. O Laboratório 3 usa herança para reutilizar o comportamento básico definido pela classe `BankAccount`. `BankAccount` define uma implementação comum para sacar fundos, depositar fundos e consultar o saldo. Através da herança, as subclasses de conta obtêm essa implementação.

O Laboratório 4 começa apresentando um caso de herança para reutilização da implementação, mas termina usando composição, para obter uma forma de reutilização mais limpa.

## Dia 6 Respostas do teste e dos exercícios

### Respostas do teste

1. Inclusão

Paramétrico

Sobreposição

Sobrecarga

2. O polimorfismo de inclusão permite que você trate um objeto como se fosse um tipo diferente de objeto. Como resultado, um objeto pode demonstrar muitos tipos diferentes de comportamento.
3. Os polimorfismos de sobrecarga e paramétrico permitem que você modele algo em nível conceitual. Em vez de se preocupar com os tipos de parâmetros que algo processa, você pode escrever seu código de forma mais genérica. Em vez disso, você modela seus métodos e tipos em nível conceitual do que eles fazem e não para que fazem.
4. Uma interface pode ter qualquer número de implementações. Programando para uma interface, você não fica vinculado a nenhuma implementação específica. Como resultado, seu programa pode usar automaticamente qualquer implementação que apareça. Essa liberdade de implementação permite que você troque entre diferentes implementações para mudar o comportamento de seu programa.
5. Quando você sobrepõe um método, o polimorfismo garante que a versão correta do método seja chamada.
6. Polimorfismo *ad-hoc* é outro nome para sobrecarga.
7. A sobrecarga permite a você definir um nome de método várias vezes. Cada definição difere simplesmente no número e nos tipos de argumentos. A sobrecarga expressa um comportamento diferente porque você simplesmente chama o método. Você não precisa fazer nada para garantir que a versão correta do método seja chamada.

A sobrecarga permite a você modelar um método em nível conceitual do que ele faz. A natureza polimórfica da sobrecarga cuida dos argumentos específicos.

8. O polimorfismo paramétrico permite a você escrever tipos e métodos verdadeiramente genéricos, adiando as definições de tipo até o momento da execução. Esse tipo de polimorfismo permite que você escreva código realmente natural, pois pode programar tipos e métodos muito genéricos e conceituais. Você escreve esses tipos e métodos a partir da visão conceitual do que eles fazem e não do que especificamente fazem com ela. Por exemplo, se você programar um método compare ( [T] a, [T] b), pensará em termos do conceito mais alto da comparação de dois objetos de tipo [T]. Os argumentos de tipo [T] simplesmente precisariam compartilhar uma estrutura comum, como < ou um método compare(). O ponto importante é que você escreve simplesmente um método e ele pode comparar muitos tipos diferentes de objetos.
9. O polimorfismo normalmente incorrerá em um custo na eficiência. Algumas formas e implementações de polimorfismo exigem verificações e pesquisas em tempo de execução. Essas verificações são dispendiosas, quando comparadas com as linguagens estaticamente tipadas.

A

O polimorfismo tenta fazer o desenvolvedor quebrar a hierarquia de herança. Você nunca deve mover funcionalidade para cima na hierarquia, simplesmente para aumentar as oportunidades para comportamento polimórfico.

Quando trata um subtipo como se ele fosse o tipo base, você perde o acesso a todos os comportamentos adicionados pelo subtipo. Assim, quando você criar um novo subtipo, precisará garantir que a interface do tipo base seja adequada para a interação com seu novo subtipo, em métodos que trabalham com o tipo base.

10. A herança eficaz tem um impacto direto no polimorfismo de inclusão. Para aproveitar a capacidade de conexão oferecida pelo polimorfismo de subtipo, você deve ter uma hierarquia de objetos correta.

O encapsulamento evita que um objeto fique vinculado a uma implementação específica. Sem encapsulamento, um objeto poderia se tornar facilmente dependente da implementação interna de outro objeto. Tal acoplamento forte torna a substituição difícil, se não impossível.

## Respostas dos exercícios

1. Imagine um programa que escreva seu status na linha de comando ao ser executado. Na linguagem Java, você poderia simplesmente usar `System.write.println()`, para escrever essas mensagens na tela. E se você quisesse que essas mensagens fossem gravadas em um arquivo? E se você quisesse que essas mensagens fossem enviadas para uma GUI de alarme em outro computador? Obviamente, você precisaria alterar seu código, conforme os requisitos mudassem.

E se seus requisitos exigissem que você oferecesse suporte para ambos ao mesmo tempo? Em vez de um ou outro, você quer permitir que o usuário escolha onde vai escrever, através de um argumento de linha de comando. Sem polimorfismo, você precisaria programar casos para cada tipo de escrita. Com o polimorfismo, entretanto, você pode simplesmente declarar uma classe chamada `log`, que tenha um método de escrita. Subclasses podem especificar onde as mensagens são gravadas. Você pode adicionar novos subtipos em seu programa a qualquer momento. O programa saberá automaticamente como usar os novos subtipos, desde que você programe para a interface `log`. Assim, você pode trocar para o novo comportamento de log a qualquer momento.

2. `int i = 2 + 3.0`

Dependendo da definição de `+`, essa instrução pode ser coerciva. Aqui, a instrução tenta somar um número inteiro e um número real. Ela pega o resultado e o coloca em uma variável `int`. Dependendo da linguagem, o compilador pode converter o inteiro 2 em um número real, efetuar a operação aritmética e, em seguida, converter o resultado novamente para um inteiro.

Essa instrução é interessante, pois ela também pode demonstrar uma instância de sobre-carga. + pode sobrecarregar as seguintes operações:

```
+ (real, real)  
+ (integer, integer)  
+ (integer, real)
```

Em qualquer caso, você tem polimorfismo *ad-hoc*, pois, em vez de um método polimórfico, você tem vários métodos polimórficos ou conversão.

A

### 3. Sobrecarga:

Considere `java.util.SimpleTimeZone`. `SimpleTimeZone` define os dois métodos sobre-carregados a seguir: `setEndRule` e `setStartRule`. Como resultado, esses métodos respon-dem diferentemente, dependendo do número e dos tipos de entrada.

Polimorfismo de inclusão:

Considere `java.io.Writer`. A classe abstrata `Writer` define vários métodos para gravar dados. Correntemente, a linguagem Java define várias subclasses de `Writer`: `BufferedW-riter`, `CharArrayWriter`, `FilterWriter`, `OutputStreamWriter`, `PipedWriter`, `PrintWriter` e `StringWriter`. Cada subclasse define o comportamento dos métodos `close`, `flush` e `write` (um método sobrecarregado, a propósito).

Ao programar, você deve escrever seus objetos e métodos para agirem em instâncias de `Write`. Desse modo, você pode trocar para diferentes subclasses, dependendo de como deseja que os dados sejam gravados. Se você programar dessa maneira, `Writer` expressa rá vários comportamentos diferentes, dependendo da implementação subjacente que es-teja sendo usada.

## Dia 7 Respostas do teste

### Respostas do teste

1. O método `observe()` de `PsychiatristObject` é um exemplo de sobre-carga de método.
2. O método `observe()` ilustra muito bem o problema da sobre-carga. Sempre que você adi-cionar um novo subtipo, precisará adicionar outro método sobre-carregado. À medida que o número de métodos aumentar, você desejará encontrar uma maneira de adicionar uma função comum em seus objetos, para que possa tratá-los genericamente e remover os mé-todos sobrepostos.
3. São dois passos para adicionar novo comportamento em uma hierarquia polimórfica. Primeiro, crie o novo tipo. Segundo, altere seu programa de modo que ele possa criar ins-tâncias do novo tipo. Você não deve ter de mudar nada mais, a não ser que precise tirar proveito de algum recurso especial da nova classe.

4. O método `examine()` de `PsychiatristObject` é um exemplo de polimorfismo de inclusão. Ele pode trabalhar com qualquer subtipo de `MoodyObject`.
5. Você pode eliminar as condicionais atacando os dados que fazem parte das condicionais. Se os dados não são um objeto, transforme-os em um objeto. Se os dados são um objeto, adicione um método que forneça o comportamento necessário. Uma vez feito isso, peça ao objeto para que faça algo, não faça algo nos dados.
6. O polimorfismo de inclusão permitirá que um método funcione para o tipo de argumento e qualquer subtipo. Você não precisa de um método diferente para cada subtipo. Ter apenas um método diminui o número de métodos que, de outra forma, você precisaria. Isso também simplifica a adição de novos recursos.
7. Na OO, você não deve pedir os dados de um objeto. Em vez disso, você deve pedir a um objeto para que faça algo com seus dados.
8. As condicionais o obrigam a quebrar o relacionamento delineado no Exercício 7. Quebrar o relacionamento o obriga a misturar responsabilidades, pois todo usuário precisará saber o quê os dados representam e como manipulá-los.
9. Se você se encontrar atualizando várias condicionais, sempre que adiciona um novo tipo, então a condicional é um problema. Se você se encontrar escrevendo a mesma condicional em vários lugares (ou chamando um método que tenha a condicional), então a condicional é um problema.
10. O polimorfismo permite que você trate um subtipo como se ele fosse o supertipo. Entretanto, o polimorfismo permite que você use o comportamento do tipo subjacente real. O polimorfismo faz parecer que o supertipo manifesta muitos comportamentos diferentes.

## Dia 8 Respostas do teste e dos exercícios

### Respostas do teste

1. A UML é a Unified Modeling Language. A UML é uma linguagem de modelagem padrão do setor.
2. Uma metodologia descreve como projetar software. Uma linguagem de modelagem ajuda a capturar esse projeto graficamente. Uma metodologia freqüentemente conterá sua própria linguagem de modelagem.
3. O laboratório demonstra um relacionamento de dependência.
4. Você pode fazer duas afirmativas sobre `MoodyObject`. `MoodyObject` tem um método chamado `queryMood()`. `MoodyObject` também é uma classe abstrata. O nome em itálico indica que a classe é abstrata.

5. O relacionamento Employee, do Laboratório 1, é um exemplo de dependência. O método payEmployees() de Payroll depende da interface pública de Employee.
6. Cada um desses símbolos transmite informações de visibilidade. + é público, # é protegido, e - é privado .
7. Um objeto Queue e seus elementos são um exemplo de agregação.
8. O objeto Deck tem muitas cartas. Entretanto, se você destruir o baralho, deverá destruir as cartas. O objeto Deck é um exemplo de relacionamento de composição.
9. Basta deixar um nome de classe ou método em itálico para mostrar que ele é abstrato.
10. O objetivo final da modelagem é transmitir seu projeto. Conseqüentemente, você não deve se preocupar com o uso de toda notação de modelagem disponível. Em vez disso, você deve usar a notação mínima que ainda transmite sua mensagem com êxito.
11. Uma associação modela relacionamentos estruturais entre objetos. A agregação e a composição são subtipos da associação que modela relacionamentos ‘todo/parte’. Uma agregação é um relacionamento estrutural entre pares. A composição é um relacionamento estrutural onde a parte não é independente do todo. A parte não pode existir separadamente do todo.
12. Modele uma associação quando o objetivo de seu modelo for modelar os papéis entre os objetos. Use agregação ou composição quando você estiver tentando transmitir o projeto estrutural.

A

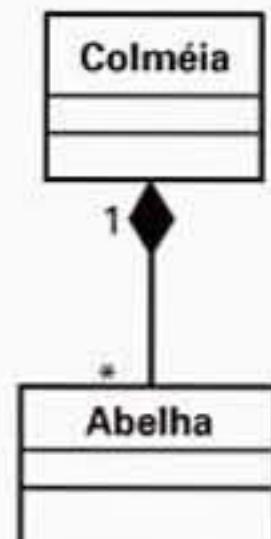
## Respostas dos exercícios

1.

**FIGURA A.1***Uma fila.*

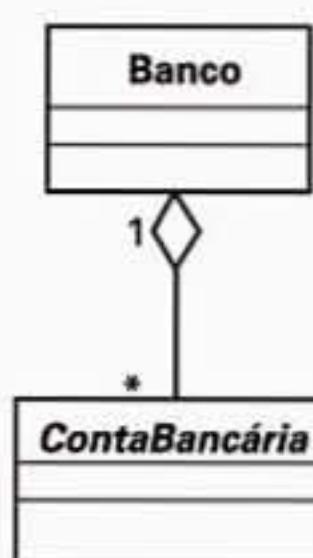
Fila
+ enqueue (obj : Object) : void
+ dequeue () : Object
+ isEmpty () : boolean
+ peek () : Object

2.

**FIGURA A.2***Um relacionamento de composição Abelha/Colméia.*

3.

**FIGURA A.3**  
*Um relacionamento de agregação Banco/Conta Bancária.*



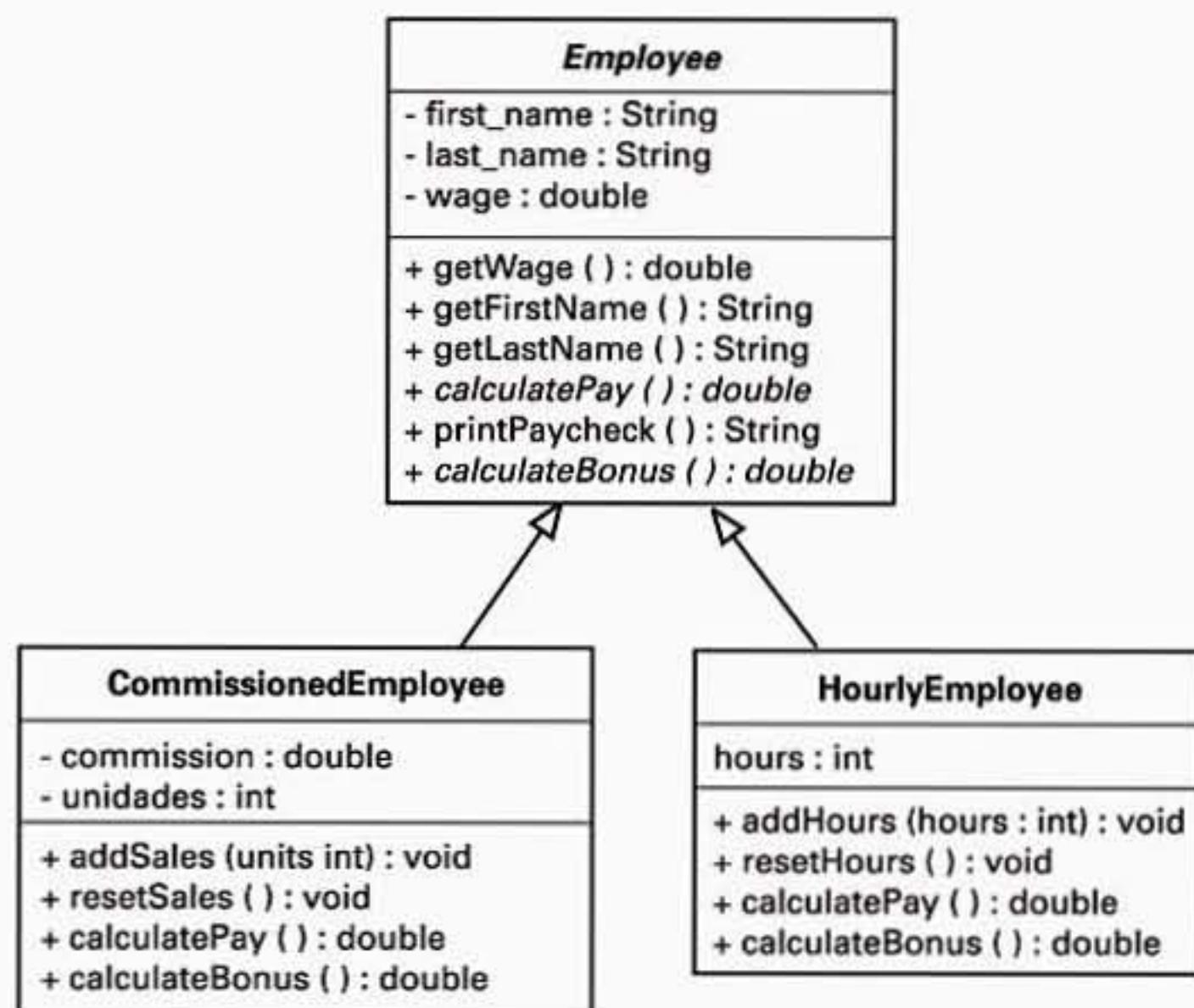
4.

**FIGURA A.4**  
*A associação Comprador/Comerciante.*

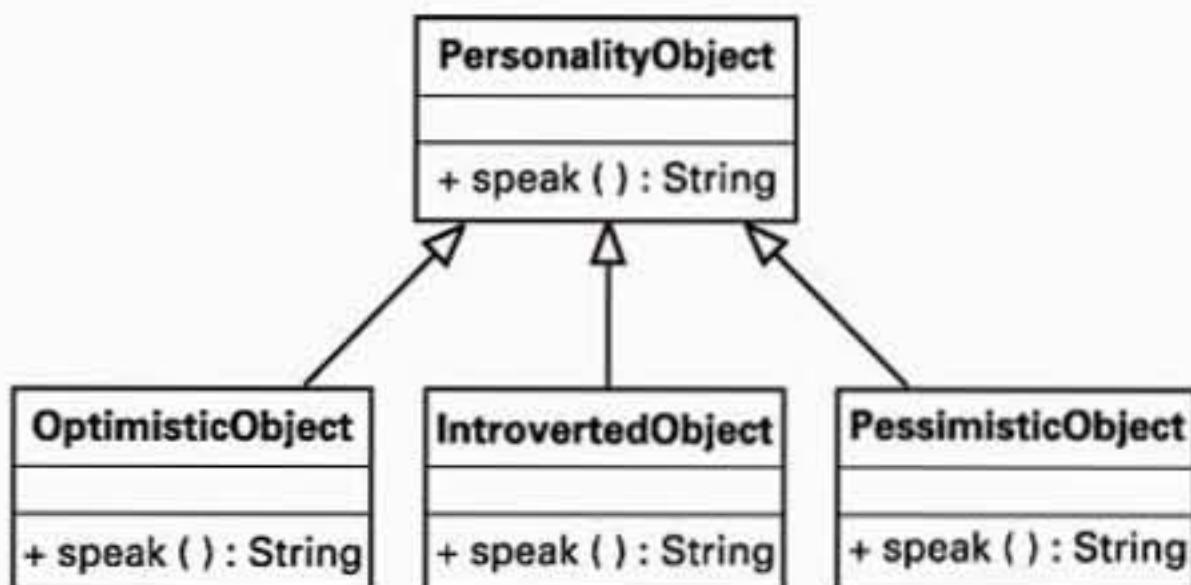


5.

**FIGURA A.5**  
*A hierarquia Employee.*



**FIGURA A.6**  
*A hierarquia PersonalityObject.*



A

## Dia 9 Respostas do teste e dos exercícios

### Respostas do teste

1. Um processo de software dispõe os vários estágios do desenvolvimento do software.
2. Um processo iterativo é um processo que permite a você voltar e refazer ou melhorar continuamente o produto de iterações anteriores. Um processo iterativo adota uma estratégia iterativa e incremental para o desenvolvimento de software.

Incremental significa que cada iteração acrescenta um pequeno aumento na funcionalidade. Não tão pequena para passar desapercebida, mas não tão grande para ser deixada de lado por ser demasiadamente dispendiosa.

3. No final da AOO você deve ter um bom entendimento dos requisitos do sistema, assim como do vocabulário do domínio do sistema.
4. Os requisitos informam o que os usuários querem fazer com o sistema e quais tipos de respostas eles esperam receber.

Os requisitos são aqueles recursos que o sistema deve ter para resolver determinado problema.

5. Um caso de uso descreve a interação entre o usuário do sistema e o sistema. O caso de uso descreve como o usuário utilizará o sistema a partir de seu ponto de vista.
6. Você deve dar os seguintes passos para definir seus casos de uso:

1. Identificar os atores
2. Criar uma lista preliminar de casos de uso
3. Refinar e nomear os casos de uso
4. Definir a seqüência de eventos de cada caso de uso
5. Modelar seus casos de uso
7. Um ator é algo que interage com o sistema.

8. Você pode fazer as seguintes perguntas para ajudar a encontrar atores:

- Quem principalmente vai usar o sistema?
- Existem outros sistemas que usarão o sistema? Por exemplo, existem usuários não-humanos?
- O sistema vai se comunicar com qualquer outro sistema? Por exemplo, já existe um banco de dados que você precisa integrar?
- O sistema responde a estímulos não gerados pelo usuário? Por exemplo, o sistema precisa fazer algo em determinado dia de cada mês? Um estímulo pode ser proveniente de fontes normalmente não consideradas ao se pensar do ponto de vista puramente do usuário.

9. Um caso de uso pode conter e utilizar outro caso de uso, ou estender outro caso de uso. Um caso de uso também pode ser uma variante de outro caso de uso.

10. Uma variante de caso de uso é um caso especial de um caso de uso mais geral.

11. Um cenário é uma seqüência ou fluxo de eventos entre o usuário e o sistema.

12. Você pode modelar seus casos de uso através de diagramas de interação e diagramas de atividade. Existem dois tipos de diagramas de interação: diagramas de seqüência e de colaboração.

13. Os diagramas de seqüência modelam a seqüência de eventos com o passar do tempo. Um diagrama de colaboração modela as interações entre os atores de um caso de uso. Os dois tipos de diagramas são diagramas de interação. Entretanto, cada um adota um ponto de vista diferente em relação ao sistema. Use diagramas de seqüência quando quiser controlar eventos e diagramas de colaboração, quando quiser destacar relacionamentos.

Os diagramas de atividade o ajudam a modelar processos paralelos. Use diagramas de atividade quando quiser transmitir o fato de que um processo pode ser executado em paralelo com outros processos, durante um cenário de caso de uso.

14. Um modelo de domínio apresenta várias vantagens. O modelo de domínio pode servir como base ou esqueleto de seu modelo de objetos. Você pode usar essa base como um início e construir a partir dela.

Os modelos de domínio também fornecem um vocabulário comum e um entendimento do problema.

15. Os casos de uso o ajudam a entender o sistema, seus requisitos e seus usos.

Os casos de uso podem ajudá-lo a planejar as iterações de seu projeto.

Finalmente, os casos de uso o ajudam a definir seu modelo de domínio.

## Respostas dos exercícios

1. Alguns outros casos de uso:

- Remover Item: um usuário pode remover um item do carrinho.
- Excluir Usuário: um administrador pode remover contas inativas.
- Premiar Usuário: o sistema pode recompensar clientes freqüentes, oferecendo descontos dinamicamente.

2. O usuário seleciona um item do carrinho de compras. O usuário remove o item selecionado do carrinho de compras.

- Remover Item.

1. O usuário convidado seleciona um item do carrinho de compras.

2. O usuário convidado pede ao carrinho para que remova o item:

- Condições prévias.
- O carrinho contém um item para remover.
- Condições posteriores.
- O item não aparece mais no carrinho.
- Alternativa: Operação Cancelada.

O usuário pode optar por cancelar a transação após o passo 1.

3. Os dois casos de uso a seguir são variantes do caso de uso Pesquisar o Catálogo de Produtos:

- Os usuários convidados podem pesquisar o catálogo de produtos.
- Os usuários convidados podem procurar um item específico.

Os dois casos de uso a seguir são variantes do caso de uso Assinar Notificações:

- Os usuários registrados podem assinar notificações.
- Os usuários registrados podem assinar várias listas de mensagens.

4. Existem muitos outros objetos de domínio. Aqui estão alguns: Administrador, Lista de Produtos Destacados e Lista de Pedidos.

A

## Dia 10 Respostas do teste e dos exercícios

### Respostas do teste

1. Existem três vantagens em um projeto formal. Um projeto formal o ajuda a descobrir quais objetos aparecerão em seu programa e como eles vão interagir ou se encaixar.

Um projeto o ajuda a prever muitos dos problemas de projeto que apareceriam durante a implementação. É muito mais fácil corrigir um projeto antes que seja codificado.

Finalmente, um projeto ajuda a garantir que todos os desenvolvedores estejam na mesma página; caso contrário, você corre o risco de que cada desenvolvedor desenvolva partes incompatíveis.

2. POO é o processo de construir o modelo de objetos de uma solução. Dito de outra maneira, POO é o processo de decompor uma solução em vários objetos constituintes.
3. O *modelo de objetos* é o projeto dos objetos que aparecem na solução de um problema. O *modelo de objetos* final pode conter muitos objetos não encontrados no domínio. O *modelo de objetos* descreverá as responsabilidades, relacionamentos e estrutura dos vários objetos.
4. Simplesmente não é possível prever cada decisão de projeto, antes que você a tome, e isso nem sempre vale a pena. Algum projeto pode ser deixado até a construção. Além disso, você não quer ser pego tentando criar o projeto perfeito. Você precisa começar a codificar, algum dia.
5. As partes significativas são aqueles aspectos do sistema que alteram completamente sua estrutura ou seu comportamento. Essas são as partes que realmente importam, quando você codifica a solução. Uma mudança em uma parte arquitetônica importante mudará a estrutura da solução.
6. Os cinco passos básicos do POO são:
  1. Gerar a lista inicial de objetos.
  2. Refinar as responsabilidades de seus objetos através de cartões CRC.
  3. Desenvolver os pontos de interação.
  4. Detalhar os relacionamentos entre os objetos.
  5. Construir seu modelo.
7. Começar com o domínio para gerar sua lista inicial de objetos. Cada objeto e cada ator do domínio deve se tornar uma classe em seu modelo de objetos.

Sistemas de terceiros, interfaces de hardware, relatórios, telas e dispositivos também devem se tornar classes.
8. Um projeto completo capturará as responsabilidades de cada objeto, assim como a estrutura e os relacionamentos do objeto. Um projeto mostrará como tudo se encaixa.
9. Os cartões CRC o ajudam a identificar responsabilidades e colaborações de cada classe.
10. *Colaboração* é o relacionamento de delegação entre dois objetos. Você pode considerar uma colaboração como um relacionamento cliente/servidor entre dois objetos.

11. Praticamente, as responsabilidades se traduzirão em métodos. Os relacionamentos se traduzirão em uma estrutura; entretanto, um entendimento global das responsabilidades o ajudará a dividir as responsabilidades eficientemente entre os objetos. Você precisa evitar ter um conjunto pequeno de objetos muito grandes. Através do projeto, você garantirá a dispersão das responsabilidades.
12. Um cartão CRC é uma ficha de arquivo 4x6 que o ajuda a descobrir as responsabilidades e colaborações de um objeto, explorando casos de uso.
13. Você está intencionalmente limitado pelo tamanho de um cartão CRC. Se você verificar que está ficando sem espaço, são boas as chances de que sua classe esteja fazendo coisas demais.
14. Você deve usar cartões CRC durante os estágios iniciais do desenvolvimento, especialmente quando você ainda for iniciante no desenvolvimento OO. Os cartões CRC se prestam para pequenos projetos ou para uma pequena seção de um projeto maior.

Você só deve usar cartões CRC para descobrir responsabilidades e colaborações. Não tente descrever relacionamentos complexos através de cartões CRC.

15. Os cartões CRC não funcionam bem para projetos grandes ou para grupos de desenvolvimento. Um grande número de classes pode atrapalhar uma sessão de cartões CRC. Desenvolvedores demais também podem danificar uma sessão de cartões CRC.
16. Um *ponto de interação* é qualquer lugar onde um objeto use outro.
17. Em um ponto de interação, você deve considerar transformação de dados, alteração futura, interfaces e o uso de agentes.
18. Um *agente* é um objeto que faz a intermediação entre dois ou mais objetos para cumprir algum objetivo.
19. Você vai definir as dependências, associações e generalizações. Detalhar esses relacionamentos é um passo importante, pois isso define como os objetos se encaixam. Isso também define a estrutura interna dos vários objetos.
20. Você poderia criar diagramas de classe, diagramas de atividade e diagramas de interação para modelar seu projeto. A UML também define diagramas de objeto e diagramas de estado.

A

## Respostas dos exercícios

1. As instâncias de ShoppingCart terão a responsabilidade global de conter itens. Especificamente, um objeto ShoppingCart pode adicionar um item em si mesmo, remover um item de si mesmo e permitir que um objeto externo selecione um item, sem removê-lo.

# Dia 11 Respostas do teste e dos exercícios

## Respostas do teste

1. Uma classe adaptadora transforma a interface de um objeto naquela esperada por seu programa. Um adaptador contém um objeto e delega mensagens da nova interface para a interface do objeto contido.
2. O padrão Iterator descreve um mecanismo para fazer laço pelos elementos de uma coleção.
3. Você usaria o padrão Iterator para conter lógica de navegação em um local, fornecer um modo padrão de percorrer coleções e ocultar do usuário a implementação da coleção.
4. O padrão Adapter descreve um mecanismo que permite transformar uma interface de objetos.
5. Você usaria o padrão Adapter quando precisasse utilizar um objeto que tivesse uma interface incompatível. Você também pode usar empacotadores preventivamente, para isolar seu código das mudanças de API.
6. O padrão Proxy intermedia de forma transparente o acesso a um objeto. Os proxies adicionam um procedimento indireto no uso do objeto.
7. Você usaria o padrão Proxy sempre que quisesse intermediar o acesso a um objeto de maneira que uma simples referência não permite. Exemplos comuns incluem recursos remotos, otimizações e limpeza geral de objeto, como contagem de referência ou reunião de estatísticas de utilização.
8. Nessa situação, você pode usar o padrão Adapter para criar uma interface independente daquela fornecida pela Sun, IBM ou Apache. Criando sua própria interface, você pode permanecer independente da API ligeiramente diferente de cada fornecedor. Empacotando a biblioteca, você está livre para trocar de biblioteca a qualquer momento, seja para migrar para uma nova versão ou para trocar de fornecedor, pois você controla a interface do adaptador.
9. Nessa situação, você pode usar o padrão Proxy para ocultar a identidade do objeto que armazena os dados dos objetos que o chamam. Dependendo da localização do cliente, você pode instanciar um proxy interligado em rede ou um proxy local. De qualquer modo, o restante do programa não notará a diferença; portanto, todos os seus objetos podem usar uma interface proxy sem ter de se preocupar com a implementação subjacente.
10. O padrão Proxy não muda uma interface, no sentido de que ele não retira nada dela. Entretanto, um proxy está livre para adicionar mais métodos e atributos na interface.

## Respostas dos exercícios

1.

### LISTAGEM 11.13 ShoppingCart.java

```
public class ShoppingCart {  
  
    java.util.LinkedList items = new java.util.LinkedList();  
  
    /**  
     * adiciona um item no carrinho  
     * @param item o item a ser adicionado  
     */  
    public void addItem( Item item ) {  
        items.add( item );  
    }  
  
    /**  
     * remove o item dado do carrinho  
     * @param item o item a ser removido  
     */  
    public void removeItem( Item item ) {  
        items.remove( item );  
    }  
  
    /**  
     * @return int o número de itens no carrinho  
     */  
    public int getNumberItems() {  
        return items.size();  
    }  
  
    /**  
     * recupera o item indexado  
     * @param index o índice do item  
     * @return Item o item no índice  
     */  
    public Item getItem( int index ) {  
        return (Item) items.get( index );  
    }  
    public Iterator iterator() {  
        // ArrayList tem um método iterator() que retorna um iterator  
        // entretanto, para propósitos de demonstração, ajuda ver um iterator  
        // simples  
        return new CartIterator( items );  
    }  
}
```

A

**LISTAGEM 11.14** CartIterator.java.

```
public class CartIterator implements Iterator {  
  
    private Object [] items;  
    private int index;  
  
    public CartIterator( java.util.LinkedList items ) {  
        this.items = items.toArray();  
    }  
  
    public boolean isDone() {  
        if( index >= items.length ) {  
            return true;  
        }  
        return false;  
    }  
  
    public Object currentItem() {  
        if( !isDone() ) {  
            return items [index];  
        }  
        return null;  
    }  
  
    public void next() {  
        index++;  
    }  
  
    public void first() {  
        index = 0;  
    }  
}
```

---

2. Tornando o adaptador mutante, você pode usar o mesmo empacotador para empacotar muitos objetos diferentes e não precisa instanciar um empacotador para cada objeto que precise ser empacotado. A reutilização de empacotadores faz melhor uso da memória e libera seu programa de ter de pagar o preço da instanciação de muitos empacotadores.

**LISTAGEM 11.15** MutableAdapter.java

```
public class MutableAdapter extends MoodyObject {  
  
    private Pet pet;  
  
    public MutableAdapter( Pet pet ) {
```

**LISTAGEM 11.15 MutableAdapter.java (continuação)**

```
        setPet( pet );
    }

protected String getMood() {
    // implementando apenas porque é exigido
    // por MoodyObject, pois também sobrepõe queryMood
    // não precisamos disso
    return pet.speak();
}

public void queryMood() {
    System.out.println( getMood() );
}

public void setPet( Pet pet ) {
    this.pet = pet;
}
}
```

A

## **Dia 12 Respostas do teste e dos exercícios**

### **Respostas do teste**

1. Uma classe empacotadora transforma a interface de um objeto naquela esperada por seu programa. Um empacotador contém um objeto e delega mensagens da nova interface para a interface do objeto contido.
2. O padrão Abstract Factory fornece um mecanismo que instancia instâncias de classe descendentes específicas, sem revelar qual descendente é realmente criada. Isso permite conectar, de forma transparente, diferentes descendentes em seu sistema.
3. Você usa o padrão Abstract Factory para ocultar os detalhes da instanciação, para ocultar a classe de objetos que é instanciada e quando quer que um conjunto de objetos seja usado junto.
4. O padrão Singleton garante que um objeto seja instaciado apenas uma vez.
5. Você usa o padrão Singleton quando quer que um objeto seja instaciado apenas uma vez.
6. Usar constantes primitivas não é uma estratégia de OO para programação, pois você tem de aplicar um significado externo à constante. Você viu quantos problemas o desdobramento da responsabilidade poderia causar!

O padrão Typesafe Enum resolve esse problema, transformando a constante em um objeto de nível mais alto. Usando um objeto de nível mais alto, você pode encapsular melhor a responsabilidade dentro do objeto constante.

7. Você deve usar o padrão Typesafe Enum quando se encontrar declarando constantes públicas que devem ser objetos propriamente ditos.
8. Não, os padrões não garantem um projeto perfeito, pois você poderia acabar usando um padrão incorretamente. Além disso, usar um padrão corretamente não significa que o restante de seu projeto seja válido. Muitos projetos válidos nem mesmo contêm um padrão.

## Respostas dos exercícios

1.

---

### LISTAGEM 12.19 Bank.java

---

```
public class Bank {  
  
    private java.util.Hashtable accounts = new java.util.Hashtable();  
  
    private static Bank instance;  
  
    protected Bank() {}  
  
    public static Bank getInstance(){  
        if( instance == null ) {  
            instance = new Bank();  
        }  
        return instance;  
    }  
  
    public void addAccount( String name, BankAccount account ) {  
        accounts.put( name, account );  
    }  
  
    public double totalHoldings() {  
        double total = 0.0;  
  
        java.util.Enumeration enum = accounts.elements();  
        while( enum.hasMoreElements() ) {  
            BankAccount account = (BankAccount) enum.nextElement();  
            total += account.getBalance();  
        }  
        return total;  
    }  
}
```

**LISTAGEM 12.19** Bank.java (*continuação*)

```
public int totalAccounts() {
    return accounts.size();
}

public void deposit( String name, double amount ) {
    BankAccount account = retrieveAccount( name );
    if( account != null ) {
        account.depositFunds( amount );
    }
}

public double balance( String name ) {
    BankAccount account = retrieveAccount( name );
    if( account != null ) {
        return account.getBalance();
    }
    return 0.0;
}

private BankAccount retrieveAccount( String name ) {
    return (BankAccount) accounts.get( name );
}
```

A

2.

**LISTAGEM 12.20** Level.java

```
public final class Level {

    public final static Level NOISE    = new Level( 0, "NOISE" );
    public final static Level INFO     = new Level( 1, "INFO" );
    public final static Level WARNING  = new Level( 2, "WARNING" );
    public final static Level ERROR    = new Level( 3, "ERROR" );

    private int level;
    private String name;

    private Level( int level, String name ) {
        this.level = level;
        this.name = name;
    }

    public int getLevel() {
```

**LISTAGEM 12.20** Level.java (*continuação*)

```
    return level;
}

public String getName() {
    return name;
}
}
```

---

**LISTAGEM 12.21** Error.java

```
public class Error {

    private Level level;

    public Error( Level level ) {
        this.level = level;
    }

    public Level getLevel() {
        return level;
    }

    public String toString() {
        return level.getName();
    }
}
```

---

3. A solução consiste em uma factory abstrata de conta bancária (escrita como uma interface; entretanto, ela também pode ser uma classe abstrata) e em uma Factory concreta de conta bancária. A factory tem um método para criar cada tipo de conta bancária.

Essa factory oculta os detalhes da instanciação e não necessariamente o subtipo do objeto.

**LISTAGEM 12.22** AbstractAccountFactory.java

```
public interface AbstractAccountFactory {

    public CheckingAccount createCheckingAccount( double initDeposit, int trans,
double fee );

    public OverdraftAccount createOverdraftAccount( double initDeposit, double
rate );

    public RewardsAccount createRewardsAccount( double initDeposit, double
interest, double min );
```

**LISTAGEM 12.22 AbstractAccountFactory.java (continuação)**

```
public SavingsAccount createSavingsAccount( double initBalance, double  
interestRate );  
public TimedMaturityAccount createTimedMaturityAccount( double initBalance,  
double interestRate, double feeRate );  
}
```

A

**LISTAGEM 12.23 ConcreteAccountFactory.java**

```
public class ConcreteAccountFactory implements AbstractAccountFactory {  
  
    public CheckingAccount createCheckingAccount( double initDeposit, int trans,  
double fee ) {  
        return new CheckingAccount( initDeposit, trans, fee );  
    }  
  
    public OverdraftAccount createOverdraftAccount( double initDeposit, double  
rate ) {  
        return new OverdraftAccount( initDeposit, rate );  
    }  
  
    public RewardsAccount createRewardsAccount( double initDeposit, double  
interest, double min ) {  
        return new RewardsAccount( initDeposit, interest, min );  
    }  
  
    public SavingsAccount createSavingsAccount( double initBalance, double  
interestRate ) {  
        return new SavingsAccount( initBalance, interestRate );  
    }  
  
    public TimedMaturityAccount createTimedMaturityAccount( double initBalance,  
double interestRate, double feeRate ){  
        return new TimedMaturityAccount( initBalance, interestRate, feeRate );  
    }  
}
```

## Dia 13 Respostas do teste e dos exercícios

### Respostas do teste

1. A análise, projeto e implementação de uma UI não são diferentes do restante do sistema. A UI deve ter consideração igual durante todas as fases do desenvolvimento. Antes de tudo, você deve certificar-se de não ignorar as considerações sobre a UI.
2. Você deve desacoplar as UIs para que o sistema e a UI não se tornem entrelaçados. É difícil fazer alterações na UI, quando entrelaçada com a funcionalidade básica do sistema. Também é impossível compartilhar o sistema com outras UIs ou tipos de UI, quando a UI e o sistema estão entrelaçados.
3. Os três componentes são o modelo, o modo de visualização e o controlador.
4. O padrão PAC e o Document/View Model são duas alternativas para o padrão MVC.
5. O modelo é a camada da tríade MVC que gerencia o comportamento básico e o estado do sistema. O controlador usa o modelo para instigar o comportamento do sistema. O modo de visualização usa o modelo para recuperar informações de estado para exibição. O modelo também fornece um mecanismo de notificação de alteração. O modo de visualização e o controlador podem usar esse mecanismo para estar a par das mudanças de estado no modelo.
6. O modo de visualização é o membro da tríade MVC responsável por apresentar o modelo para o usuário.
7. O controlador é responsável por interpretar os eventos gerados pelo usuário. O controlador instiga o comportamento no modelo ou no modo de visualização, em resposta a esses eventos.
8. Um sistema pode ter muitos modelos. Um modelo pode ter muitos modos de visualização diferentes. Um modo de visualização pode ter um controlador e um controlador pode controlar apenas um modo de visualização.
9. Ineficárias podem ser encontradas no modelo do modo de visualização e do controlador. Um modelo deve evitar notificações de alteração de estado desnecessárias. Os modos de visualização e controladores devem colocar os dados em cache, quando possível.
10. O padrão MVC supõe um modelo estável e uma apresentação variável.
11. Um resumo muito detalhado da história e da motivação por trás do padrão MVC está em “Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)”, de Steve Burbeck, Ph.D. Você pode encontrar uma cópia no endereço <http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html>

Então, qual é o objetivo dessa pergunta? Bem, a resposta proporciona a você uma perspectiva importante da motivação por trás do padrão MVC. Lendo a história, você também notará que o padrão MVC foi desenvolvido inicialmente como parte da Smalltalk. Agora seu uso é encontrado em quase qualquer linguagem. Isso traz um ponto importante: os padrões não são utilizações de linguagem — eles são padrões que funcionam em qualquer linguagem com os recursos requisitados.

A MVC não está relacionada com Java ou Smalltalk. Está relacionada a um projeto que transcende a linguagem de implementação.

A

## Respostas dos exercícios

1. A listagem 13.11 apresenta a nova classe Employee.

### LISTAGEM 13.11 Employee.java

```
import java.util.ArrayList;
import java.util.Iterator;

public abstract class Employee {

    private String first_name;
    private String last_name;
    private double wage;
    private ArrayList listeners = new ArrayList();

    public Employee(String first_name, String last_name, double wage) {
        this.first_name = first_name;
        this.last_name = last_name;
        this.wage = wage;
    }

    public double getWage() {
        return wage;
    }

    public void setWage( double wage ) {
        this.wage = wage;
        updateObservers();
    }

    public String getFirstName() {
        return first_name;
    }

    public String getLastNames() {
```

**LISTAGEM 13.11 Employee.java (continuação)**

---

```
        return last_name;
    }

    public abstract double calculatePay();

    public abstract double calculateBonus();

    public void printPaycheck() {
        String full_name = last_name + ", " + first_name;
        System.out.println( "Pay: " + full_name + " $ " + calculatePay() );
    }

    public void register( Observer o ) {
        listeners.add( o );
        o.update();
    }

    public void deregister( Observer o ) {
        listeners.remove( o );
    }

    private void updateObservers() {
        Iterator i = listeners.iterator();
        while( i.hasNext() ) {
            Observer o = (Observer) i.next();
            o.update();
        }
    }
}
```

---

2.

A Listagem 13.12 apresenta a nova implementação de BankAccountController.

**LISTAGEM 13.12 BankAccountController.java**

---

```
public class BankAccountController implements BankActivityListener {

    private BankAccountView view;
    private BankAccountModel model;

    public BankAccountController( BankAccountView view, BankAccountModel model )
    {
        this.view = view;
        this.modelo = model;
    }
```

**LISTAGEM 13.12 BankAccountController.java} (continuação)**

```
public void withdrawPerformed( BankActivityEvent e ) {  
    double amount = e.getAmount();  
    model.withdrawFunds( amount );  
}  
  
public void depositPerformed( BankActivityEvent e ) {  
    double amount = e.getAmount();  
    model.depositFunds( amount );  
}  
}
```

A

Esta versão de BankAccountController é muito mais fácil de ler do que a original; entretanto, o modo de visualização agora é muito mais complexo.

## Dia 14 Respostas do teste e dos exercícios

### Respostas do teste

1. Erros de digitação, erros na lógica ou enganos bobos cometidos durante a codificação podem surgir. Os erros também podem resultar da interação incorreta entre objetos ou de falhas no projeto ou na análise.

2. Um caso de teste é o bloco de construção dos testes. Cada forma de teste é constituída de casos de teste e cada caso de teste testa um aspecto do sistema.

3. Você pode basear seus casos de teste em teste de caixa preta ou caixa branca.

4. Os testes de caixa branca são baseados na estrutura do código-fonte subjacente. Os testes de caixa branca são projetados para atingir 100% de cobertura do código testado.

Os testes de caixa preta são baseados na especificação. Os testes de caixa preta verificam se o sistema se comporta conforme o esperado.

5. As quatro formas de teste são: *teste de unidade*, *teste de integração*, *teste de sistema* e *teste de regressão*.

6. Um teste de unidade é o dispositivo de teste de nível mais baixo. Um teste de unidade envia uma mensagem para um objeto e verifica se recebe o resultado esperado. Um teste de unidade só deve verificar um recurso por vez.

7. O teste de integração confirma se os objetos interagem corretamente. O teste de sistema verifica se o sistema se comporta conforme definido nos casos de uso e se ele pode manipular uso imprevisto normalmente.

8. Você não deve deixar os testes para o fim. Testar enquanto você desenvolve o ajuda a encontrar erros imediatamente. Se você deixar os testes para o final, haverá mais erros e eles serão mais difíceis de rastrear e corrigir.

Testar enquanto você desenvolve também torna mais fácil alterar seu código e pode melhorar seu projeto.

9. A validação manual ou visual é propensa a erros. Você deve evitá-la o máximo possível. Em vez disso, você deve contar com um mecanismo automático para validação dos testes de unidade.
10. Uma estrutura define um modelo de domínio reutilizável. Você pode usar as classes desse modelo como base para seu aplicativo específico.
11. Um objeto falsificado é um substituto simplista de um objeto real, que o ajuda a fazer o teste de unidade de seus objetos.
12. Os objetos falsificados permitem que você faça o teste de unidade de suas classes isoladamente. Eles também abrem possibilidades de teste que, de outra forma, seriam difíceis ou impossíveis de fazer.
13. Um erro surge de uma falha ou de um defeito no sistema. Uma condição de erro, por outro lado, não é um erro, mas sim uma condição para a qual seu sistema deve estar preparado e deve manipular normalmente.
14. Ao escrever seu código, você pode garantir a qualidade através dos testes de unidade, do tratamento correto de exceções e através de documentação correta.

## Respostas dos exercícios

1. Cookstour dará idéias sobre o projeto e as noções por trás de JUnit.
2. A Listagem 14.12 apresenta um possível teste de unidade.

### **LISTAGEM 14.12 HourlyEmployeeTest.java**

---

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class HourlyEmployeeTest extends TestCase {

    private HourlyEmployee emp;

    private static final String FIRST_NAME = "FNAME";
    private static final String LAST_NAME   = "LNAME";
    private static final double WAGE        = 500.00;

    protected void setUp() {
```

**LISTAGEM 14.12 HourlyEmployeeTest.java (continuação)**

```
    emp = new HourlyEmployee( FIRST_NAME, LAST_NAME, WAGE );  
}  
  
public void test_calculatePay() {  
    emp.addHours( 10 );  
  
    double expected = WAGE * 10;  
    assertTrue( "incorrect pay calculation", emp.calculatePay() == expected );  
}  
  
public HourlyEmployeeTest( String name ) {  
    super( name );  
}  
}
```

A

## Dia 15 Respostas do teste e dos exercícios

### Respostas do teste

1. PlayerListener é um exemplo do padrão observável.  
Console é singleton. Ele implementa o padrão singleton.  
Rank e Suit implementam o padrão Typesafe Enum.
2. BlackjackDealer trata de HumanPlayer de forma polimórfica como um Player. Você poderia criar jogadores não-humanos e BlackjackDealer saberia como jogar vinte-e-um com eles.
3. Player/BlackjackDealer/HumanPlayer é um exemplo de hierarquia de herança.
4. Deck encapsula completamente os objetos Card que contém. Ele não fornece quaisquer métodos de obtenção ou configuração. Em vez disso, Deck adiciona seus objetos Card em objetos Deckpile.
5. BlackjackDealer e HumanPlayer atuam de forma polimórfica, fornecendo suas próprias versões personalizadas do método hit(). Quando o método play() chamar hit(), o comportamento do método play() variará de acordo com a implementação subjacente de hit().

### Respostas dos exercícios

1. Sem resposta.
2. Sem resposta.

# Dia 16 Respostas do teste e dos exercícios

## Respostas do teste

1. As estruturas condicionais são perigosas quando retiram responsabilidade de um objeto e a colocam em outro lugar. O comportamento pertence ao objeto e não é distribuído por todo o programa. Uma lógica distribuída o obriga a repetir lógica por todo o seu programa, em vez de tê-la em apenas um lugar.

As estruturas condicionais também são perigosas porque tornam mais difícil testar um objeto e cobrir todas as combinações de caminhos através do objeto.

2. Anteriormente, você viu que podia usar polimorfismo para remover estruturas condicionais.

Hoje, você viu que pode usar uma combinação de polimorfismo e estado para remover estruturas condicionais. O estado é um modo excelente de implementar regras.

3. A versão anterior de Hand exigia que você mesmo comparasse os objetos Card de Hand para verificar se dois objetos Hand são iguais ou se um objeto Hand é maior do que outro. Agora, Hand faz essa verificação para você, sem comprometer seu estado interno.

Hand faz outra coisa para se encapsular. Agora, Hand informa aos receptores de mudanças de estado. Como Hand coloca prontamente suas informações de estado nos receptores, não há motivo para que um objeto interessado faça uma consulta em Hand para conhecer seu estado.

4. Hand e HandListener implementam padrão observador.

5. Sem resposta.

## Respostas dos exercícios

1. Sem resposta.
2. O segredo desse problema é perceber que todos os métodos levam as mesmas ações para cima, até o ponto onde uma chamada é feita em PlayerListener. A solução é empacotar essa chamada em um objeto.

Considere o método `notifyListener()` a seguir:

```
protected void notifyListeners( NotifyHelper helper ) {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = ( PlayerListener )i.next();
        helper.notifyListener( pl );
    }
}
```

Note que esse método é exatamente igual ao antigo `notifyChanged()` ou `notifyBusted()`, exceto quanto a uma diferença. Em vez de chamar um método diretamente em `PlayerListener`, o método `notifyListeners()` delega a chamada para um objeto `NotifyHelper`.

A Listagem 16.17 apresenta `NotifyHelper`.

---

**LISTAGEM 16.17** O estado de espera da banca personalizado

```
protected interface NotifyHelper {  
    public void notifyListener( PlayerListener p1 );  
}
```

A

A interface `NotifyHelper` define um método: `notifyListener()`. Os implementadores decidirão qual método vão chamar em `PlayerListener`.

Ao todo, você precisará definir sete implementadores de `NotifyHelper`, uma implementação para cada método na interface `PlayerListener`. A Listagem 16.18 apresenta essas sete implementações.

---

**LISTAGEM 16.18** As implementações de `NotifyHelper`.

```
protected class NotifyBusted implements NotifyHelper {  
    public void notifyListener( PlayerListener p1 ) {  
        p1.playerBusted( Player.this );  
    }  
}  
protected class NotifyBlackjack implements NotifyHelper {  
    public void notifyListener( PlayerListener p1 ) {  
        p1.playerBlackjack( Player.this );  
    }  
}  
protected class NotifyWon implements NotifyHelper {  
    public void notifyListener( PlayerListener p1 ) {  
        p1.playerWon( Player.this );  
    }  
}  
protected class NotifyLost implements NotifyHelper {  
    public void notifyListener( PlayerListener p1 ) {  
        p1.playerLost( Player.this );  
    }  
}  
protected class NotifyChanged implements NotifyHelper {  
    public void notifyListener( PlayerListener p1 ) {  
        p1.playerChanged( Player.this );  
    }  
}  
protected class NotifyStanding implements NotifyHelper {  
    public void notifyListener( PlayerListener p1 ) {  
        p1.playerStanding( Player.this );  
    }  
}  
protected class NotifyStandoff implements NotifyHelper {
```

**LISTAGEM 16.18** As implementações de NotifyHelper. (continuação)

```
public void notifyListener( PlayerListener pl ) {  
    pl.playerStandoff( Player.this );  
}
```

---

Agora, em vez de chamar `notifyChanged()` ou `notifyBlackjack()`, você chamaria `notifyListeners( new NotifyChanged() )` ou `notifyListeners( new NotifyBlackjack() )`.

Se você acha que essa é uma boa solução ou não, é uma questão de gosto pessoal. Entretanto, ela remove os métodos `notifyXXX` redundantes.

## Dia 17 Respostas do teste e dos exercícios

### Respostas do teste

1. Tornar abstrato um método protegido é uma boa maneira de estabelecer um protocolo de herança.
2. Após a análise e o projeto de hoje, uma nova hierarquia Player foi descoberta. Como os requisitos se apresentaram através dos casos de uso, a necessidade de uma nova hierarquia de herança se apresentou.
3. A programação por especulação raramente funciona. As hierarquias que você precisará para modelar corretamente um domínio abstrato se apresentarão após você ter trabalhado com um domínio por algum tempo. Se você tentar abstrair um domínio sem trabalhar com esse domínio, estará supondo uma solução.
4. Refazendo a hierarquia, você ficou com um modelo que representa mais detalhadamente o jogo vinte-e-um. O código também é mais fácil de entender. Colocar responsabilidade extra na classe base Player teria resultado em um código difícil de entender.

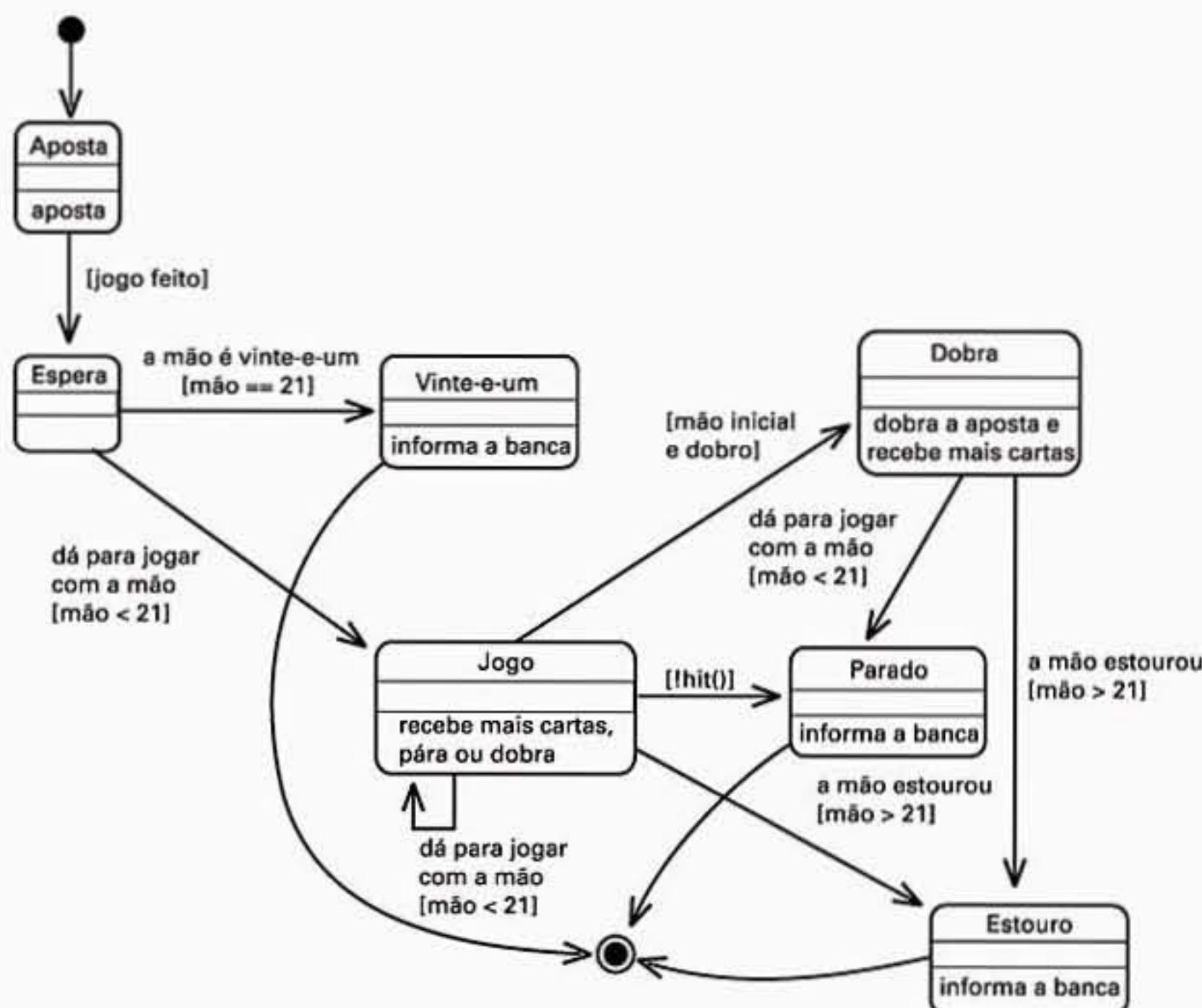
### Respostas dos exercícios

1. Sem resposta.
2. Dobro é apenas outro estado de BettingPlayer. Você sabe que ele precisa ser um estado separado, porque Player deve reagir de uma forma diferente ao evento handPlayable. Em vez de permanecer no estado *Jogo*, Player deve fazer a transição para o estado *Parado*.

A Figura A.7 ilustra o novo diagrama de estado para um objeto BettingPlayer que pode dobrar.

**FIGURA A.7**

O diagrama de estado de BettingPlayer atualizado.



A

Você também precisa alterar o estado *Jogo* para que ele possa fazer a transição para o estado *Dobra*. A Listagem 17.6 apresenta os novos estados de BettingPlayer.

#### **LISTAGEM 17.6** DoublingDown e Jogo

```

private class DoublingDown implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    public void handBlackjack() {
        // impossível no estado de dobro
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        bank.doubleDown();
        dealer.hit( BettingPlayer.this );
        getCurrentState().execute( dealer );
    }
}
  
```

**LISTAGEM 17.6** DoublingDown e Jogo (*continuação*)

```

}
private class BetterPlaying implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // pode ignorar no estado de jogo
    }
    public void handBlackjack() {
        // impossível no estado de jogo
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        if( getHand().canDoubleDown() &&doubleDown() ) {
            setCurrentState( getDoublingDownState() );
            getCurrentState().execute( dealer );
            return;
        }
        if( hit() ) {
            dealer.hit( BettingPlayer.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        getCurrentState().execute( dealer );
        // transição
    }
}

```

Você também precisará atualizar `HumanPlayer` para que ofereça a opção de dobrar. A Listagem 17.7 apresenta a classe `HumanPlayer` atualizada.

**LISTAGEM 17.7** HumanPlayer.java

```

public class HumanPlayer extends BettingPlayer {

    private final static String HIT      = "H";
    private final static String STAND    = "S";
    private final static String PLAY_MSG = "[H]it or [S]tay";
    private final static String BET_MSG  = "Place Bet: [10] [50] or [100]";
    private final static String DD_MSG   = "Double Down? [Y]es [N]o";

```

**LISTAGEM 17.7** HumanPlayer.java (*continuação*)

```
private final static String BET_10 = "10";
private final static String BET_50 = "50";
private final static String BET_100 = "100";
private final static String NO = "N";
private final static String YES = "Y";
private final static String DEFAULT = "invalid";

public HumanPlayer( String name, Hand hand, Bank bank ) {
    super( name, hand, bank );
}

protected boolean hit() {
    while( true ) {
        Console.INSTANCE.printMessage( PLAY_MSG );
        String response = Console.INSTANCE.readInput( DEFAULT );
        if( response.equalsIgnoreCase( HIT ) ) {
            return true;
        }else if( response.equalsIgnoreCase( STAND ) ) {
            return false;
        }
        // se chegarmos até aqui, faz um laço até obtermos entrada
        // significativa
    }
}

protected boolean doubleDown() {
    while( true ) {
        Console.INSTANCE.printMessage( DD_MSG );
        String response = Console.INSTANCE.readInput( DEFAULT );
        if( response.equalsIgnoreCase( NO ) ) {
            return false;
        } else if( response.equalsIgnoreCase( YES ) ) {
            return true;
        }
        // se chegarmos até aqui, faz um laço até obtermos entrada
        // significativa
    }
}

protected void bet() {
while( true ) {
    Console.INSTANCE.printMessage( BET_MSG );
    String response = Console.INSTANCE.readInput( DEFAULT );
    if( response.equals( BET_10 ) ) {
        getBank().place10Bet();
```

A

**LISTAGEM 17.7** HumanPlayer.java (*continuação*)

```
        return;
    }
    if( response.equals( BET_50 ) ) {
        getBank().place50Bet();
        return;
    }
    if( response.equals( BET_100 ) ) {
        getBank().place100Bet();
        return;
    }
    // se chegarmos até aqui, faz um laço até obtermos entrada
    // significativa
}
}
```

---

## Dia 18 Respostas do teste e dos exercícios

### Respostas do teste

1. Para introduzir um Vcard, você pode simplesmente fazer uma subclasse de Card. Para colocá-la no jogo, você pode fazer uma subclasse de Deck e, em seguida, sobrepor buildCards para que a subclasse crie objetos VCard, em vez de objetos Card.
2. Originalmente, o método buildCards de Deck era privado . Quando verificamos que uma subclasse precisava sobrepor o comportamento de buildCards, o tornamos protegido.

### Respostas dos exercícios

1. Sem resposta.
2. O projeto e implementação iniciais que fizemos para o Exercício 2 do Capítulo 17 ainda são válidos. Dobrar é apenas outro estado em BettingPlayer. Você deve começar escrevendo o código para este exercício, adicionando o novo estado DoublingDown em BettingPlayer. Você também precisará fazer as mesmas alterações feitas no Capítulo 17, nas classes Bank e Hand. Quando fizer essas alterações, você precisará atualizar GUIPlayer, assim como OptionView e OptionViewController.

Antes de fazer as alterações exigidas nas classes de modo de visualização e de controlador, examinaremos as alterações exigidas em BettingPlayer, Hand e Bank.

A Listagem 18.13 destaca as alterações que foram feitas na classe Hand.

**LISTAGEM 18.13** Destaques das alterações feitas em Hand

```
public boolean canDoubleDown() {  
    return ( cards.size() == 2 );  
}
```

O novo método `canDoubleDown` permite que `BettingPlayer` verifique o objeto `Hand` para ver se o jogador pode dobrar.

Você também precisa adicionar um novo método `doubleDown` na classe `Bank`. A Listagem 18.14 apresenta esse novo método.

**LISTAGEM 18.14** O novo método `doubleDown`

```
public void doubleDown() {  
    placeBet( bet );  
    bet = bet * 2;  
}
```

O método `doubleDown` dobra a aposta corrente.

Para poder adicionar a função de dobrar, você precisa adicionar um novo estado em `BettingPlayer`. Você sabe que precisa de um novo estado porque `BettingPlayer` tem de tratar do evento `handPlayable`, especialmente ao se dobrar. Normalmente, um evento `handPlayable` significa que o jogador pode receber mais cartas novamente, se quiser. Ao dobrar, o jogador deve parar imediatamente, após a operação (caso ele não estoure). A Listagem 18.15 apresenta o novo estado de `DoubleDown`.

**LISTAGEM 18.15** The New DoubleDown State

```
private class DoublingDown implements PlayerState {  
    public void handChanged() {  
        notifyChanged();  
    }  
    public void handPlayable() {  
        setCurrentState( getStandingState() );  
        notifyStanding();  
    }  
    public void handBlackjack() {  
        // impossível no estado de dobro  
    }  
    public void handBusted() {  
        setCurrentState( getBustedState() );  
        notifyBusted();  
    }  
    public void execute( Dealer dealer ) {
```

A

**LISTAGEM 18.15** The New DoubleDown State (*continuação*)

```
    bank.doubleDown();
    dealer.hit( BettingPlayer.this );
    getCurrentState().execute( dealer );
}
}
```

---

Quando executado, esse novo estado dirá ao objeto Bank para que duplique a aposta, pedirá à banca para que distribua mais cartas para o jogador e, então, fará a transição para o próximo estado. O estado seguinte será parado ou estouro, dependendo do evento enviado para o estado por Hand.

Para obter o estado DoublingDown (Dobro), você precisará fazer algumas alterações no estado Jogo. A Listagem 18.16 apresenta o novo estado BetterPlaying.

**LISTAGEM 18.16** O novo estado BetterPlaying

```
private class BetterPlaying implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // pode ignorar no estado de jogo
    }
    public void handBlackjack() {
        // impossível no estado de jogo
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        if( getHand().canDoubleDown() &&doubleDown() ) {
            setCurrentState( getDoublingDownState() );
            getCurrentState().execute( dealer );
            return;
        }
        if( hit() ) {
            dealer.hit( BettingPlayer.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
        getCurrentState().execute( dealer );
        // transição
    }
}
```

---

Quando executado, o estado BetterPlaying primeiro verifica se o jogador pode dobrar. Se assim for, o estado chamará o método doubleDown de BettingPlayer (você precisará adicionar um método abstrato doubleDown em BettingPlayer). Se o método indicar que o jogador gostaria de dobrar, o estado BetterPlaying configurará o estado corrente como DoublingDown e fará a transição para ele.

Se o jogador não quiser dobrar, o estado continuará e jogará normalmente. Veja todas as alterações no código-fonte. Existem algumas outras pequenas alterações, como a adição de um método getDoubleDownState em BettingPlayer. Esse método efetivamente adiciona a possibilidade de dobro no sistema.

Agora, você precisa atualizar GUIPlayer, OptionView e OptionViewController.

A boa nova é que você não precisa fazer quaisquer alterações nos estados de GUIPlayer. Esses estados não devem fazer nada, pois eles precisam esperar até que o jogador humano clique em um botão. Você precisa implementar o método doubleDown. A Listagem 18.17 apresenta o método.

#### **LISTAGEM 18.17** O método doubleDown

```
protected boolean doubleDown() {  
    setCurrentState( getDoublingDownState() );  
    getCurrentState().execute( dealer );  
    return true;  
}
```

O botão da GUI pode chamar esse método, caso o usuário decida dobrar. O método configura o estado corrente como o dobro e, em seguida, faz a transição para ele. O estado manipula o restante.

Todas as alterações restantes precisam entrar em OptionView e OptionViewController. Você precisa principalmente adicionar um botão de dobro no modo de visualização e garantir que ele seja ativado pelo controlador imediatamente após a aposta e desativado assim que o jogador pressionar esse botão, o botão de parada ou o de receber mais cartas.

Não se preocupe muito, se você não entender totalmente o código da GUI. O importante é que você entenda como a operação de dobrar é adicionada no sistema e as idéias básicas por trás do modo de visualização e do controlador.

## **Dia 19 Respostas do teste e dos exercícios**

### **Respostas do teste**

1. As três camadas são: *apresentação, abstração e controle*.

A

2. A apresentação é responsável por exibir a abstração, assim como por responder à interação do usuário.

A abstração é semelhante ao modelo na MVC. A abstração representa o sistema básico.

O controle é responsável por pegar as diversas apresentações e combiná-las em um modo de visualização.

3. Você pode usar herança para fazer uma subclasse de cada uma das classes de sistema que exigirão uma apresentação. Desse modo, você não enxerta uma classe de apresentação diretamente na classe de sistema. Em vez disso, você pode adicionar a classe de apresentação na subclasse.

Usando herança dessa maneira, você pode fornecer vários modos de visualização do mesmo sistema. Sempre que você precisar de um modo de visualização diferente, bastará fazer uma subclasse das classes que precisa exibir e fazê-las criar uma nova apresentação. Quando você precisar reunir todas as classes como um aplicativo, bastará garantir a instanciação das subclasses corretas.

4. É melhor usar o PAC em um sistema estável com requisitos de interface bem definidos. Às vezes, a estratégia da herança pode falhar para projetos mais complicados. Quando a herança falhar, você terá de enxertar a apresentação diretamente na classe de sistema. Tal eventualidade torna difícil servir muitos modos de visualização diferentes.
5. Você conseguiu manter duas UIs porque deixou a estrutura de observador intacta. Nada o obriga a remover a estrutura, apenas porque você utiliza PAC. Na verdade, BettingView e DealerView usam o mecanismo PlayerListener para ficar a par das alterações no jogador e na banca.
6. O padrão factory foi usado para garantir que as instâncias de classe corretas fossem usadas juntas. Em particular, você deve tomar o cuidado de usar um objeto VDeck quando utilizar as classes que criam uma apresentação.

## Respostas dos exercícios

1. Sem resposta.
2. O projeto e a implementação iniciais que você realizou para o Exercício 2 do Capítulo 17 ainda são válidos. Dobro é apenas outro estado em BettingPlayer. Você deve começar a escrever o código deste exercício, adicionando o novo estado DoublingDown em BettingPlayer. Você também precisará fazer as mesmas alterações que fez no Capítulo 17, nas classes Bank e Hand. Quando você fizer essas alterações, precisará atualizar GUIPlayer e sua classe de apresentação.

Antes de fazer as alterações exigidas nas classes da GUI, vamos examinar as alterações exigidas em BettingPlayer, Hand e Bank.

A Listagem 19.8 destaca as alterações que foram feitas na classe Hand.

#### **LISTAGEM 19.8** Destaques das alterações feitas em Hand

```
public boolean canDoubleDown() {  
    return ( cards.size() == 2 );  
}
```

A

O novo método `canDoubleDown` permite que `BettingPlayer` verifique o objeto `Hand` para ver se o jogador pode dobrar.

Você também precisa adicionar um novo método `doubleDown` na classe `Bank`. A Listagem 19.9 apresenta esse novo método.

#### **LISTAGEM 19.9** O novo método `doubleDown`

```
public void doubleDown() {  
    placeBet( bet );  
    bet = bet * 2;  
}
```

O método `doubleDown` dobra a aposta corrente.

Para adicionar a aposta em dobro, você precisa adicionar um novo estado em `BettingPlayer`. Você sabe que precisa de um novo estado, porque `BettingPlayer` tem de tratar do evento `handPlayable` de uma maneira especial quando se dobra. Normalmente, um evento `handPlayable` significa que o jogador pode receber mais cartas novamente, se quiser. Ao dobrar, o jogador deve parar imediatamente após tê-lo feito (caso ele não estoure). A Listagem 19.10 apresenta o novo estado `DoubleDown`.

#### **LISTAGEM 19.10** O novo estado `DoubleDown`

```
private class DoublingDown implements PlayerState {  
    public void handChanged() {  
        notifyChanged();  
    }  
    public void handPlayable() {  
        setCurrentState( getStandingState() );  
        notifyStanding();  
    }  
    public void handBlackjack() {  
        // impossível no estado de dobro  
    }  
    public void handBusted() {  
        setCurrentState( getBustedState() );  
    }
```

**LISTAGEM 19.10** O novo estado DoubleDown (*continuação*)

```
    notifyBusted();
}
public void execute( Dealer dealer ) {
    bank.doubleDown();
    dealer.hit( BettingPlayer.this );
    getCurrentState().execute( dealer );
}
}
```

---

Quando executado, esse novo estado diz a Bank para que duplique a aposta, pede à banca para que distribua mais cartas para o jogador e, em seguida, faz a transição para o próximo estado. O estado seguinte será parado ou vai estourar, dependendo do evento que for enviado para o estado por Hand.

Para obter o estado DoublingDown, você precisará fazer algumas alterações no estado Jogo. A Listagem 19.11 apresenta o novo estado BetterPlaying.

**LISTAGEM 19.11** O novo estado Jogo

```
private class BetterPlaying implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        // pode ignorar no estado de jogo
    }
    public void handBlackjack() {
        // impossível no estado de jogo
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        if( getHand().canDoubleDown() && doubleDown() ) {
            setCurrentState( getDoublingDownState() );
            getCurrentState().execute( dealer );
            return;
        }
        if(hit() ) {
            dealer.hit( BettingPlayer.this );
        } else {
            setCurrentState( getStandingState() );
            notifyStanding();
        }
    }
}
```

**LISTAGEM 19.11** O novo estado Jogo (*continuação*)

```
    }
    getCurrentState().execute( dealer );
    // transição
}
}
```

A

Quando executado, o estado BetterPlaying primeiro verifica se o jogador pode dobrar. Se assim for, o estado chamará o método doubleDown de BettingPlayer (você precisará adicionar um método abstrato doubleDown em BettingPlayer). Se o método indicar que o jogador gostaria de dobrar, o estado BetterPlaying configurará o estado corrente como DoublingDown e depois fará a transição para ele.

Se o jogador não quiser dobrar, o estado continuará e jogará normalmente. Veja todas as alterações no código-fonte; existem algumas outras pequenas alterações, como a adição de um método getDoubleDownState em BettingPlayer.

Você efetivamente adicionou a capacidade de dobrar no sistema. Agora, você precisa atualizar GUIPlayer e sua classe de apresentação, para que possa suportar a aposta em dobro.

A boa nova é que você não precisa fazer quaisquer alterações nos estados de GUIPlayer. Esses estados não fazem nada, pois eles precisam esperar até que o jogador humano clique em um botão. Você precisa implementar o método doubleDown. A Listagem 19.12 apresenta o método.

**LISTAGEM 19.12** O método doubleDown

```
protected boolean doubleDown() {
    setCurrentState( getDoublingDownState() );
    getCurrentState().execute( dealer );
    return true;
}
```

O botão da GUI pode chamar esse método, caso o usuário decida dobrar. O método configura o estado corrente como dobro e, em seguida, faz a transição para ele. O estado trata do resto.

Todas as alterações restantes precisam entrar na classe de apresentação: GUIView. Você precisa adicionar um botão de dobro, certificar-se de que ele seja ativado após a aposta e desativado assim que o jogador pressionar esse botão, o botão de parar ou de receber mais cartas.

Não se preocupe muito, caso você não entenda totalmente o código da GUI. O importante é que você entenda como a capacidade de dobrar é adicionada no sistema.

## Dia 20 Respostas do teste e dos exercícios

### Respostas do teste

1. Através dos relacionamentos com capacidade de substituição e do polimorfismo, você pode criar qualquer subclasse de BettingPlayer que queira e adicioná-la ao jogo.

O jogo não sabe a diferença entre um jogador humano e um jogador automático não-humano. Como resultado, você pode configurar o jogo sem nenhum jogador humano. Implementando o método `hit` nas subclasses de `Player`, elas poderão jogar sem intervenção humana.

2. Você nunca deve seguir a estratégia de `OneHitPlayer`.

### Respostas dos exercícios

1. Sem resposta.
2. Suas soluções podem variar. As listagens 20.8 e 20.9 apresentam a implementação de `KnowledgeablePlayer` e `OptimalPlayer`, respectivamente.

#### **LISTAGEM 20.8** KnowledgeablePlayer.java

---

```
public class KnowledgeablePlayer extends BettingPlayer {  
  
    public KnowledgeablePlayer(String name, Hand hand, Bank bank) {  
        super(name, hand, bank);  
    }  
  
    public boolean hit(Dealer dealer) {  
  
        int total = getHand().total();  
        Card card = dealer.getUpCard();  
  
        // nunca recebe mais cartas, não importa qual, se total > 15  
        if( total > 15 ) {  
            return false;  
        }  
  
        // sempre recebe mais cartas para 11 e menos  
        if( total <= 11 ) {  
            return true;  
        }  
    }  
}
```

**LISTAGEM 20.8** KnowledgeablePlayer.java (*continuação*)

```
// isso deixa 11, 12, 13, 14  
// baseia a decisão na banca  
  
if( card.getRank().getRank() > 7 ) {  
    return true;  
}  
  
return false;  
}  
  
public void bet() {  
    getBank().place10Bet();  
}  
}
```

A

**LISTAGEM 20.9** OptimalPlayer.java

```
public class OptimalPlayer extends BettingPlayer {  
  
    public OptimalPlayer( String name, Hand hand, Bank bank ) {  
        super( name, hand, bank );  
    }  
  
    public boolean hit( Dealer dealer ) {  
  
        int total = getHand().total();  
        Card card = dealer.getUpCard();  
  
        if( total >= 17 ) {  
            return false;  
        }  
  
        if( total == 16 ) {  
            if( card.getRank() == Rank.SEVEN ||  
                card.getRank() == Rank.EIGHT ||  
                card.getRank() == Rank.NINE ) {  
                return true;  
            } else {  
                return false;  
            }  
        }  
        if( total == 13 || total == 14 || total == 15 ) {  
            if( card.getRank() == Rank.TWO ||  
                card.getRank() == Rank.THREE ||
```

**LISTAGEM 20.9** OptimalPlayer.java (*continuação*)

```
        card.getRank() == Rank.FOUR ||
        card.getRank() == Rank.FIVE ||
        card.getRank() == Rank.SIX ) {
    return false;
} else {
    return true;
}
}
if( total == 12 ) {
    if( card.getRank() == Rank.FOUR ||
        card.getRank() == Rank.FIVE ||
        card.getRank() == Rank.SIX ) {
    return false;
} else {
    return true;
}
}
return true;
}

public void bet() {
    getBank().place10Bet();
}

}
```

---

Então, como esses jogadores se comportam?

Em nossos testes, KnowledgeablePlayer e OptimalPlayer executam melhor do que SmartPlayer, apresentado no capítulo. Comparando um com outro, OptimalPlayer executa melhor.

Contudo, com o passar do tempo, ambos ainda perdem dinheiro, apenas que muito lentamente.

3. Suas soluções podem variar. As listagens 20.10 e 20.11 apresentam a implementação de KnowledgeablePlayer e OptimalPlayer, respectivamente.

**LISTAGEM 20.10** KnowledgeablePlayer.java

```
public class KnowledgeablePlayer extends BettingPlayer {

    public KnowledgeablePlayer(String name, Hand hand, Bank bank) {
        super( name, hand, bank );
    }
```

**LISTAGEM 20.10** KnowledgeablePlayer.java (*continuação*)

```
public boolean doubleDown( Dealer d ) {  
    int total = getHand().total();  
    if( total == 10 || total == 11 ) {  
        return true;  
    }  
    return false;  
}  
  
public boolean hit( Dealer d ) {  
  
    int total = getHand().total();  
    Card c = d.getUpCard();  
  
    // nunca recebe mais cartas, não importa qual, se total > 15  
    if( total > 15 ) {  
        return false;  
    }  
  
    // sempre recebe mais cartas para 11 e menos  
    if( total <= 11 ) {  
        return true;  
    }  
  
    // isso deixa 11, 12, 13, 14  
    // baseia a decisão na banca  
  
    if( c.getRank().getRank() > 7 ) {  
        return true;  
    }  
  
    return false;  
}  
  
public void bet() {  
    getBank().place10Bet();  
}  
}
```

A

**LISTAGEM 20.11** OptimalPlayer.java

```
public class OptimalPlayer extends BettingPlayer {  
  
    public OptimalPlayer( String name, Hand hand, Bank bank ) {  
        super( name, hand, bank );
```

**LISTAGEM 20.11 OptimalPlayer.java (*continuação*)**

```
}

public boolean doubleDown( Dealer d ) {
    int total = getHand().total();
    Card c = d.getUpCard();
    if( total == 11 ) {
        return true;
    }
    if( total == 10 ) {
        if( c.getRank().getRank() != Rank.TEN.getRank() &&
            c.getRank() != Rank.ACE ) {
            return true;
        }
        return false;
    }
    if( total == 9 ) {
        if( c.getRank() == Rank.TWO ||
            c.getRank() == Rank.THREE ||
            c.getRank() == Rank.FOUR ||
            c.getRank() == Rank.FIVE ||
            c.getRank() == Rank.SIX ) {
            return true;
        }
        return false;
    }
    return false;
}

public boolean hit( Dealer d ) {

    int total = getHand().total();
    Card c = d.getUpCard();

    if( total >= 17 ) {
        return false;
    }

    if( total == 16 ) {
        if( c.getRank() == Rank.SEVEN ||
            c.getRank() == Rank.EIGHT ||
            c.getRank() == Rank.NINE ) {
            return true;
        } else {
            return false;
        }
    }
}
```

**LISTAGEM 20.11 OptimalPlayer.java (continuação)**

```
    }
    if( total == 13 || total == 14 || total == 15 ) {
        if( c.getRank() == Rank.TWO ||
            c.getRank() == Rank.THREE ||
            c.getRank() == Rank.FOUR ||
            c.getRank() == Rank.FIVE ||
            c.getRank() == Rank.SIX ) {
            return false;
        } else {
            return true;
        }
    }
    if( total == 12 ) {
        if( c.getRank() == Rank.FOUR ||
            c.getRank() == Rank.FIVE ||
            c.getRank() == Rank.SIX ) {
            return false;
        } else {
            return true;
        }
    }
    return true;
}

public void bet() {
    getBank().place10Bet();
}
}
```

A

Em nossos testes, KnowledgeablePlayer e OptimalPlayer executam melhor do que as versões do Exercício 2.

Contudo, com o passar do tempo, ambos ainda perdem dinheiro — apenas muito lentamente.

## **Dia 21 Respostas do teste e dos exercícios**

### **Respostas do teste**

1. Você pode cuidar do problema da chamada de método recursiva colocando cada jogador em uma linha de execução. Quando você chamar Thread.start, a chamada retornará imediatamente, ao contrário de um método normal. Como o método retorna imediatamente, a pilha de chamada de métodos pode desenrolar e retornar.

## Respostas dos exercícios

1. Sem resposta.
2. As respostas dependerão dos interesses pessoais. O Apêndice D, “Bibliografia selecionada”, apresenta uma lista de recursos excelente, na qual você pode basear a continuação de seus estudos.

# APÊNDICE B

## Resumo do Java

### O Java Developer's Kit: J2SE 1.3 SDK

O JDK (Java Developer's Kit) da Sun Microsystems fornece o ambiente para todo o desenvolvimento Java. Com o passar dos anos, a Sun mudou o nome do kit de desenvolvimento de JDK para Java 2 Standard Edition (J2SE) Software Development Kit (SDK; entretanto, os objetivos das ferramentas e das bibliotecas permanecem os mesmos — ajudar os desenvolvedores em seus esforços para escrever software de qualidade, independente da plataforma, e orientado a objetos).

Muitos ambientes de desenvolvimento integrados (IDEs) populares incorporam o SDK, além de um editor e depurador poderoso. O Forte da Sun e o JBuilder da Borland fornecem gratuitamente versões mais simples do IDE; entretanto, para o objetivo desta discussão, a abrangência será limitada ao uso do SDK com um editor de textos como o TextPad, Notepad ou vi.

A Sun fornece o J2SE SDK para várias plataformas:

- Windows NT, 2000, 95, 98, ME
- Sun Solaris
- Linux

Você pode obter o SDK para outras plataformas, como HP ou AIX, com o fornecedor da plataforma apropriada. O J2SE SDK para Windows será utilizado como exemplo. A instalação e configuração em outras plataformas vai variar pouco em relação aos procedimentos discutidos a seguir.

Os desenvolvedores podem fazer download do J2SE SDK no site Web J2SE da JavaSoft, no endereço [java.sun.com/j2se/1.3](http://java.sun.com/j2se/1.3). Siga os links apropriados para fazer download do J2SE 1.3 SDK. Além do J2SE SDK, você deve fazer download da documentação da API J2SE. Embora não seja exigida, a documentação da API é tremendamente útil. A documentação da API J2SE fornece documentação detalhada de atributo, método e classe que até os desenvolvedores de Java altamente experientes considerarão úteis.

## Configuração do ambiente de desenvolvimento

A JavaSoft empacota o Windows J2SE 1.3 SDK em um pacote InstallShield. Quando você fizer download do arquivo, execute-o e percorra as caixas de diálogo para instalar o SDK no diretório de destino apropriado. (O diretório de destino padrão é C:\jdk1.3.) Instale todos os componentes, quando solicitado. Isso exigirá cerca de 54MB de espaço no disco rígido. Quando terminar, você deverá ver a seguinte estrutura de diretório em seu diretório de destino de instalação:

```
C:\jdk1.3
  \bin
  \demo
  \include
  \include-old
  \jre
  \lib
```

O pacote de instalação apenas distribui o SDK nos diretórios apropriados. Para iniciar o desenvolvimento, você deve configurar as seguintes variáveis de ambiente.

- JAVA\_HOME
- PATH
- CLASSPATH

Primeiro, configure JAVA\_HOME com o diretório instalado apropriado. Por exemplo, você executaria o seguinte na linha de comando:

```
set JAVA_HOME=c:\jdk1.3
```

Em seguida, configure o PATH:

```
set PATH=%PATH%;%JAVA_HOME%\bin
```

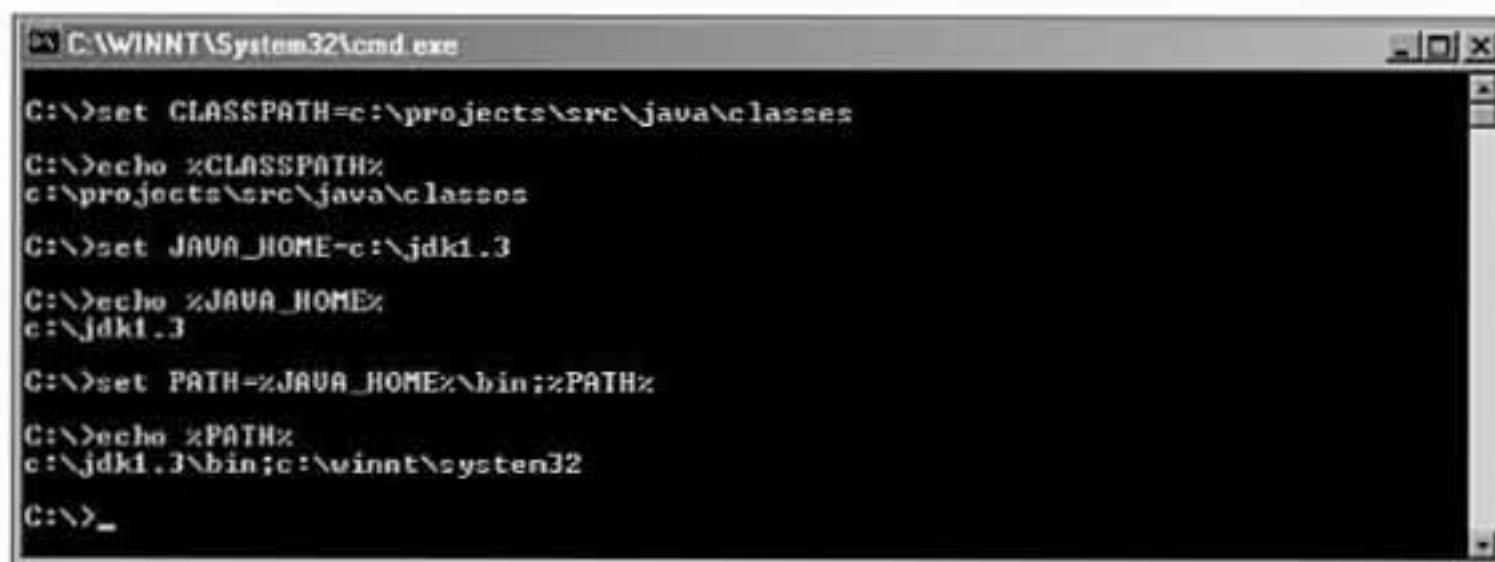
Finalmente, configure o CLASSPATH. O CLASSPATH informará o compilador e a máquina virtual onde deve procurar os arquivos de classe compilados.

Em geral, esse caminho é o mesmo da raiz da árvore-fonte. Escolha c:\projects\src\java\classes. Assim, você configurará o CLASSPATH:

```
set CLASSPATH=c:\projects\src\java\classes
```

A Figura B.1 demonstra as configurações de ambiente que acabamos de descrever.

**FIGURA B.1**  
*Configurando  
o ambiente.*



A screenshot of a Windows Command Prompt window titled 'C:\WINNT\System32\cmd.exe'. The window contains the following text:

```
C:\>set CLASSPATH=c:\projects\src\java\classes
C:\>echo %CLASSPATH%
c:\projects\src\java\classes
C:\>set JAVA_HOME=c:\jdk1.3
C:\>echo %JAVA_HOME%
c:\jdk1.3
C:\>set PATH=%JAVA_HOME%\bin;%PATH%
C:\>echo %PATH%
c:\jdk1.3\bin;c:\winnt\system32
C:\>
```

A maior parte dos outros fornecedores empacotará suas bibliotecas na forma de arquivos jar ou zip. Para usar essas bibliotecas, você deve anexar a localização do arquivo jar no CLASSPATH. Por exemplo, para usar c:\projects\lib\myclasses.jar, você deve executar o seguinte:

```
set CLASSPATH=%CLASSPATH%;c:\projects\lib\myclasses.jar
```

Você deve anexar cada arquivo jar em CLASSPATH, antes de usar.

B

## Panorama das ferramentas do SDK

Além de fornecer as bibliotecas Java, o Java SDK fornece várias ferramentas necessárias para o desenvolvimento. As ferramentas mais usadas são:

- javac
- java
- jar
- javadoc

Você precisará das outras ferramentas para recursos adicionais, como chamadas de métodos remotos e interface Java nativa.

### Compilador Java: javac

javac , o compilador Java, compila código-fonte Java em byte code. Quando você digitar javac FirstProgram.java, o compilador gerará um arquivo FirstProgram.class no diretório corrente — supondo que o compilador não detecte nenhum erro dentro de seu código Java.

Se um programa Java utilizar bibliotecas de terceiros, o compilador tentará localizar essas bibliotecas a partir do CLASSPATH especificado; entretanto, você pode optar por modificar a CLASSPATH ao compilar. A opção `-classpath <caminho>` permite aos desenvolvedores substituir o CLASSPATH.

Você também pode especificar diferentes diretórios de origem e destino, através de `-sourcepath <caminho>` e `-d <caminho>`, respectivamente. A opção `-sourcepath` especifica uma nova localização para arquivos-fonte de entrada. Assim, você pode optar por compilar arquivos-fonte loca-

lizados em um lugar diferente do diretório corrente. A opção `-d` informa o compilador para que deposite os arquivos `.class` no caminho especificado, em vez de fazê-lo no diretório corrente.

Você não precisará das opções `-classpath`, `-sourcepath` ou `-d` para os exercícios deste livro.

## Interpretador Java: `java`

`java`, o interpretador Java, fornece o ambiente de tempo de execução. Ele interpretará e executará os arquivos de classe compilados. Para executar um programa Java, digite:

```
java FirstProgram
```

Note que o comando omite a extensão `.class`. O interpretador anexa `.class` automaticamente no nome da classe.

Assim como no compilador, você pode especificar opções de linha de comando para o interpretador. Algumas das opções usadas mais freqüentemente são:

- `-classpath`, para especificar um caminho de classe diferente daquele definido no `CLASSPATH`
- `-DpropName=propValue`, para especificar propriedades de sistema

Essas e outras opções do interpretador, entretanto, estão fora dos objetivos desta discussão.

## Utilitário de compactação de arquivos Java: `jar`

O utilitário `jar` gera arquivos compactados java (`jar`). Os arquivos `jar` são equivalentes aos arquivos `zip`, compactando arquivos para um tamanho menor e fornecendo uma maneira conveniente de distribuir classes java compiladas.

Para criar um arquivo compactado, basta chamar:

```
jar cvf <nomedoarquivojar> <arquivos_a_compactar>
```

Você também pode especificar todos os arquivos `jar` e subdiretórios de determinado diretório, através de:

```
jar cvf <nomedoarquivojar> <diretório>
```

Note que todos os arquivos `jar` devem ter o sufixo `.jar`.

Se você quiser ver o conteúdo de um arquivo `jar`, chame:

```
jar tvf <arquivojar>
```

O comando `jar tvf` exibirá o tamanho, a data de inserção e o nome de arquivo que está no arquivo compactado.

Mais opções estão descritas na documentação do SDK Java.

## Documentação Java e o gerador de documentação: javadoc

Todos os desenvolvedores já ouviram o mantra da documentação de código. A JavaSoft facilita o processo de documentação, fornecendo uma ferramenta para gerar documentação HTML amigável para o usuário. O desenvolvedor precisa utilizar apenas tags padrão, ao escrever comentários de atributos, métodos e classes, e depois chamar a ferramenta javadoc com as opções apropriadas para gerar documentação de API. A documentação do SDK Java fornece informações elaboradas sobre a ferramenta e o processo javadoc; entretanto, vamos abordar alguns recursos básicos para começar.

Os desenvolvedores devem primeiro documentar seu código com os comentários e tags apropriados. Seria ótimo se a ferramenta de documentação pudesse adentrar nosso código e decifrar exatamente o que estávamos pensando quando escrevemos aquele método em particular; entretanto, a ferramenta javadoc não é suficientemente avançada para ler a mente.

Em Java, existem três níveis principais de documentação: *classe*, *método* e *atributo*. A ferramenta javadoc reconhecerá comentários que começam com `/**` como um comentário javadoc. Os comentários terminam com `*/`.

Para comentários de classe, normalmente você verá:

- `@author <nome_autor>` especifica o autor da classe. Você pode ter mais de um autor; entretanto, cada autor deve começar com a tag `@author`.
- `@version <número_versão>` especifica a versão da classe. Alguns softwares de controle de versão fornecem tags que gerarão o número automaticamente.
- `@see <nomedaclasse>` fornece links para outras classes, para mais informações. Você pode ter mais de uma referência; entretanto, cada referência deve começar com a tag `@see`.

Uma documentação de classe seria semelhante a:

```
/**  
 * <Comentários e descrição da classe>  
 *  
 * @author Michael C. Han  
 * @author Tony Sintes  
 * @version 1.0  
 * @see SecondJavaClass  
 */
```

Os comentários de método usam as tags anteriores, além de:

- `@param <nome_parâmetro> <comentários>` descreve os parâmetros do método.
- `@return <comentários>` descreve o valor de retorno do método.
- `@exception <NomeExceção> <comentários>` descreve todas as exceções lançadas pelo método corrente.

B

Uma documentação de método seria semelhante a:

```
/**  
 * <descrição e comentários do método>  
 *  
 * @param value1 parâmetro obtido pelo método de teste  
 * @param value2 segundo parâmetro obtido pelo método de teste  
 * @return verdadeiro se value1 == value2  
 * @exception NumberFormatException lançado se value1 ou value2 não forem  
 inteiros  
 */
```

Os comentários javadoc de atributo tendem a não ter tags especiais. Em vez disso, você deve denotar o bloco de comentário como um comentário javadoc. A seguir está um exemplo de comentário de atributo javadoc:

```
/**  
 * Atributo de classe para conter o estado da operação de comparação anterior  
 */
```

## Cercadinho Java: seu primeiro programa Java

Para ajudar a confirmar sua instalação do SDK e praticar com as ferramentas descritas até aqui, você vai escrever o infame exemplo `HelloWorld`. Primeiro, crie um arquivo chamado `HelloWorld.java`, na raiz da origem. Se estiver usando a raiz da origem sugerida (`\projects\src\java\classes`), crie o arquivo sob `c:\projects\src\java\classes`.

Após criar o arquivo, você pode começar a escrever seu primeiro programa Java. A Listagem B.1 contém o exemplo `HelloWorld` em Java.

### **LISTAGEM B.1** `HelloWorld.java`

---

```
/**  
 * Um programa hello world para confirmar que o SDK foi configurado corretamente.  
 * Também utilizado para ajudar a demonstrar as ferramentas básicas do SDK.  
 *  
 * @author Michael C.Han  
 * @version 1.0  
 */  
public class HelloWorld {  
    /**  
     * Método principal do programa. Todas as classes Java executáveis devem  
     * conter esse método.  
     *  
     * @param argumentos passados da linha de comando
```

**LISTAGEM B.1** HelloWorld.java (*continuação*)

```
/*
public static void main(String [] args) {
    HelloWorld helloTest = new HelloWorld();
    System.out.println(helloTest.sayHello());
    System.out.println("");
    System.out.println(helloTest.sayHi());
}

/**
 * Construtor da classe padrão
 *
 */
public HelloWorld() {
}

/**
 * Método para dizer hello para o chamador
 * @return String dizendo "Hello"
 */
public String sayHello() {
    return "Hello";
}

/**
 * Método para dizer hi para o chamador
 * @return String dizendo "Hi!"
 */
public String sayHi() {
    return "Hi!";
}
```

B

## Compilando e executando

Para compilar a classe, execute:

```
javac HelloWorld
```

no diretório-raiz da origem. Se tiver êxito, você verá um arquivo `HelloWorld.class` gerado no mesmo diretório.

Em seguida, execute o programa, digitando:

```
java HelloWorld
```

Se o programa `HelloWorld` executar com sucesso e imprimir “Hello” e “Hi！”, como mostrado na Figura B.2, você configurou seu SDK corretamente; entretanto, se você ver o seguinte ao executar `java HelloWorld`,

`Exception in thread "main" java.lang.NoClassDefFoundError:HelloWorld`

ou não configurou o `CLASSPATH` corretamente ou colocou o arquivo `HelloWorld.java` em um lugar que não é a raiz da origem. Em qualquer caso, confirme primeiro se o `CLASSPATH` contém a raiz da origem (`\projects\src\java\classes`) e, em seguida, confirme se o arquivo `HelloWorld.java` está na raiz da origem.

**FIGURA B.2**  
*Compilando e  
executando  
HelloWorld.*

```
C:\WINNT\System32\cmd.exe
C:\projects\src\java\classes>dir
Volume in drive C is WINDOWS2000
Volume Serial Number is 0063-11FD
Directory of C:\projects\src\java\classes
03/20/2001 02:56a <DIR> .
03/20/2001 02:56a <DIR> ..
03/20/2001 02:59a 939 HelloWorld.java
               1 File(s)    939 bytes
               2 Dir(s) 2,278,046,464 bytes free

C:\projects\src\java\classes>javac HelloWorld.java

C:\projects\src\java\classes>dir
Volume in drive C is WINDOWS2000
Volume Serial Number is 0063-11FD
Directory of C:\projects\src\java\classes
03/20/2001 02:56a <DIR> .
03/20/2001 02:56a <DIR> ..
03/20/2001 02:59a 939 HelloWorld.java
03/20/2001 03:08a 627 HelloWorld.class
               2 File(s) 1,566 bytes
               2 Dir(s) 2,278,838,272 bytes free

C:\projects\src\java\classes>java HelloWorld
Hello
Hi!
C:\projects\src\java\classes>
```

## Criando um arquivo .jar

Em seguida, tente executar o utilitário `jar` nos arquivos do diretório. Na raiz da origem, execute:

`jar cvf hello.jar *.java *.class`

Você verá um arquivo `hello.jar` gerado.

Para confirmar o conteúdo de `hello.jar`, execute:

`jar tvf hello.jar`

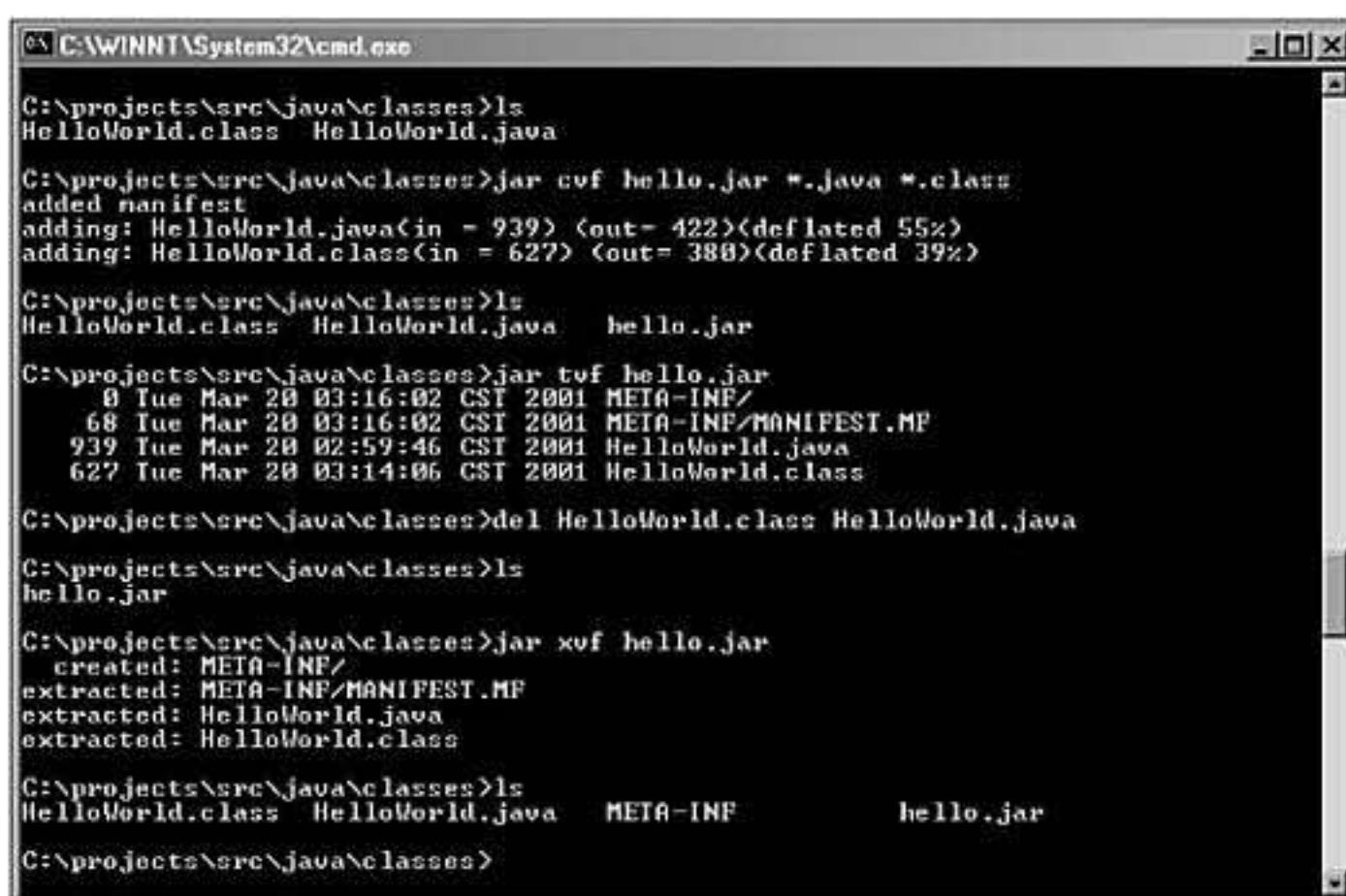
Você verá dois arquivos—`HelloWorld.java` e `HelloWorld.class`—na listagem, como na Figura B.3.

Em seguida, exclua `HelloWorld.java` e `HelloWorld.class` da raiz da origem (`\projects\src\java\classes`). Finalmente, você precisa executar:

`jar xvf hello.jar`

Isso extrairá os arquivos `HelloWorld.java` e `HelloWorld.class` na raiz da origem (veja a Figura B.3).

**FIGURA B.3**  
*Criando, listando e  
 extraindo hello.jar.*



```

C:\WINNT\System32\cmd.exe
C:\projects\src\java\classes>ls
HelloWorld.class HelloWorld.java

C:\projects\src\java\classes>jar cvf hello.jar *.java *.class
added manifest
adding: HelloWorld.java(in = 939) (out= 422)(deflated 55%)
adding: HelloWorld.class(in = 627) (out= 380)(deflated 39%)

C:\projects\src\java\classes>ls
HelloWorld.class HelloWorld.java hello.jar

C:\projects\src\java\classes>jar tvf hello.jar
  0 Tue Mar 20 03:16:02 CST 2001 META-INF/
  68 Tue Mar 20 03:16:02 CST 2001 META-INF/MANIFEST.MF
  939 Tue Mar 20 02:59:46 CST 2001 HelloWorld.java
  627 Tue Mar 20 03:14:06 CST 2001 HelloWorld.class

C:\projects\src\java\classes>del HelloWorld.class HelloWorld.java
C:\projects\src\java\classes>ls
hello.jar

C:\projects\src\java\classes>jar xvf hello.jar
  created: META-INF/
  extracted: META-INF/MANIFEST.MF
  extracted: HelloWorld.java
  extracted: HelloWorld.class

C:\projects\src\java\classes>ls
HelloWorld.class HelloWorld.java META-INF hello.jar

C:\projects\src\java\classes>

```

B

## Gerando javadoc

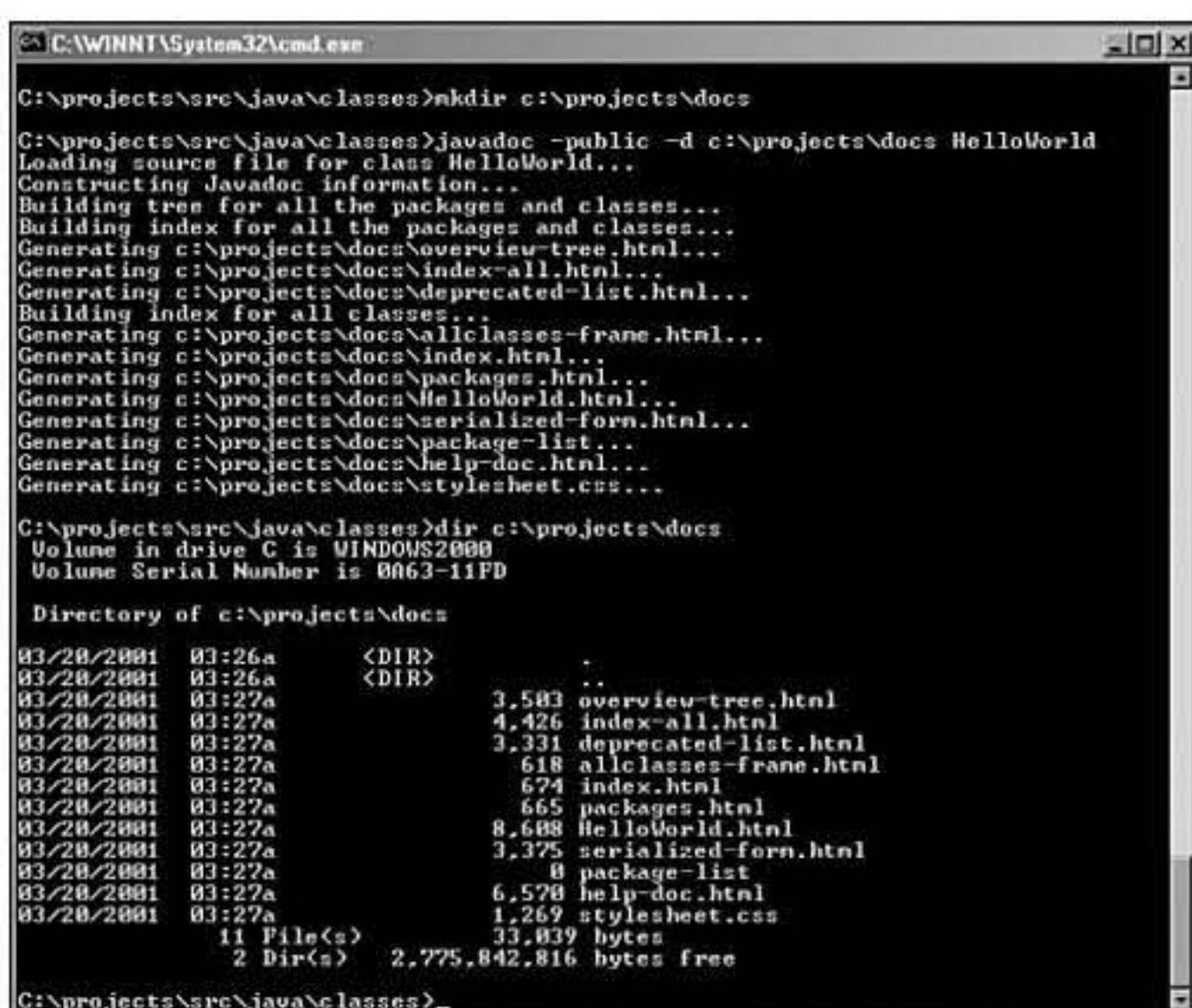
Se você examinar o código-fonte de `HelloWorld`, notará comentários no estilo javadoc para a classe e para os métodos da classe.

Para gerar a documentação, crie o diretório de documentação. Sob `c:\projects`, crie um diretório `docs` e depois execute:

```
javadoc -public -d c:\projects\docs HelloWorld
```

O comando depositará a documentação HTML no diretório `c:\projects\docs` para todos os métodos públicos da classe `HelloWorld`, como se vê na Figura B.4. Para ver a documentação, abra `c:\projects\docs\index.html` em um navegador da Web. Observe a semelhança de estilos entre a documentação gerada e a documentação da API J2SE da Sun.

**FIGURA B.4**  
*Tela da geração de javadoc  
 de HelloWorld.*



```

C:\WINNT\System32\cmd.exe
C:\projects\src\java\classes>mkdir c:\projects\docs
C:\projects\src\java\classes>javadoc -public -d c:\projects\docs HelloWorld
Loading source file for class HelloWorld...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating c:\projects\docs\overview-tree.html...
Generating c:\projects\docs\index-all.html...
Generating c:\projects\docs\deprecated-list.html...
Building index for all classes...
Generating c:\projects\docs\allclasses-frame.html...
Generating c:\projects\docs\index.html...
Generating c:\projects\docs\packages.html...
Generating c:\projects\docs\HelloWorld.html...
Generating c:\projects\docs\serialized-form.html...
Generating c:\projects\docs\package-list...
Generating c:\projects\docs\help-doc.html...
Generating c:\projects\docs\stylesheet.css...

C:\projects\src\java\classes>dir c:\projects\docs
Volume in drive C is WINDOWS2000
Volume Serial Number is 0063-11FD

Directory of c:\projects\docs

03/28/2001  03:26a      <DIR>          .
03/28/2001  03:26a      <DIR>          ..
03/28/2001  03:27a          3,583  overview-tree.html
03/28/2001  03:27a          4,426  index-all.html
03/28/2001  03:27a          3,331  deprecated-list.html
03/28/2001  03:27a          618   allclasses-frame.html
03/28/2001  03:27a          674   index.html
03/28/2001  03:27a          665   packages.html
03/28/2001  03:27a          8,608  HelloWorld.html
03/28/2001  03:27a          3,375  serialized-form.html
03/28/2001  03:27a          8     package-list
03/28/2001  03:27a          6,570  help-doc.html
03/28/2001  03:27a          1,269  stylesheet.css
               11 File(s)      33,039 bytes
                2 Dir(s)   2,775,842,816 bytes free

C:\projects\src\java\classes>

```

## Mecânica da linguagem Java

Após seguir os passos anteriores, você deve ter um ambiente de desenvolvimento Java totalmente configurado, assim como um entendimento das ferramentas básicas do SDK. De posse do ambiente de desenvolvimento, agora é hora de escrever algumas classes Java.

### Classe Java simples

A seguir está a versão simplificada do programa `HelloWorld` apresentado na seção anterior. Agora que você sabe como faz para compilar e executar programas Java, pode examinar o código-fonte com mais detalhes:

```
public class SimpleHelloWorld {  
    public static void main( String args [] ) {  
        String hi = new String("Hello All");  
        System.out.println(hi);  
    }  
}
```

A primeira palavra-chave utilizada é `public`. A palavra `public` é denominada modificador de acesso. Semelhantemente à linguagem C++ ou SmallTalk, a linguagem Java fornece modificadores de acesso para especificar quem pode acessar um método, atributo ou classe em particular. O acesso público garante o acesso a todos aqueles que quiserem usar uma classe, método ou atributo em particular. Os outros modificadores de acesso são `protected`, `private` e *nível de pacote*. Nível de pacote é algo especial na linguagem Java. Basicamente, um modificador de nível de pacote garante o acesso a todas as classes dentro do mesmo pacote ou diretório.

A próxima palavra-chave do exemplo de classe é `class`. Em Java, uma classe é o fundamento básico e os blocos de construção de programas. Ela é o encapsulamento de variáveis de dados ou atributos e operações, funções, ou métodos. Tudo em Java deve residir dentro de uma classe.

O nome do exemplo de classe é `SimpleHelloWorld`. Em Java, o código-fonte dessa classe deve residir em um arquivo chamado `SimpleHelloWorld.java`. Se a classe residir em qualquer outro arquivo, o compilador reclamará que “`class SimpleHelloWorld is public`, deveria ser declarado em um arquivo com nome `SimpleHelloWorld.java`”. O arquivo de classe compilado reside em `SimpleHelloWorld.class`.

A classe `SimpleHelloWorld` também contém `public static void main(String args [])`. Ignore os modificadores `static` e `void`. A linguagem Java exige um método principal, se você quiser executar uma classe Java através do comando `java`. Você pode optar por adicionar mais métodos e operações na classe; entretanto, sem um método com essa assinatura, você não pode executar a classe.

Observe as chaves (`{}`) e pontos-e-vírgulas (`;`) no código-fonte. Em Java, as chaves designam blocos de programa. Um método e uma classe devem começar e terminar com uma chave.

Assim, você deve tomar o cuidado especial de terminar chaves de abertura com chaves de fechamento. Os pontos-e-vírgulas designam o final de uma instrução. Se você é programador de C/C++, então já está familiarizado com eles e sabe que deve fechar todas as instruções com pontos-e-vírgulas.

As palavras-chave `static` e `void` descrevem um método. A palavra-chave `void` é o valor de retorno. Semelhantemente às outras linguagens, os métodos Java têm um valor de retorno. Neste caso, o método principal não retorna nenhum valor. A palavra-chave `static` designa o método como acessível através da classe. Em outras palavras, você pode chamar um método estático de uma classe, sem criar uma instância da classe.

As duas linhas de código a seguir, encapsuladas pelas chaves do método principal, representam o miolo do programa.

```
String hi = new String("Hello All");
System.out.println(hi);
```

B

Aqui, uma referência chamada `hi` aponta para uma instância construída da classe `String`. Como em SmallTalk ou C++, o operador `new` cria a instância. A instância de `String` contém o valor “Hello All”. Ela recebe o valor como parâmetro do construtor. Na linha seguinte, o objeto `System` imprime o valor de `hi` no console do sistema.

Note também os pontos-e-vírgulas no final de cada linha de código. A falha em terminar instruções de programa ou declarações de variável com um ponto-e-vírgula resultará em erros de compilação.

## Tipo de Dados

A linguagem Java, assim como C++ e Smalltalk, é fortemente tipada. Assim, ao declarar variáveis e valores de retorno, você deve especificar a variável ou tipo de retorno. A linguagem Java contém oito tipos primitivos. A Tabela B.1 lista os tipos primitivos válidos em um programa Java e o número de bits associados ao tipo:

**TABELA B.1** Tipos primitivos Java e tamanhos de armazenamento

<i>Tipo</i>	<i>Número de bits</i>
<code>byte</code>	8 bits
<code>short</code>	16 bits
<code>char</code>	16 bits
<code>int</code>	32 bits
<code>float</code>	32 bits
<code>double</code>	64 bits
<code>long</code>	64 bits
<code>boolean</code>	1 bit

Na maioria dos cenários, você pode usar `int` para representar inteiros e `float` para representar valores de ponto flutuante; entretanto, certos tipos de dados podem exigir espaço de armazenamento maior. Por exemplo, talvez você queira expressar o número de milissegundos desde 1970. Esse número é suficientemente grande para exigir um valor `long`, em vez de um inteiro.

Você pode representar um valor `long` como `1000000000L`. A letra L pos-fixada denota o número como `long`. Analogamente, você pode representar valores `float` como `4.3405F`.

Se você for programador de C++, sabe que nessa linguagem os caracteres são ASCII. A linguagem Java, entretanto, usa Unicode para representar caracteres. O padrão Unicode usa 2 bytes para representar caracteres, em oposição a 1 byte usado pelo padrão ASCII. Isso permite a representação de um conjunto de caracteres maior para propósitos de internacionalização. Por exemplo, a maioria das linguagens Asian exige armazenamento maior do que o padrão ASCII. A linguagem Java não o impede completamente de usar ASCII. Em vez disso, você tem a escolha de usar um dos dois. Por exemplo, você pode usar seqüências ASCII comuns, como `\n` para nova linha ou `\t` para tabulação.

Em certas ocasiões, talvez você queira converter um tipo numérico em outro, como de `int` para `long`. A linguagem Java fornece conversão automática entre tipos numéricos, se a conversão não levar a nenhuma perda de precisão. Ela converterá um valor `int` de 32 bits automaticamente para um valor `long` de 64 bits; entretanto, você deve especificar explicitamente a conversão de tipo, para converter de um valor `long` para um valor `int` ou de um valor `double` para um valor `int`. A linguagem Java chama essa operação de conversão. Se você não converter ao realizar uma conversão de redução de precisão, o compilador rejeitará.

As linhas a seguir demonstram uma conversão explícita:

```
long value1 = 40000L;
int value = (int)value1;
float value2 = 4.003F;
double value3 = value2;
```

Observe que a conversão de `float` para `double` não exige conversão explícita, pois um valor `float` tem 32 bits e um valor `double` tem 64 bits. Assim, a operação aumenta a precisão.

## Variáveis

Além de ser fortemente tipada, a linguagem Java também leva em consideração letras maiúsculas e minúsculas. Conseqüentemente, a linguagem Java considera duas variáveis com posicionamento de letras maiúsculas e minúsculas diferente como duas variáveis separadas. Por exemplo, `variableOne` e `variableone` são duas declarações de variável diferentes. Um nome de variável Java deve começar com uma letra e conter quaisquer caracteres alfanuméricos. Os caracteres alfanuméricos podem ser qualquer caractere Unicode que denote uma letra em qualquer idioma; entretanto, o nome não pode conter quaisquer símbolos, como \$, %, & etc.

As seguintes palavras reservadas não podem ser utilizadas:

abstract	boolean	break	byte
case	catch	char	class
const	continue	default	do
double	else	extends	final
finally	float	for	future
generic	goto	if	implements
import	inner	instanceof	int
interface	long	native	new
null	operator	outer	package
private	protected	public	rest
return	short	static	super
switch	synchronized	this	throw
throws	transient	try	var
void	volatile	while	

B

A seguir estão alguns exemplos de declarações de variável:

```
int value1;
double Value2;
float _value3;
char VALUE4,Value5;
```

Observe a última declaração de variável. A linguagem Java permite várias declarações de tipo de variável por linha. Nesse cenário, VALUE4 e Value5 são valores char. Para declarar múltiplas variáveis, use uma vírgula para separar cada nome de variável e termine com um ponto-e-vírgula.

Também existem modificadores de atributo que você pode adicionar em uma declaração de variável. Por exemplo, as variáveis podem ser declaradas public, private ou protected, para controle de acesso. Uma variável também pode ser declarada static ou final, ou ambos. Uma variável static é acessível através da classe, enquanto uma variável final não pode ser modificada.

Embora você tenha poucas restrições quanto aos nomes de variável, é altamente aconselhável seguir alguma forma de convenção de atribuição de nomes ou padrões de codificação. Os padrões de codificação levam a um código uniforme e, assim, melhoram a legibilidade do código. Uma convenção de atribuição de nomes de variável comum é colocar a primeira letra de toda palavra em maiúscula, exceto a primeira palavra. Os nomes seriam como:

```
int anIntegerValue;
char aCharValue;
```

Outra convenção é prefixar todas as variáveis privadas e protegidas com um sublinhado. Assim, as variáveis seriam como:

```
private int _value;
private char _aCharValue;
```

Você deve seguir os padrões de codificação de sua empresa ou seguir um padrão de codificação comum.

## Constantes

As constantes podem ser consideradas como um tipo especial de variável. Assim como em outras linguagens, as constantes Java são variáveis com valores que não mudam. Você pode declarar uma constante anexando as palavras-chave `static final` no início da declaração de variável.

Exemplos de declarações de constante incluem:

```
private static final String _NAME_VALUE      = "MyName";
private static final int _AN_INTEGER_CONSTANT = 1;
public static final String PARAMETER_NAME   = "MyParam";
protected static final char _TYPE_CHAR_VALUE = 'c';
```

Observe que os nomes de constante estão em letras maiúsculas. Embora não seja um requisito, uma prática normalmente seguida entre os desenvolvedores de Java é nomear as constantes com todas as letras maiúsculas.

Você pode acessar essas constantes através de `<classname>.<constant_name>`. Por exemplo, se as constantes anteriores forem declaradas em `MyClass`, você poderá acessar a constante `PARAMETER_NAME` através de `MyClass.PARAMETER_NAME`. Essa é a única constante pública declarada e, assim, essa é a única constante disponível fora de `MyClass`; entretanto, você pode usar qualquer uma das outras constantes dentro de `MyClass`. Você também pode usar a constante `_TYPE_CHAR_VALUE` em todas as subclasses de `MyClass`. Ao usar constantes dentro da mesma classe, você pode omitir o prefixo `<classname>`. Por exemplo, para acessar `_NAME_VALUE` dentro de `MyClass`, você pode fazer referência à constante como `MyClass._NAME_VALUE` ou simplesmente `_NAME_VALUE`.

## Operadores

A linguagem Java fornece vários operadores para operações aritméticas e booleanas. Aqueles que estão familiarizados com a sintaxe da linguagem C/C++, devem conhecer a sintaxe desses operadores. Conforme mencionamos anteriormente, todas as instruções Java devem terminar com um ponto-e-vírgula.

A linguagem Java tem os operadores `+, /, -, *` normais para operações aritméticas. O operador `=` é usado para atribuição. Além disso, a linguagem Java suporta o operador de módulo ou resto, `%`. Por exemplo, `15 / 3` é igual a `5`, enquanto `15 % 3` é igual a `0`. Você também pode usar operadores aritméticos ao inicializar variáveis. Por exemplo, inicializar `x` como `n + 2` é semelhante a:

```
int x = n + 2;
```

Você também pode optar por usar uma sintaxe abreviada ao efetuar operações na mesma variável. Por exemplo, para incrementar `x` por `n`, você pode optar por escrever:

```
x = n + x;
```

Você também pode adotar uma estratégia abreviada e escrever:

```
x += n;
```

A linguagem Java também oferece suporte para exponenciação. A linguagem Java não fornece um operador. Em vez disso, você usa o método `pow`, fornecido na classe `java.lang.Math`. assim, para elevar `x` à potência `n`, a expressão seria semelhante a:

```
int y = Math.pow(x, n);
```

A linguagem Java também fornece operadores de incremento e decremento, como em C/C++. Para incrementar `x` por 1, você pode escrever o seguinte:

```
x = x + 1;  
x += 1;
```

A melhor maneira, entretanto, é escrever:

```
x++;
```

O mesmo se aplica ao operador de decremento:

```
x--;
```

Para os dois operadores, existem duas formas. Você pode colocar o operador antes ou depois da variável. Por exemplo, para decrementar `x`, você pode se expressar como segue:

```
x--;  
-- x;
```

A localização do operador diz quando ele é efetuado. Em uma expressão aritmética, colocar o operador antes da variável incrementará (ou decrementará) a variável primeiro, antes da avaliação da expressão. Se você colocar o operador depois da variável, a expressão será avaliada primeiro. Pegue como exemplo o seguinte trecho de código:

```
int x = 5;  
int y = 6;  
int k = ++x; //após esta expressão, k = 6 e x = 6  
int j = y++; //após esta expressão, j = 6 e y = 7
```

Como você pode ver, o posicionamento do operador de incremento ou decremento é fundamental. Tome o cuidado especial de usar os operadores apropriadamente.

Os operadores booleanos da linguagem Java são muito parecidos com os operadores booleanos de outras linguagens de programação. Ao comparar maior que ou menor que, a linguagem Java fornece `>` e `<`, respectivamente. Além disso, `<=` e `>=` denotam menor ou igual a e maior ou igual a, respectivamente. Isso se aplica à comparação de tipos primitivos, como valores inteiros (`integers`), `long`, `double`, `caracteres`, etc:

```
int x = 5;  
int y = 6;  
boolean z = x < y; // z = verdadeiro
```

B

Para fazer comparação de igualdade, a linguagem Java fornece dois operadores diferentes. Você pode usar o operador `==` para comparar tipos primitivos; entretanto, ao comparar dois objetos (como `String`, `MyObject`), você deve usar o método `equals()`, definido para o objeto. Se você usar `==` por engano, para comparar dois objetos, isso na verdade comparará a referência dos dois objetos e não os objetos reais. A não ser que você esteja comparando as mesmas referências de objeto, esta operação retornará falsa:

```
String x = new String("My String");
String y = new String("My String");
boolean z = (x == y); // z = falso
boolean w = (x.equals(y)); // w = verdadeiro
```

Para fazer comparação de desigualdade, a linguagem Java usa o operador `!=`. A desigualdade entre tipos primitivos usará `!=` e entre dois objetos será `!obj1.equals(obj2)`.

Você também pode optar por usar `&&` e `||` para verificar condicionais (operações `and` (e) e `or` (ou), respectivamente).

## Estruturas condicionais

As estruturas condicionais são uma parte importante de qualquer linguagem de programação. Assim como C/C++, a linguagem Java fornece o operador `if/else`. A sintaxe, assim como em C/C++, é como segue:

```
if (condição) {
    ...//comportamento para quando a condição for verdadeira
}
else {
    ...// comportamento para quando a condição for falsa
}
```

Você pode optar por ter um único `if`, `if/else` ou `if` com várias instruções `else`:

```
if (x == y) {
    x--;
}
else if (x > y) {
    x * 2;
}
else {
    x / 2;
}
```

Observe as chaves que denotam o início e o final de um bloco `if`. Embora as chaves não sejam exigidas, se você tiver apenas uma instrução no bloco `if`, usá-las nessa situação é recomendado. Fazendo isso, você pode evitar dores de cabeça em potencial posteriormente, quando decidir

adicionar mais instruções no bloco `if`. Se você se esquecer de adicionar chaves nesse momento, terá que depurar para encontrar as chaves ausentes.

A linguagem Java, assim como C/C++, fornece um operador ternário para reduzir a expressão `if/else`. Uma expressão ternária é como segue:

```
z = (x > y) ? 3 : 1;
```

`(x > y)` é o teste condicional. O primeiro valor após `?` representa o valor de `z` se a condicional for verdadeira. O segundo valor representa o valor de `z` se a condicional for falsa.

Seguindo o exemplo anterior, se `x = 5` e `y = 7`, então `z = 1`. A condicional é falsa (`5 > 7`) e, assim, `z` recebe o segundo valor na expressão, 1.

B

## Laços ou estruturas de repetição

Os laços ou estruturas de repetição são um componente importante para qualquer linguagem de programação. A linguagem Java, assim como muitas outras, fornece uma variedade de mecanismos de laço.

Um laço usado comumente, o laço `for`, tem a seguinte sintaxe:

```
for (int x = 0; x < 8; x++) {  
    //...alguma operação em laço  
}
```

A restrição do laço, `x`, é definida dentro dos parênteses e inicializada como zero. A instrução seguinte testa a restrição para garantir que ela esteja dentro dos limites necessários. Neste caso, `x` deve ser menor que 8. A última instrução incrementa a restrição do laço.

Outro laço comumente usado, o laço `while`, tem a seguinte sintaxe:

```
while (condição for verdadeira) {  
    //...executa alguma operação  
}
```

O laço continuará a executar, desde que a condição dentro do teste entre parênteses seja verdadeira.

## Classes e interfaces — blocos de construção da linguagem Java

Em Java, você não pode fazer nada sem classes e, até certo ponto, interfaces. As classes compreendem os blocos de construção básicos da linguagem. Assim, quando escrever seu próprio programa, você terá que criar classes e também usar classes de bibliotecas já existentes. Talvez você também precise construir interfaces para melhor expressar seu projeto através do programa.

## Usando classes já existentes

Para usar classes já existentes no SDK Java, você precisará criar uma instância da classe, iniciá-la e depois trabalhar com as instâncias. Para utilizar a classe existente `java.util.StringBuffer`, você criaria uma instância ou objeto dessa classe:

```
StringBuffer sbTest = new StringBuffer();
```

Agora que você já criou um novo `StringBuffer` para usar, pode operar no objeto. Por exemplo, talvez você quisesse anexar outra `String` no buffer.

```
sbTest.append("This is a test String");
```

Talvez seja preciso criar várias instâncias da mesma classe, para executar as operações necessárias. Assim, você pode criar quantos objetos `StringBuffer` forem necessários:

```
StringBuffer sbTest2 = new StringBuffer();
```

Agora, você criou uma nova instância de um objeto `StringBuffer`, identificado pela referência `sbTest2`. Você também pode atribuir uma variável a outra, através do sinal `=`. Isso configurará as duas variáveis para que apontem para o mesmo objeto. Assim, as operações executadas nas duas variáveis afetarão o mesmo objeto. Tome como exemplo as instruções a seguir:

```
StringBuffer sbTest = new StringBuffer();
StringBuffer sbTest2 = new StringBuffer();
sbTest = sbTest2;
sbTest.append( "Test String 1." );
sbTest2.append( "Test String 2." );
System.out.println("Output for sbTest = " +sbTest.toString() );
System.out.println("Output for sbTest2 = " +sbTest2.toString() );
```

Como você configura `sbTest = sbTest2`, as operações de anexação em `sbTest` e `sbTest2` modificarão o mesmo objeto subjacente. A saída do trecho de código anterior será:

```
Output for sbTest = Test String 1.Test String 2.
Output for sbTest2 = Test String 1.Test String 2.
```

## Criando suas próprias classes

A sintaxe Java para uma classe é:

```
public class ClassName {
    //métodos e atributos da classe
}
```

Todos os métodos e atributos da classe residirão dentro das chaves. Todas as classes também devem residir em um arquivo chamado `ClassName.java`.

Examine o trecho de código do arquivo `Product.java`, apresentado na Listagem B.2.

**LISTAGEM B.2** Trecho de Product.java

```
/**  
 *  
 * @author Michael C.Han  
 * @version 1.0  
 */  
public class Product {  
    /**  
     *  
     * @param argumentos passados da linha de comando  
     */  
    public static void main(String [] args) {  
        Product prod =  
            new Product("widget1", "Acme Inc", "Master Widget",  
                        "This is the master widget product to solve all  
problems!",  
                        90.10f);  
        System.out.println(prod.getProductId());  
        System.out.println(prod.getManufacturer());  
        System.out.println(prod.getProductDesc());  
    }  
  
    private String prodName, productId, manufacturer, desc;  
    float price;  
    /**  
     *  
     * @param argumentos passados da linha de comando  
     */  
    public Product(String productId, String manufacturer,  
                  String prodName, String desc, float price) {  
        this.prodName = prodName;  
        this.productId = productId;  
        this.manufacturer = manufacturer;  
        this.desc = desc;  
        this.price = price;  
    }  
  
    /**  
     *  
     * @return id do produto  
     */  
    public String getProductId() {  
        return productId;  
    }  
}
```

B

**LISTAGEM B.2** Trecho de Product.java (*continuação*)

```
/*
 * @return nome do produto
 */
public String getProductName() {
    return prodName;
}

/**
 *
 * @return nome do fabricante
 */
public String getManufacturer() {
    return manufacturer;
}

/**
 *
 * @return descrição do produto
 */
public String getProductDesc() {
    return desc;
}

/**
 *
 * @return preço do produto
 */
public float getPrice() {
    return price;
}

/**
 *
 * @param price – novo preço desse produto
 */
public void setPrice(float price) {
    this.price = price;
}
```

---

A classe Product tem um método construtor com cinco parâmetros: product name, manufacturer, product id, description e price. O construtor efetua as operações necessárias para inicializar o objeto. Neste caso, você inicializa as informações do produto designadas. Um construtor deve seguir as regras que citaremos:

- O construtor deve ter o mesmo nome da classe.
- Um construtor pode receber qualquer número de parâmetros. Um construtor sem parâmetros é chamado de *construtor padrão*.
- Um construtor não retorna valores.
- Um construtor só pode ser chamado através da palavra-chave new (por exemplo, Product prod = new Product (...)).
- Uma classe pode ter mais de um construtor (ou, mais comumente conhecidos como *construtores sobrecarregados*).

Note o uso da palavra-chave this dentro do construtor. A palavra-chave this opera sobre a instância corrente da classe. No construtor, você usa a palavra-chave this para evitar confundir os atributos privados com os parâmetros do construtor.

Além dos construtores, a classe Product também tem quatro *acessores* ou *métodos de obtenção*. Os acessores fornecem acesso de leitura aos atributos da classe Product. Para modificar atributos da classe, você deve usar métodos chamados *modificadores* ou *de configuração*. A classe Product tem um modificador, setPrice, para modificar o preço de um produto.

Observe que todos os atributos da classe Product são privados e apenas um, price, pode ser modificado após um objeto Product ser criado. É importante lembrar que as classes não devem expor seus atributos internos, a menos que o usuário da classe tenha motivos legítimos para ler e/ou modificar o atributo. Conseqüentemente, você fornece acesso aos atributos através de modificadores e acessores.

A classe Product tem apenas métodos de acesso público. Você pode optar por implementar métodos com outros níveis de acesso; você pode optar por implementar métodos privados ou protegidos, além dos métodos públicos. É importante lembrar que os métodos privados são acessíveis dentro da classe que está implementando e que os métodos protegidos são acessíveis tanto dentro da classe que está implementando quanto de todas as suas subclasses. Ao decidir-se sobre os níveis de acesso para um método, lembre-se das seguintes diretrizes para métodos privados:

- Um método privado não traz preocupações para os usuários da classe.
- O método mudará se a implementação da classe mudar.

Lembre-se também do seguinte em relação aos métodos protegidos:

- Um método protegido não traz preocupações para os usuários dessa classe; entretanto, as classes que estendem funcionalidade da classe que está implementando exigirão acesso ao método.
- O método atende aos requisitos funcionais de todas as subclasses.

B

## Interfaces

Assim como as classes, as interfaces são recursos básicos da linguagem Java. Ao contrário das classes, entretanto, as interfaces não definem atributos e métodos para uma classe. Em vez disso,

as interfaces fornecem definições de método que podem ser implementados pelas classes. Para aqueles que estão familiarizados com C++, as interfaces serão semelhantes às definições de classe dentro de arquivos de cabeçalho.

Dê uma olhada em uma interface para objetos que podem ser ordenados ou comparados:

```
/**  
 * Interface para comparar a igualdade de dois objetos  
 *  
 */  
public interface Comparable {  
  
    /**  
     * Compara esse objeto com um objeto desejado. Se o corrente  
     * for maior, então retorna 1, o objeto toCompare é maior,  
     * então retorna -1. Se os dois objetos forem iguais, retorna 0  
     * @param toCompare – objeto para comparação  
     * @return -1 se toCompare > this, 0 se toCompare == this,  
     * 1 se this > toCompare  
     */  
    public int compare(Object toCompare);  
}
```

Essa interface reside dentro do SDK Java. Ela define os métodos que todos os objetos de tipo Comparable devem implementar. Neste caso, todos os objetos Comparable devem implementar o método de comparação. De forma simples, uma interface é um contrato que uma classe que esteja implementando deve cumprir. Se uma classe implementa uma interface em particular, ela promete implementar todos os métodos designados pela interface.

Atualmente, a linguagem Java fornece suporte apenas para herança simples. Em outras palavras, uma classe Java só pode estender uma classe Java. Conseqüentemente, se você optar por classificar um objeto de ClassA e ClassB (herança múltipla), então só poderá fazer isso através de interfaces. Usando interfaces para simular herança múltipla, a linguagem Java recupera grande parte da funcionalidade fornecida pela herança múltipla. Parte da outra funcionalidade perdida poderia ser recuperada parcialmente, através de métodos de projeto, como composição de objeto.

As interfaces também têm várias outras propriedades. Você não pode instanciar uma interface através de new. Sua classe pode implementar várias interfaces. Por exemplo, a classe Product pode implementar as interfaces Sortable e Clonable:

```
public class Product implements Sortable, Clonable
```

As interfaces também podem estender outras interfaces. A interface Sortable também pode estender uma interface Collectible. Uma classe implementando a interface Sortable também deve cumprir o contrato da interface Collectible:

```
public interface Sortable extends Clonable
```

## Classes internas e classes internas anônimas

Os arquitetos da linguagem Java acrescentaram o conceito de classes internas com a Java 1.1. Uma classe interna é uma classe declarada dentro do escopo de uma classe pública. Por que você deve usar uma? As classes internas têm as seguintes vantagens importantes:

- Um objeto de uma classe interna pode acessar quaisquer atributos, privados ou não, da classe que está encapsulando.
- As classes internas anônimas simplificam tarefas, como callbacks e tratamento de eventos.
- As classes internas são extremamente úteis para a criação de objetos que armazenam dados, que não têm nenhum significado fora do contexto da classe que os empacota, como uma chave de hashing especializada para uma cache. Fora da cache, a chave de hashing não tem nenhum significado. Assim, você cria uma classe interna HashKey dentro do escopo da classe ObjectCache.

Aqui está como a classe ObjectCache com a classe interna HashKey seria:

```
import java.util.Map;
import java.util.HashMap;

public class ObjectCache {
    private Map cache;

    public ObjectCache() {
        cache = new HashMap();
    }

    public void add(String oid, String objName, Object obj) {
        HashKey key = new HashKey(oid, objName);
        cache.put(key, obj);
    }

    public Object get(String oid, String objName) {
        HashKey key = new HashKey(oid, objName);
        Object obj = cache.get(key);
        return obj;
    }

    //....Mais métodos ...

    private class HashKey {
        private String oid, objName;

        public HashKey(String oid, String objName) {
            this.oid = oid;
            this.objName = objName;
        }
    }
}
```

B

```
public String getOid() {
    return oid;
}

public String getObjName() {
    return objName;
}

public boolean equals(Object obj) {
    if (obj instanceof HashKey) {
        HashKey key = (HashKey)obj;
        return (key.getOid().equals(getOid()) ) &&
               key.getObjName().equals(getObjName() ) );
    }
    return false;
}

public int hashCode() {
    return 17;
}
}
```

Nas classes internas, assim como nas classes normais, você pode acessar os atributos da classe interna com a palavra-chave `this`; entretanto, você também pode acessar os atributos da classe que está encapsulando. Você precisará usar a palavra-chave `outer` para acessar esses atributos.

As classes anônimas são uma forma especial de classes internas. As classes anônimas são mais predominantes na escrita de código de tratamento de eventos. Tome como exemplo o trecho de código a seguir, para tratar de eventos de ação em um botão OK:

```
public class FooFrame {
    //...

    JButton ok = new JButton("ok");
    ok.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
            // realiza algum tratamento de evento para o botão ok
        }
    });
    //...
}
```

Embora a sintaxe seja um pouco complicada, as classes internas anônimas permitem que você reduza o número de classes em seu projeto. Em vez de uma classe ou classe interna para cada manipulador de evento, você colocaria as declarações de classe em linha. Isso economiza muito tempo; entretanto, devido à natureza complexa do código, classes internas anônimas em dema-

sia reduzirão a legibilidade do código. Conseqüentemente, se você tiver de realizar muito tratamento de evento, talvez queira usar alguns padrões de projeto (como o padrão Command) para melhorar seu projeto e, assim, esclarecer sua implementação.

## Resumo

De modo algum este resumo é uma introdução completa à linguagem Java. Entretanto, ele fornece informações suficientes, dentro do contexto deste texto. Se você quiser obter mais informações, existem vários sites na Web que fornecem exercícios dirigidos aprofundados sobre os fundamentos da linguagem Java. Um recurso excelente para desenvolvedores iniciantes e experientes é o JDC (Java Developer's Connection), localizado no endereço [developer.java.sun.com](http://developer.java.sun.com). Esse site fornece muitos exercícios dirigidos e software de acesso adiantado para desenvolvedores de Java.

B

PÁGINA EM BRANCO

# APÊNDICE C

## Referência da UML

### Referência da UML

Este apêndice fornece uma referência rápida para a notação UML usada por todo o livro.

### Classes

A UML representa uma classe através de uma caixa dividida em três seções. A seção superior contém o nome, a seção do meio contém os atributos e a seção inferior contém os métodos.

A Figura C.1 ilustra como a UML representa uma classe.

### Objeto

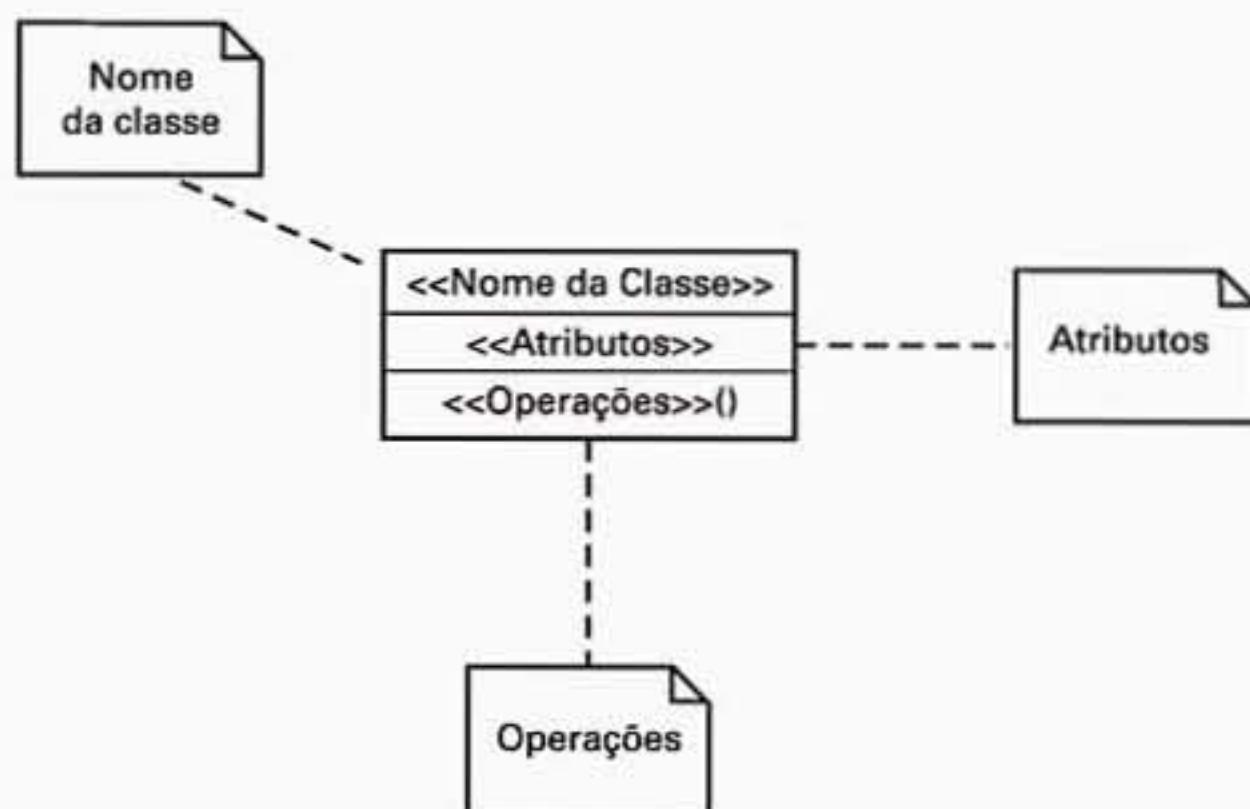
A UML representa objeto igual a uma classe. A única diferença é que o nome do objeto é sublinhado e os métodos são omitidos. Os atributos também podem exibir um valor.

### Visibilidade

A Figura C.2 ilustra como a UML representa visibilidade de atributo e método:

- + representa visibilidade pública
- # representa visibilidade protegida
- - representa visibilidade privada

**FIGURA C.1**  
*Uma classe UML.*



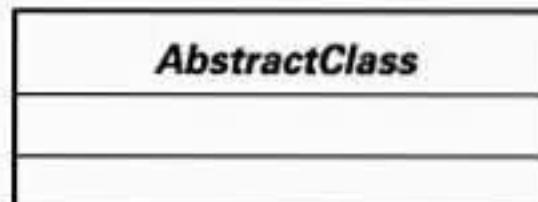
**FIGURA C.2**  
*Visibilidade de método e atributo.*

Visibilidade
+public_attr #protected_attr -private_attr
+public_opr( ) #protected_opr( ) -private_opr( )

## Classes e métodos abstratos

As classes e métodos abstratos são simbolizados pelo nome abstrato em itálico. A Figura C.3 dá um exemplo de classe abstrata.

**FIGURA C.3**  
*Uma classe abstrata.*



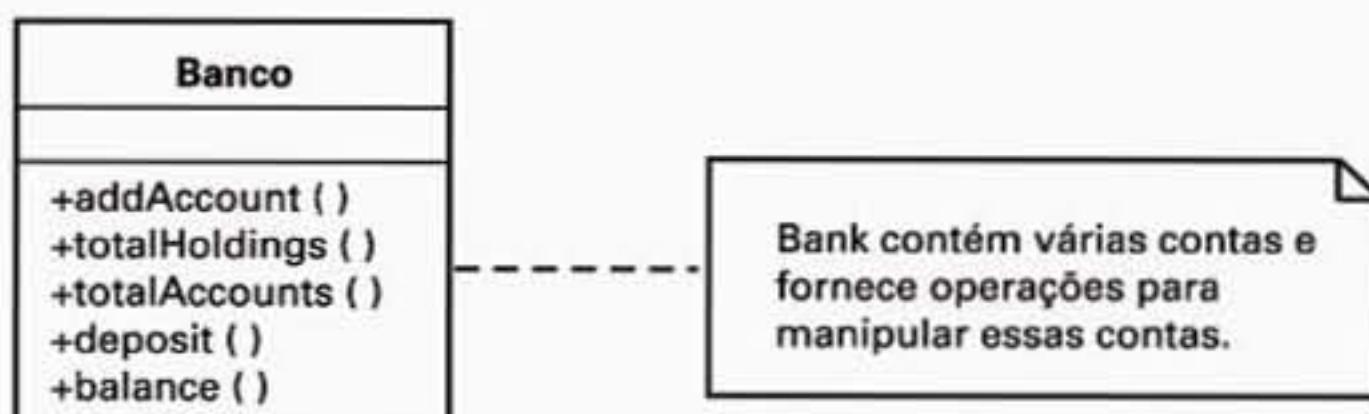
Você também pode adicionar a restrição {abstract} após o nome. Usar o rótulo de restrição ajuda ao se desenhar um diagrama à mão.

## Notas

Às vezes, uma nota tornará um modelo mais inteligível. Na UML, uma nota é semelhante a uma nota adesiva e é ligada ao elemento que a está recebendo com uma linha tracejada.

A Figura C.4 ilustra a nota UML. Você pode anexar uma nota em qualquer parte de seu modelo UML.

**FIGURA C.4**  
A nota UML.

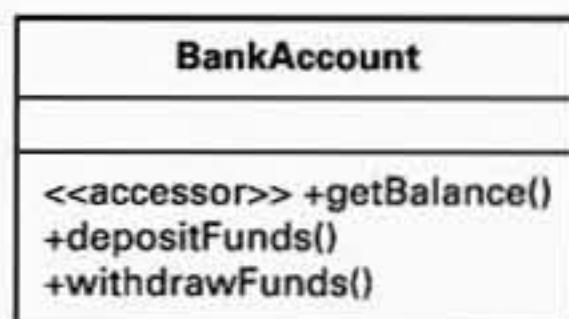


## Estereótipos

Um *estereótipo* é um elemento da UML que permite a você estender o vocabulário da própria linguagem UML ou classificar uma marcação. Um estereótipo consiste em uma palavra ou frase incluída entre <>. Você coloca um estereótipo acima ou ao lado de um elemento existente.

A Figura C.5 ilustra um estereótipo que define um tipo de método.

**FIGURA C.5**  
O estereótipo UML.



## Relacionamentos

Um *relacionamento* descreve como as classes interagem umas com as outras. Na UML, um relacionamento é uma conexão entre dois ou mais elementos de notação. Na UML, um relacionamento é normalmente ilustrado através de uma linha ou de uma seta entre as classes.

### Dependência

Em um relacionamento de *dependência*, um objeto é dependente da especificação de outro objeto. Se a especificação mudar, você precisará atualizar o objeto dependente.

Na UML, você representa uma dependência como uma seta tracejada entre as classes dependentes. A Figura C.6 ilustra a notação de dependência da UML. O relacionamento informa que ClassA depende de ClassB.

**FIGURA C.6**  
Um relacionamento de dependência simples.



## Associação

Uma *associação* indica que um objeto contém outro. Nos termos da UML, quando se está em um relacionamento de associação, um objeto está conectado a outro.

Na UML, você representa uma associação como uma linha que conecta as duas classes. Diz-se que uma associação que não tem seta é bidirecional. Uma seta significa que o relacionamento funciona apenas em uma direção.

A Figura C.7 ilustra a notação de associação da UML. O relacionamento diz que ClassA está associada a ClassB.

**FIGURA C.7**  
*Um relacionamento de associação.*

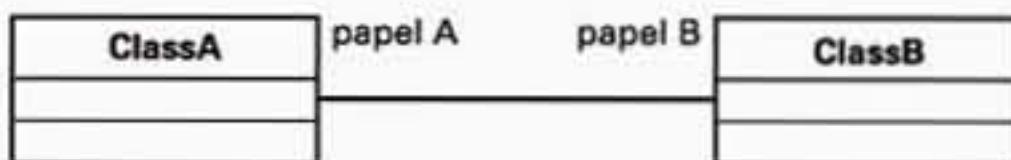


## Papéis

A UML permite que você denote o papel de cada classe na associação. O *papel* da associação é a parte que um objeto desempenha em um relacionamento.

A Figura C.8 ilustra a notação UML para o papel.

**FIGURA C.8**  
*O papel na UML.*



## Multiplicidade

A UML permite que você denote a multiplicidade da associação. A *multiplicidade* indica quantos objetos podem tomar parte na instância de uma associação.

O intervalo dos valores de multiplicidade está listado na Tabela C.1.

**TABELA C.1** Valores de multiplicidade

Notação	Valor
1	Um
*	Qualquer número
1..*	Pelo menos um
x..y	Qualquer número de valores no intervalo x a y

A Figura C.9 ilustra a notação UML para multiplicidade.

**FIGURA C.9**  
*Multiplicidade.*



## Agregação

A UML fornece notação para agregação. Uma *agregação* é um tipo especial de associação que modela relacionamentos ‘tem um’ de todo/parte entre pares.

Você modela uma agregação como uma linha com um losango aberto na extremidade relativa a ‘todo/parte’. A Figura C.10 ilustra a notação UML para agregação.

**FIGURA C.10**  
*Agregação.*



## Composição

A UML fornece notação para composição. Uma *composição* é um tipo especial de associação que modela relacionamentos ‘tem um’ de todo/parte entre classes que não são pares. A parte não é independente do todo em um relacionamento de composição.

Você modela a composição como uma linha com um losango fechado na extremidade relativa ao ‘todo/parte’. A Figura C.11 ilustra a notação UML para composição.

**FIGURA C.11**  
*Composição.*



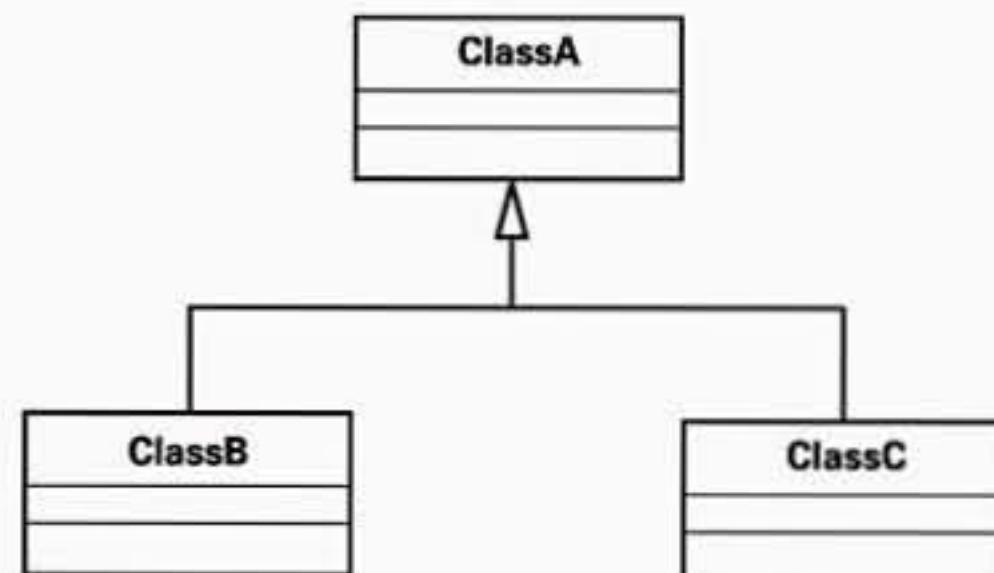
C

## Generalização

Um relacionamento de generalização existe entre o geral e o específico. É a herança.

A generalização é simbolizada através de uma linha cheia com uma seta fechada e vazada. A Figura C.12 ilustra a notação UML para generalização.

**FIGURA C.12**  
*Generalização.*



## Diagramas de interação

Os diagramas de interação modelam as interações entre os objetos.

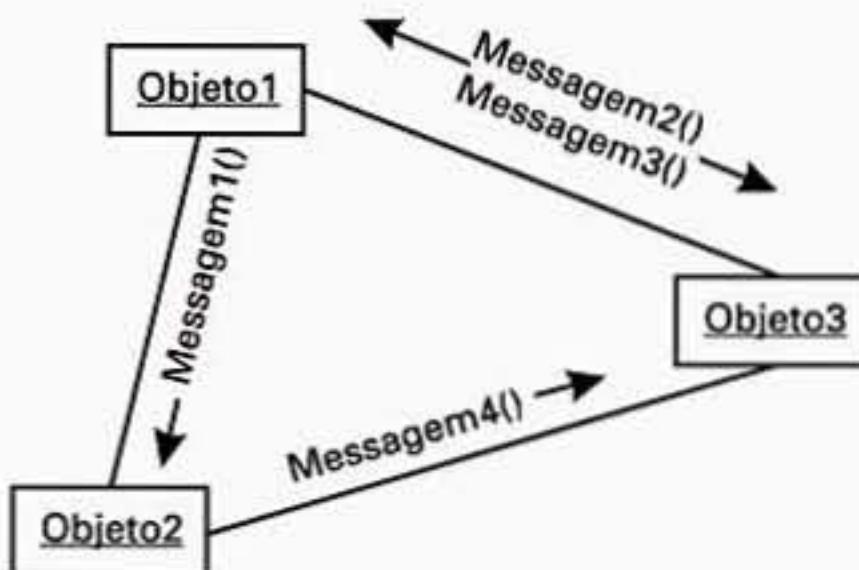
## Diagramas de colaboração

Os diagramas de colaboração representam as mensagens que os objetos enviam uns para os outros.

Cada objeto é simbolizado como uma caixa no diagrama. Uma linha conecta cada objeto que interage. Sobre essa linha, você escreve as mensagens que os objetos enviam e a direção dessas mensagens.

Os diagramas de colaboração destacam os relacionamentos entre os atores. A Figura C.13 ilustra o diagrama de colaboração UML.

**FIGURA C.13**  
*Um diagrama de colaboração.*

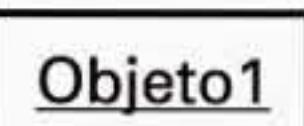


## Diagramas de seqüência

Os diagramas de seqüência modelam a seqüência de eventos em um cenário, com o passar do tempo.

Cada objeto no diagrama é simbolizado na parte superior do diagrama como uma caixa. As linhas que saem das caixas representam a linha da vida do objeto. As mensagens são passadas entre os objetos. A Figura C.14 ilustra o diagrama de seqüência UML.

**FIGURA C.14**  
*Um diagrama de seqüência.*



# APÊNDICE D

## Bibliografia selecionada

O domínio dos objetos só pode vir com o tempo, a prática e o estudo. Nenhum livro pode ensiná-lo tudo que há a respeito da programação orientada a objetos.

Este Apêndice apresenta uma lista de recursos de OO classificados. Use esta lista como guia para seus próximos passos no estudo e na aplicação de POO. Embora não seja importante que você investigue cada recurso, esta lista pode conduzi-lo para mais informações sobre os assuntos que você achar interessantes.

## Análise, projeto e metodologias

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley, 2000.

Booch, Grady. *Object Oriented Analysis And Design: With Applications*. Reading: Addison-Wesley, 1994.

Booch, Grady, James Rumbaugh e Ivar Jacobson. *The Unified Modeling Language User Guide*. Reading: Addison-Wesley, 1999.

Fowler, Martin e Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 2<sup>nd</sup> ed. Reading: Addison-Wesley, 2000.

Johnson, Ralph E. e Brian Foote. “Designing Reusable Classes.” *Journal of Object Oriented Programming* 1.2 (Junho/Julho 1988): 22-35.

Liberty, Jesse. *Beginning Object Oriented Analysis and Design*. Olton, UK: Wrox, 1998.

## Programação com C++

Stroustrup, Bjarne. *The C++ Programming Language*. Special ed. Reading: Addison-Wesley, 2000.

## Padrões de projeto

Buschman, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Chichester: Wiley, 1996.

Gamma, Erich, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1995.

## Princípios e teoria geral da OO

Booch, Grady. *Object Oriented Analysis And Design: With Applications*. Reading: Addison-Wesley, 1994.

Bud, Timothy. *An Introduction to Object Oriented Programming*. 2<sup>nd</sup> ed. Reading: Addison-Wesley, 1997.

Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley, 2000.

Rumbaugh, James. "Disinherited! Examples of Misuse of Inheritance." *Journal of Object Oriented Programming* 5.9 (Fevereiro 1993): 22-24.

Seidewitz, Ed. "Controlling Inheritance". *Journal of Object Oriented Programming* 8.8 (Janeiro 1996): 36-42.

Taenzer, David, Murthy Ganti e Sunil Podar. "Object-Oriented Software Reuse: The Yoyo Problem." *Journal of Object Oriented Programming* 2.3 (setembro/outubro de 1989): 30-35.

## Teoria "Hard Core" (mas não deixe isso assustá-lo!)

Cardelli, Luca e Peter Wegner. "On Understanding Types, Data Abstractions and Polymorphism". *Computing Surveys* 17.4 (1985): 471-523.

Danforth, Scott e Chris Tomlinson. "Type Theories and Object Oriented Programming". *ACM Computing Surveys* 20.1 (1988): 29-72.

Snyder, Alan. "Encapsulation and Inheritance in Object Oriented Programming Languages". Proceedings of the 1986 OOPSLA Conference on Object Oriented Programming Systems, Languages and Applications. *Sigplan Notices* 21.11 (1986): 38-45.

Wegner, Peter e Stanley B. Zdonik. "Inheritance as an Incremental Modification Mechanism or What Like is and isn't Like". *ECOOP* 1988: 55-77.

## Programação com Java

Bloch, Joshua. *Effective Java Programming Language Guide*. Boston: Addison-Wesley, 2001.

Eckel, Bruce. *Thinking in Java*. 2<sup>nd</sup> ed. Upper Saddle River: Prentice Hall, 2000.

Warren, Nigel e Philip Bishop. *Java in Practice: Design Styles and Idioms for Effective Java*. Harlow, UK: Addison-Wesley, 1999.

## Miscelânea

Brooks, Frederick P., Jr. *The Mythical Man-Month*. Anniversary ed. Reading: Addison-Wesley, 1995.

Scarne, John. Scarne's Encyclopedia of Card Games: All the Rules for All the Games You'll Want to Play. Nova York: Harper, 1983.

D

## Smalltalk

Budd, Timothy. *A Little Smalltalk*. Reading: Addison-Wesley, 1987.

Goldberg, Adele e David Robson. *Smalltalk-80: The Language*. Reading: Addison-Wesley, 1989.

## Teste

Beck, Kent. *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley, 2000.

Mackinnon, Tim, Steve Freeman e Philip Craig. "Endo-Testing: Unit Testing with Mock Objects". Artigo apresentado na conferência eXtreme Programming and Flexible Processes in Software Engineering — XP2000, Cagliari, Sardinia, Itália, junho de 2000.

PÁGINA EM BRANCO

# APÊNDICE E

## Listagens do código do jogo vinte-e-um

Nos dias 15 a 21, você construiu um jogo vinte-e-um completo. Muitos dos exercícios também pediram para que você adicionasse mais funcionalidade ao sistema. Esta listagem de código-fonte apresenta a versão final do jogo vinte-e-um. A versão final combina cada recurso que foi adicionado nos dias 15 a 21.

Este apêndice contém a listagem completa de todo o código. O código foi empacotado para evitar arquivos-fonte duplicados. O que pode ser compartilhado é compartilhado. O que não pode ser compartilhado não é compartilhado.

O código-fonte contém uma GUI MVC, uma GUI PAC, uma UI CLI e um simulador. O código-fonte foi dividido nos seguintes pacotes:

- blackjack.core
- blackjack.core.threaded
- blackjack.exe
- blackjack.players
- blackjack.ui
- blackjack.ui.mvc
- blackjack.ui.pac

## blackjack.core

blackjack.core contém todas as classes comuns encontradas nas listagens E.1 a E.14. Essas classes constroem o sistema do jogo vinte-e-um básico.

### **LISTAGEM E.1** Bank.java

---

```
package blackjack.core;

public class Bank {

    private int total;
    private int bet;

    public Bank( int amount ) {
        total = amount;
    }

    public void doubleDown() {
        placeBet( bet );
        bet = bet * 2;
    }

    public void place100Bet() {
        placeBet( 100 );
    }

    public void place50Bet() {
        placeBet( 50 );
    }

    public void place10Bet() {
        placeBet( 10 );
    }

    public void win() {
        total += ( 2 * bet );
        bet = 0;
    }

    public void lose() {
        // já extraído de total
        bet = 0;
    }

    public void blackjack() {
```

**LISTAGEM E.1** Bank.java (*continuação*)

```
    total += ( ( ( 3 * bet )/ 2 )+ bet );
    bet = 0;
}

public void standoff() {
    total += bet;
    bet = 0;
}

public String toString() {
    return ( "$" + total + ".00" );
}

private void placeBet( int amount ) {
    bet = amount;
    total -= amount;
}
}
```

**LISTAGEM E.2** BettingPlayer.java

```
package blackjack.core;

public abstract class BettingPlayer extends Player {

    private Bank bank;

    public BettingPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand );
        this.bank = bank;
    }

    //*****
    //comportamento sobreposto
    public String toString() {
        return ( super.getName() + ":" + getHand().toString() + "\n" +
bank.toString() );
    }

    public String getName() {
        return ( super.getName() + " " + bank.toString() );
    }

    public void win() {
```

E

**LISTAGEM E.2** BettingPlayer.java (*continuação*)

```
    bank.win();
    super.win();
}

public void lose() {
    bank.lose();
    super.lose();
}

public void standoff() {
    bank.standoff();
    super.standoff();
}

public void blackjack() {
    bank.blackjack();
    super.blackjack();
}

protected PlayerState getInitialState() {
    return getBettingState();
}

protected PlayerState getPlayingState() {
    return new BetterPlaying();
}

//*****adicionado recentemente para BettingPlayer*****
protected final Bank getBank() {
    return bank;
}

protected PlayerState getBettingState() {
    return new Betting();
}

protected PlayerState getDoublingDownState() {
    return new DoublingDown();
}

protected abstract void bet();
protected abstract boolean doubleDown( Dealer dealer );

private class Betting implements PlayerState {
```

**LISTAGEM E.2** BettingPlayer.java (*continuação*)

```
public void handChanged() {
    // impossível no estado de estouro
}

public void handPlayable() {
    // impossível no estado de estouro
}

public void handBlackjack() {
    // impossível no estado de estouro
}

public void handBusted() {
    // impossível no estado de estouro
}

public void execute( Dealer dealer ) {
    bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( BettingPlayer.this );
    // termina
}
}

private class DoublingDown implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    public void handBlackjack() {
        // impossível no estado de dobro
    }
    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
    }
    public void execute( Dealer dealer ) {
        bank.doubleDown();
        dealer.hit( BettingPlayer.this );
        getCurrentState().execute( dealer );
    }
}

private class BetterPlaying implements PlayerState {
```

E

**LISTAGEM E.2** BettingPlayer.java (*continuação*)

```
public void handChanged() {
    notifyChanged();
}
public void handPlayable() {
    //pode ignorar no estado de jogo
}
public void handBlackjack() {
    //impossível no estado de jogo
}
public void handBusted() {
    setCurrentState( getBustedState() );
    notifyBusted();
}
public void execute( Dealer dealer ) {
    if( getHand().canDoubleDown() && doubleDown( dealer ) ) {
        setCurrentState( getDoublingDownState() );
        getCurrentState().execute( dealer );
        return;
    }
    if( hit( dealer ) ) {
        dealer.hit( BettingPlayer.this );
    } else {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    getCurrentState().execute( dealer );
    // transição
}
}
```

---

**LISTAGEM E.3** BlackjackDealer.java

```
package blackjack.core;

import java.util.ArrayList;
import java.util.Iterator;

public class BlackjackDealer extends Player implements Dealer {

    private Deckpile cards;

    private ArrayList players = new ArrayList();
```

**LISTAGEM E.3** BlackjackDealer.java (continuação)

```
protected ArrayList waiting_players;
protected ArrayList betting_players;
private ArrayList standing_players;
private ArrayList busted_players;
private ArrayList blackjack_players;

public BlackjackDealer( String name, Hand hand, Deckpile cards ) {
    super( name, hand );
    this.cards = cards;
}

//*****
//Métodos que os jogadores podem chamar
public void blackjack( Player player ) {
    blackjack_players.add( player );
    play( this );
}

public void busted( Player player ) {
    busted_players.add( player );
    play( this );
}

public void standing( Player player ) {
    standing_players.add( player );
    play( this );
}

public void doneBetting( Player player ) {
    waiting_players.add( player );
    play( this );
}

public void hit( Player player ) {
    player.addCard( cards.dealUp() );
}

public Card getUpCard() {
    Iterator i = getHand().getCards();
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        if( card.isFaceUp() ) {
            return card;
        }
    }
}
```

E

**LISTAGEM E.3** BlackjackDealer.java (*continuação*)

```
// não deve chegar aqui
    return null;
}

//*****
// Métodos de configuração do jogo
public void addPlayer( Player player ) {
    players.add( player );
}
public void reset() {
    super.reset();

    //configura os baldes do jogador
    waiting_players = new ArrayList();
    standing_players = new ArrayList();
    busted_players = new ArrayList();
    blackjack_players = new ArrayList();
    betting_players = new ArrayList();
    betting_players.addAll(players);

    cards.reset();
    Iterator i = players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        player.reset();
    }
}
public void newGame() {
    reset();
    // vai!
    play( this );
}
//*****


public void deal() {

    cards.shuffle();
    // reconfigura cada jogador e distribui uma carta aberta para cada um e
    // para si mesma
    Player [] player = new Player [waiting_players.size()];
    waiting_players.toArray( player );
    for( int i = 0; i < player.length; i ++ ) {
        player [i].addCard( cards.dealUp() );
    }
    this.addCard( cards.dealUp() );
```

**LISTAGEM E.3** BlackjackDealer.java (*continuação*)

```
// distribui mais uma carta aberta para cada jogador e uma fechada para
// si mesma
for( int i = 0; i < player.length; i ++ ) {
    player[i].addCard( cards.dealUp() );
}
this.addCard( cards.dealDown() );
}

protected boolean hit( Dealer dealer ) {
    if( standing_players.size() > 0 && getHand().total() < 17 ) {
        return true;
    }
    return false;
}

protected void exposeHand() {
    getHand().turnOver();
    notifyChanged();
}

protected PlayerState getBlackjackState() {
    return new DealerBlackjack();
}
protected PlayerState getDealingState() {
    return new DealerDealing();
}
protected PlayerState getCollectingBetsState() {
    return new DealerCollectingBets();
}
protected PlayerState getBustedState() {
    return new DealerBusted();
}
protected PlayerState getStandingState() {
    return new DealerStanding();
}
protected PlayerState getWaitingState() {
    return new DealerWaiting();
}
protected PlayerState getInitialState() {
    return new DealerCollectingBets();
}

private class DealerCollectingBets implements PlayerState {
    public void handChanged() {
```

E

**LISTAGEM E.3** BlackjackDealer.java (*continuação*)

```
// impossível no estado de aposta
}
public void handPlayable() {
    // impossível no estado de aposta
}
public void handBlackjack() {
    // impossível no estado de aposta
}
public void handBusted() {
    // impossível no estado de aposta
}
public void execute( Dealer dealer ) {
    if( !betting_players.isEmpty() ) {
        Player player =( Player )betting_players.get(0);
        betting_players.remove( player );
        player.play( dealer );
    } else {
        setCurrentState( getDealingState() );
        getCurrentState().execute( dealer );
        // faz a transição e executa
    }
}

private class DealerBusted implements PlayerState {
    public void handChanged() {
        // impossível no estado de estouro
    }
    public void handPlayable() {
        // impossível no estado de estouro
    }
    public void handBlackjack() {
        // impossível no estado de estouro
    }
    public void handBusted() {
        // impossível no estado de estouro
    }
    public void execute( Dealer dealer ) {
        Iterator i = standing_players.iterator();
        while( i.hasNext() ) {
            Player player =( Player ) i.next();
            player.win();
        }
        i = blackjack_players.iterator();
        while( i.hasNext() ) {
```

**LISTAGEM E.3** BlackjackDealer.java (*continuação*)

```
        Player player = (Player) i.next();
        player.blackjack();
    }
    i = busted_players.iterator();
    while( i.hasNext() ) {
        Player player = (Player) i.next();
        player.lose();
    }
}
private class DealerBlackjack implements PlayerState {
public void handChanged() {
    notifyChanged();
}
public void handPlayable() {
    // impossível no estado de vinte-e-um
}
public void handBlackjack() {
    // impossível no estado de vinte-e-um
}
public void handBusted() {
    // impossível no estado de vinte-e-um
}
public void execute( Dealer dealer ) {
    exposeHand();
    Iterator i = players.iterator();
    while( i.hasNext() ) {
        Player player =(Player) i.next();
        if( player.getHand().blackjack() ) {
            player.standoff();
        } else {
            player.lose();
        }
    }
}
private class DealerStanding implements PlayerState {
public void handChanged() {
    // impossível no estado de parada
}
public void handPlayable() {
    // impossível no estado de parada
}
public void handBlackjack() {
    // impossível no estado de parada
}
```

E

**LISTAGEM E.3** BlackjackDealer.java (continuação)

```
    }
    public void handBusted() {
        // impossível no estado de parada
    }
    public void execute( Dealer dealer ) {
        Iterator i = standing_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            if( player.getHand().isEqual( getHand() ) ) {
                player.standoff();
            } else if( player.getHand().isGreaterThan(getHand()) ) {
                player.win();
            } else {
                player.lose();
            }
        }
        i = blackjack_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.blackjack();
        }
        i = busted_players.iterator();
        while( i.hasNext() ) {
            Player player = (Player) i.next();
            player.lose();
        }
    }
}
private class DealerWaiting implements PlayerState {
    public void handChanged() {
        // impossível no estado de espera
    }
    public void handPlayable() {
        // impossível no estado de espera
    }
    public void handBlackjack() {
        // impossível no estado de espera
    }
    public void handBusted() {
        // impossível no estado de espera
    }
    public void execute( Dealer dealer ) {
        if(!waiting_players.isEmpty() ) {
            Player player = (Player) waiting_players.get(0);
            waiting_players.remove( player );
        }
    }
}
```

**LISTAGEM E.3** BlackjackDealer.java (continuação)

```
        player.play( dealer );
    } else {
        setCurrentState( getPlayingState() );
        exposeHand();
        getCurrentState().execute( dealer );
        // faz a transição e executa
    }
}
private class DealerDealing implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getWaitingState() );
        // transição
    }
    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        // transição
    }
    public void handBusted() {
        // impossível no estado de distribuição
    }
    public void execute( Dealer dealer ) {
        deal();
        getCurrentState().execute( dealer );
        // faz a transição e executa
    }
}
```

E

**LISTAGEM E.4** Card.java

```
package blackjack.core;

public class Card {

    private Rank rank;
    private Suit suit;
    private boolean face_up;

    public Card( Suit suit, Rank rank ) {
```

**LISTAGEM E.4** Card.java (*continuação*)

```
    this.suit = suit;
    this.rank = rank;
}

public Suit getSuit() {
    return suit;
}

public Rank getRank() {
    return rank;
}

public void setFaceUp( boolean up ) {
    face_up = up;
}

public boolean isFaceUp() {
    return face_up;
}

public String toString() {
    if( !isFaceUp() ) {
        return "Hidden";
    }
    return rank.toString() + suit.toString();
}
}
```

---

**LISTAGEM E.5** Dealer.java

```
package blackjack.core;

public interface Dealer {
    // usado pelo jogador para interagir com a banca
    public void hit( Player player );

    // usado pelo jogador para comunicar estado à banca
    public void blackjack( Player player );
    public void busted( Player player );
    public void standing( Player player );
    public void doneBetting( Player player );

    public Card getUpCard();
}
```

---

**LISAGEM E.6** Deck.java

```
package blackjack.core;

import java.util.Iterator;
import java.util.Random;

public class Deck {

    private Card [] deck;
    private int index;

    public Deck() {
        buildCards();
    }

    public void addToStack( Deckpile stack ) {
        stack.addCards( deck );
    }

    protected void setDeck( Card [] deck ) {
        this.deck = deck;
    }

    protected void buildCards() {

        deck = new Card[52];
        Iterator suits = Suit.SUITS.iterator();

        int counter = 0;
        while( suits.hasNext() ) {
            Suit suit = (Suit) suits.next();
            Iterator ranks = Rank.RANKS.iterator();
            while( ranks.hasNext() ) {
                Rank rank = (Rank) ranks.next();
                deck[counter] = new Card( suit, rank );
                counter++;
            }
        }
    }
}
```

E

**LISTAGEM E.7** Deckpile.java

```
package blackjack.core;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.Random;

public class Deckpile {

    private ArrayList stack = new ArrayList();
    private int index;
    private Random rand = new Random();

    public void addCards( Card [] cards ) {
        for( int i = 0; i < cards.length; i ++ ) {
            stack.add( cards [i] );
        }
    }

    public void shuffle() {
        reset();
        randomize();
        randomize();
        randomize();
        randomize();
    }

    public Card dealUp() {
        Card card = deal();
        if( card != null ) {
            card.setFaceUp( true );
        }
        return card;
    }

    public Card dealDown() {
        Card card = deal();
        if( card != null ) {
            card.setFaceUp( false );
        }
        return card;
    }

    public void reset() {
        index = 0;
        Iterator i = stack.iterator();
```

**LISTAGEM E.7** Deckpile.java (*continuação*)

```
        while( i.hasNext() ) {
            Card card = (Card) i.next();
            card.setFaceUp( false );
        }
    }

    private Card deal() {
        if( index != stack.size() ) {
            Card card = (Card) stack.get( index );
            index++;
            return card;
        }
        return null;
    }

    private void randomize() {
        int num_cards = stack.size();
        for( int i = 0; i < num_cards; i ++ ) {
            int index = rand.nextInt( num_cards );
            Card card_i = (Card) stack.get( i );
            Card card_index = (Card) stack.get( index );
            stack.set( i, card_index );
            stack.set( index, card_i );
        }
    }
}
```

**LISTAGEM E.8** Hand.java

```
package blackjack.core;

import java.util.ArrayList;
import java.util.Iterator;

public class Hand {

    private ArrayList cards = new ArrayList();
    private static final int BLACKJACK = 21;
    private HandListener holder;
    private int number_aces;

    public Hand() {
        setHolder(
            new HandListener() {
```

E

**LISTAGEM E.8** Hand.java (*continuação*)

```
        public void handPlayable() {}
        public void handBlackjack() {}
        public void handBusted() {}
        public void handChanged() {}
    }
);

public void setHolder( HandListener holder ) {
    this.holder = holder;
}

public Iterator getCards() {
    return cards.iterator();
}

public void addCard( Card card ) {
    cards.add( card );

    holder.handChanged();

    if( card.getRank() == Rank.ACE ) {
        number_aces++;
    }

    if( bust() ) {
        holder.handBusted();
        return;
    }
    if( blackjack() ) {
        holder.handBlackjack();
        return;
    }
    if ( cards.size() >= 2 ) {
        holder.handPlayable();
        return;
    }
}

public boolean canDoubleDown() {
    return ( cards.size() == 2 );
}

public boolean isEqual( Hand hand ) {
    if( hand.total() == this.total() ) {
```

**LISTAGEM E.8** Hand.java (*continuação*)

```
    return true;
}
return false;
}

public boolean isGreaterThan( Hand hand ) {
    return this.total() > hand.total();
}

public boolean blackjack() {
    if( cards.size() == 2 && total() == BLACKJACK ) {
        return true;
    }
    return false;
}

public void reset() {
    cards.clear();
    number_aces = 0;
}

public void turnOver() {
    Iterator i = cards.iterator();
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        card.setFaceUp( true );
    }
}

public String toString() {
    Iterator i = cards.iterator();
    String string = "";
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        string = string + " " + card.toString();
    }
    return string;
}

public int total() {
    int total = 0;
    Iterator i = cards.iterator();
    while( i.hasNext() ) {
        Card card = (Card) i.next();
        total += card.getRank().getRank();
```

E

**LISTAGEM E.8** Hand.java (*continuação*)

```
    }
    int temp_aces = number_aces;
    while( total > BLACKJACK && temp_aces > 0 ) {
        total = total - 10;
        temp_aces--;
    }
    return total;
}

private boolean bust() {
    if( total() > BLACKJACK ) {
        return true;
    }
    return false;
}
}
```

---

**LISTAGEM E.9** HandListener.java

```
package blackjack.core;

public interface HandListener {

    public void handPlayable();

    public void handBlackjack();

    public void handBusted();

    public void handChanged();

}
```

---

**LISTAGEM E.10** Player.java

```
package blackjack.core;

import java.util.ArrayList;
import java.util.Iterator;

public abstract class Player {

    private Hand hand;
```

**LISTAGEM E.10** Player.java (*continuação*)

```
private String name;
private ArrayList listeners = new ArrayList();
private PlayerState current_state;

public Player( String name, Hand hand ) {
    this.name = name;
    this.hand = hand;
    setCurrentState( getInitialState() );
}

public void addCard( Card card ) {
    hand.addCard( card );
}

public void play( Dealer dealer ) {
    current_state.execute( dealer );
}

public void reset() {
    hand.reset();
    setCurrentState( getInitialState() );
    notifyChanged();
}

public void addListener( PlayerListener l ) {
    listeners.add( l );
}

public String getName() {
    return name;
}

public String toString() {
    return (name + ": " + hand.toString() );
}

public void win() {
    notifyWin();
}

public void lose() {
    notifyLose();
}

public void standoff() {
```

E

**LISTAGEM E.10** Player.java (*continuação*)

```
    notifyStandoff();
}

public void blackjack() {
    notifyBlackjack();
}

public Hand getHand() {
    return hand;
}

protected void notifyChanged() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerChanged( this );
    }
}

protected void notifyBusted() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerBusted( this );
    }
}

protected void notifyBlackjack() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerBlackjack( this );
    }
}

protected void notifyStanding() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerStanding( this );
    }
}

protected void notifyStandoff() {
    Iterator i = listeners.iterator();
```

**LISTAGEM E.10** Player.java (*continuação*)

```
        while( i.hasNext() ) {
            PlayerListener pl = (PlayerListener) i.next();
            pl.playerStandoff( this );
        }
    }

protected void notifyWin() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerWon( this );
    }
}

protected void notifyLose() {
    Iterator i = listeners.iterator();
    while( i.hasNext() ) {
        PlayerListener pl = (PlayerListener) i.next();
        pl.playerLost( this );
    }
}

protected final void setCurrentState( PlayerState state ) {
    current_state = state;
    hand.setHolder( state );
}

protected final PlayerState getCurrentState() {
    return current_state;
}

protected PlayerState getBustedState() {
    return new Busted();
}

protected PlayerState getStandingState() {
    return new Standing();
}

protected PlayerState getPlayingState() {
    return new Playing();
}

protected PlayerState getWaitingState() {
    return new Waiting();
```

E

**LISTAGEM E.10** Player.java (*continuação*)

```
}
```

```
protected PlayerState getBlackjackState() {
    return new Blackjack();
}
```

```
protected abstract PlayerState getInitialState();
```

```
protected abstract boolean hit( Dealer dealer );
```

```
private class Waiting implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
        setCurrentState( getPlayingState() );
        // transição
    }
    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        // transição
    }
    public void handBusted() {
        // impossível no estado de espera
    }
    public void execute( Dealer dealer ) {
        // não faz nada enquanto espera
    }
}
```

```
private class Busted implements PlayerState {
    public void handChanged() {
        // impossível no estado de estouro
    }
    public void handPlayable() {
        // impossível no estado de estouro
    }
    public void handBlackjack() {
        // impossível no estado de estouro
    }
    public void handBusted() {
        // impossível no estado de estouro
    }
    public void execute( Dealer dealer ) {
```

**LISTAGEM E.10** Player.java (*continuação*)

```
        dealer.busted( Player.this );
        // termina
    }
}
private class Blackjack implements PlayerState {
    public void handChanged() {
        // impossível no estado de vinte-e-um
    }
    public void handPlayable() {
        // impossível no estado de vinte-e-um
    }
    public void handBlackjack() {
        // impossível no estado de vinte-e-um
    }
    public void handBusted() {
        // impossível no estado de vinte-e-um
    }
    public void execute( Dealer dealer ) {
        dealer.blackjack( Player.this );
        // termina
    }
}
private class Standing implements PlayerState {
    public void handChanged() {
        // impossível no estado de parada
    }
    public void handPlayable() {
        // impossível no estado de parada
    }
    public void handBlackjack() {
        // impossível no estado de parada
    }
    public void handBusted() {
        // impossível no estado de parada
    }
    public void execute( Dealer dealer ) {
        dealer.standing( Player.this );
        // termina
    }
}
private class Playing implements PlayerState {
    public void handChanged() {
        notifyChanged();
    }
    public void handPlayable() {
```

E

**LISTAGEM E.10** Player.java (*continuação*)

```
// pode ignorar no estado de jogo
}
public void handBlackjack() {
    // impossível no estado de jogo
}
public void handBusted() {
    setCurrentState( getBustedState() );
    notifyBusted();
}
public void execute( Dealer dealer ) {
    if( hit( dealer ) ) {
        dealer.hit( Player.this );
    } else {
        setCurrentState( getStandingState() );
        notifyStanding();
    }
    current_state.execute( dealer );
    // transição
}
}
```

---

**LISTAGEM E.11** PlayerListener.java

```
package blackjack.core;

public interface PlayerListener {

    public void playerChanged( Player player );

    public void playerBusted( Player player );

    public void playerBlackjack( Player player );

    public void playerStanding( Player player );

    public void playerWon( Player player );

    public void playerLost( Player player );

    public void playerStandoff( Player player );
}
```

---

**LISTAGEM E.12** PlayerState.java

```
package blackjack.core;

public interface PlayerState extends HandListener {

    public void execute( Dealer dealer );
}
```

**LISTAGEM E.13** Rank.java

```
package blackjack.core;

import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public final class Rank {

    public static final Rank TWO      = new Rank( 2, "2" );
    public static final Rank THREE   = new Rank( 3, "3" );
    public static final Rank FOUR    = new Rank( 4, "4" );
    public static final Rank FIVE   = new Rank( 5, "5" );
    public static final Rank SIX     = new Rank( 6, "6" );
    public static final Rank SEVEN  = new Rank( 7, "7" );
    public static final Rank EIGHT  = new Rank( 8, "8" );
    public static final Rank NINE   = new Rank( 9, "9" );
    public static final Rank TEN    = new Rank( 10, "10" );
    public static final Rank JACK   = new Rank( 10, "J" );
    public static final Rank QUEEN  = new Rank( 10, "Q" );
    public static final Rank KING   = new Rank( 10, "K" );
    public static final Rank ACE    = new Rank( 11, "A" );

    private static final Rank [] VALUES =
        { TWO, THREE, FOUR, FIVE, SIX, SEVEN,
          EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE };

    // fornece uma lista não modificável para fazer laço
    public static final List RANKS =
        Collections.unmodifiableList( Arrays.asList( VALUES ) );

    private final int    rank;
    private final String display;

    private Rank( int rank, String display ) {
        this.rank = rank;
```

E

**LISTAGEM E.13** Rank.java (*continuação*)

```
    this.display = display;
}

public int getRank() {
    return rank;
}

public String toString() {
    return display;
}
}
```

---

**LISTAGEM E.14** Suit.java

```
package blackjack.core;

import java.util.Collections;
import java.util.List;
import java.util.Arrays;

public final class Suit {

    //define estaticamente todos os valores válidos de Suit
    public static final Suit DIAMONDS = new Suit( (char)4 );
    public static final Suit HEARTS   = new Suit( (char)3 );
    public static final Suit SPADES   = new Suit( (char)6 );
    public static final Suit CLUBS    = new Suit( (char)5 );

    private static final Suit [] VALUES = { DIAMONDS, HEARTS, SPADES, CLUBS };

    // fornece uma lista não modificável para fazer laço
    public static final List SUITS =
        Collections.unmodifiableList( Arrays.asList( VALUES ) );

    // variável de instância para conter o valor de exibição
    private final char display;

    // não permite instanciação por objetos externos
    private Suit( char display ) {
        this.display = display;
    }

    // retorna o valor de Suit
    public String toString() {
```

**LISTAGEM E.14** Suit.java (*continuação*)

```
    return String.valueOf( display );
}
}
```

## blackjack.core.threaded

blackjack.core.threaded contém um objeto BlackjackDealer que coloca os jogadores em suas próprias linhas de execução (Listagem E.15).

**LISTAGEM E.15** ThreadedBlackjackDealer.java

```
package blackjack.core.threaded;

import blackjack.core.*;
import java.util.ArrayList;
import java.util.Iterator;

public class ThreadedBlackjackDealer extends BlackjackDealer {

    public ThreadedBlackjackDealer( String name, Hand hand, Deckpile cards ) {
        super( name, hand, cards );
    }

    protected PlayerState getWaitingState() {
        return new DealerWaiting();
    }

    protected PlayerState getCollectingBetsState() {
        return new DealerCollectingBets();
    }

    private class DealerCollectingBets implements PlayerState {
        public void handChanged() {
            // impossível no estado de aposta
        }
        public void handPlayable() {
            // impossível no estado de aposta
        }
        public void handBlackjack() {
            // impossível no estado de aposta
        }
        public void handBusted() {
            // impossível no estado de aposta
        }
    }
}
```

E

**LISTAGEM E.15 ThreadedBlackjackDealer.java (continuação)**

```
    }
    public void execute( final Dealer dealer ) {
        if( !betting_players.isEmpty() ) {
            final Player player = (Player) betting_players.get( 0 );
            betting_players.remove( player );
            Runnable runnable = new Runnable() {
                public void run() {
                    player.play( dealer );
                }
            };
            Thread threaded = new Thread( runnable );
            threaded.start();
        } else {
            setCurrentState( getDealingState() );
            getCurrentState().execute( dealer );
            // faz a transição e executa
        }
    }
}

private class DealerWaiting implements PlayerState {
    public void handChanged() {
        // impossível no estado de espera
    }
    public void handPlayable() {
        // impossível no estado de espera
    }
    public void handBlackjack() {
        // impossível no estado de espera
    }
    public void handBusted() {
        // impossível no estado de espera
    }
    public void execute( final Dealer d ) {
        if( !waiting_players.isEmpty() ) {
            final Player p = (Player) waiting_players.get( 0 );
            waiting_players.remove( p );
            Runnable r = new Runnable() {
                public void run() {
                    p.play( d );
                }
            };
            Thread t = new Thread( r );
            t.start();
        } else {
```

**LISTAGEM E.15** ThreadedBlackjackDealer.java (*continuação*)

```
        setCurrentState( getPlayingState() );
        exposeHand();
        getCurrentState().execute( d );
        // faz a transição e executa
    }
}
}
```

## blackjack.exe

blackjack.exe contém os executáveis para GUI MVC, GUI PAC, UI CLI e simulador (listagens E.16 a E.19).

**LISTAGEM E.16** BlackjackCLI.java

```
package blackjack.exe;

import blackjack.core.*;
import blackjack.players.*;
import blackjack.ui.*;

public class BlackjackCLI {

    public static void main( String [] args ) {
        Deckpile cards =new Deckpile();
        for(int i = 0; i < 4; i ++ ) {
            cards.shuffle();
            Deck deck = new Deck();
            deck.addToStack( cards );
            cards.shuffle();
        }

        Hand dealer_hand = new Hand();
        BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand, cards
    );Bank human_bank = new Bank( 1000 );
        Hand human_hand = new Hand();
        Player player = new CommandLinePlayer( "Human", human_hand, human_bank );
        dealer.addListener( Console.INSTANCE );
        player.addListener( Console.INSTANCE );
        dealer.addPlayer( player );

        do {
```

E

**LISTAGEM E.16** BlackjackCLI.java (*continuação*)

```
        dealer.newGame();
    } while( playAgain() );

    Console.INSTANCE.printMessage( "Thank you for playing!" );

}

private static boolean playAgain() {
    Console.INSTANCE.printMessage( "Would you like to play again?[Y]es [N]o" );
    String response =Console.INSTANCE.readInput( "invalid" );
    if( response.equalsIgnoreCase( "y" ) ) {
        return true;
    }
    return false;
}
}
```

---

**LISTAGEM E.17** BlackjackMVC.java

```
package blackjack.exe;

import blackjack.core.*;
import blackjack.core.threaded.*;
import blackjack.ui.mvc.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class BlackjackMVC extends JFrame {
    public static void main( String [] args ) {
        JFrame frame = new BlackjackMVC();
        frame.getContentPane().setBackground( FOREST_GREEN );
        frame.setSize( 580, 480 );
        frame.show();
    }

    private BlackjackDealer dealer;
    private GUIPlayer human;
    private JPanel players = new JPanel( new GridLayout( 0, 1 ) );

    private static final Color FOREST_GREEN = new Color( 35, 142, 35 );

    public BlackjackMVC() {
```

**LISTAGEM E.17** BlackjackMVC.java (*continuação*)

```
setUp();
WindowAdapter wa = new WindowAdapter() {
    public void windowClosing( WindowEvent e ) {
        System.exit( 0 );
    }
};
addWindowListener( wa );
}

// precisa ser protegido se houver subclasse
private PlayerView getPlayerView( Player player ) {
    PlayerView view = new PlayerView( player );
    view.setBackground( FOREST_GREEN );
    return view;
}

// precisa ser protegido se houver subclasse
private void setUp() {
    BlackjackDealer dealer = getDealer();
    PlayerView v1 = getPlayerView( dealer );

    GUIPlayer human = getHuman();
    PlayerView v2 = getPlayerView( human );

    PlayerView [] views = { v1, v2 };
    addPlayers( views );

    dealer.addPlayer( human );

    addOptionView( human, dealer );
}

// precisa ser protegido se houver subclasse
private void addPlayers( PlayerView [] pview ) {
    players.setBackground( FOREST_GREEN );
    for( int i = 0; i < pview.length; i ++ ) {
        players.add( pview [i] );
    }
    getContentPane().add( players, BorderLayout.CENTER );
}

private void addOptionView( GUIPlayer human, BlackjackDealer dealer ) {
    OptionView ov = new OptionView( human, dealer );
    ov.setBackground( FOREST_GREEN );
    getContentPane().add( ov, BorderLayout.SOUTH );
```

E

**LISTAGEM E.17** BlackjackMVC.java (*continuação*)

```
}

private BlackjackDealer getDealer() {
    if( dealer == null ) {
        Hand dealer_hand = new Hand();
        Deckpile cards = getCards();
        dealer = new ThreadedBlackjackDealer( "Dealer", dealer_hand, cards );
    }
    return dealer;
}

private GUIPlayer getHuman() {
    if( human == null ) {
        Hand human_hand = new Hand();
        Bank bank = new Bank( 1000 );
        human = new GUIPlayer( "Human", human_hand, bank );
    }
    return human;
}

private Deckpile getCards() {
    Deckpile cards = new Deckpile();
    for( int i = 0; i < 4; i ++ ) {
        cards.shuffle();
        Deck deck = new VDeck();
        deck.addToStack( cards );
        cards.shuffle();
    }
    return cards;
}
}
```

---

**LISTAGEM E.18** BlackjackPAC.java

```
package blackjack.exe;

import blackjack.core.*;
import blackjack.ui.pac.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class BlackjackPAC extends JFrame {
```

**LISTAGEM E.18** BlackjackPAC.java (*continuação*)

```
public static void main( String [] args ) {
    JFrame frame = new BlackjackPAC();
    frame.getContentPane().setBackground( FOREST_GREEN );
    frame.setSize( 580, 480 );
    frame.show();
}

private VPlayerFactory factory = new VPlayerFactory();
private JPanel players = new JPanel( new GridLayout( 0, 1 ) );

private static final Color FOREST_GREEN = new Color( 35, 142, 35 );

public BlackjackPAC() {
    setUp();
    WindowAdapter wa = new WindowAdapter() {
        public void windowClosing( WindowEvent e ) {
            System.exit( 0 );
        }
    };
    addWindowListener( wa );
}

// precisa ser protegido se houver subclasse
private void setUp() {
    VBlackjackDealer dealer = factory.getDealer();

    GUIPlayer human = factory.getHuman();

    dealer.addPlayer( human );

    players.add( dealer.view() );
    players.add( human.view() );
    getContentPane().add( players, BorderLayout.CENTER );
}
```

E

**LISTAGEM E.19** BlackjackSim.java

```
package blackjack.exe;

import blackjack.core.*;
import blackjack.ui.*;
import blackjack.players.*;
```

**LISTAGEM E.19** BlackjackSim.java

```
public class BlackjackSim {  
  
    public static void main( String [] args ) {  
  
        Console.INSTANCE.printMessage( "How many times should the simulator  
play?" );  
        String response = Console.INSTANCE.readInput( "invalid" );  
        int loops = Integer.parseInt( response );  
  
        Deckpile cards = new Deckpile();  
        for( int i = 0; i < 4; i ++ ) {  
            cards.shuffle();  
            Deck deck = new Deck();  
            deck.addToStack( cards );  
            cards.shuffle();  
        }  
  
        // cria uma banca  
        Hand dealer_hand = new Hand();  
        BlackjackDealer dealer = new BlackjackDealer( "Dealer", dealer_hand, cards );  
  
        // cria um OneHitPlayer  
        Bank one_bank = new Bank( 1000 );  
        Hand one_hand = new Hand();  
        Player oplayer = new OneHitPlayer( "OneHit", one_hand, one_bank );  
  
        // cria um SmartPlayer  
        Bank smart_bank = new Bank( 1000 );  
        Hand smart_hand = new Hand();  
        Player smplayer = new SmartPlayer( "Smart", smart_hand, smart_bank );  
  
        // cria um SafePlayer  
        Bank safe_bank = new Bank( 1000 );  
        Hand safe_hand = new Hand();  
        Player splayer = new SafePlayer( "Safe", safe_hand, safe_bank );  
  
        // cria um FlipPlayer  
        Bank flip_bank = new Bank( 1000 );  
        Hand flip_hand = new Hand();  
        Player fplayer = new FlipPlayer( "Flip", flip_hand, flip_bank );  
  
        // cria um jogador inteligente  
        Bank kn_bank = new Bank( 1000 );  
        Hand kn_hand = new Hand();
```

**LISTAGEM E.19** BlackjackSim.java (*continuação*)

```
Player knplayer = new KnowledgeablePlayer( "Knowledgeable", kn_hand,
kn_bank );

// cria um jogador "ótimo"
Bank opt_bank = new Bank( 1000 );
Hand opt_hand = new Hand();
Player optplayer = new OptimalPlayer( "Optimal", opt_hand, opt_bank );

// reúne todos os jogadores
dealer.addListener( Console.INSTANCE );
oplayer.addListener( Console.INSTANCE );
dealer.addPlayer( oplayer );
splayer.addListener( Console.INSTANCE );
dealer.addPlayer( splayer );
smplayer.addListener( Console.INSTANCE );
dealer.addPlayer( smplayer );
fplayer.addListener( Console.INSTANCE );
dealer.addPlayer( fplayer );
knplayer.addListener( Console.INSTANCE );
dealer.addPlayer( knplayer );
optplayer.addListener( Console.INSTANCE );
dealer.addPlayer( optplayer );

int counter = 0;
while( counter < loops ) {
    dealer.newGame();
    counter++;
}
}
```

E

## blackjack.players

blackjack.players contém os vários jogadores que foram criados no texto e em alguns dos exercícios (listagens E.20 a E.26).

**LISTAGEM E.20** CommandLinePlayer.java

```
package blackjack.players;

import blackjack.core.*;
import blackjack.ui.*;
```

**LISTAGEM E.20** CommandLinePlayer.java

```
public class CommandLinePlayer extends BettingPlayer {

    private final static String HIT = "H";
    private final static String STAND = "S";
    private final static String PLAY_MSG = "[H]it ou [S]tay ";
    private final static String BET_MSG = "Place Bet:[10] [50] or [100]";
    private final static String DD_MSG = "Double Down? [Y]es [N]o ";
    private final static String BET_10 = "10";
    private final static String BET_50 = "50";
    private final static String BET_100 = "100";
    private final static String NO = "N";
    private final static String YES = "Y";
    private final static String DEFAULT = "invalid";

    public CommandLinePlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    protected boolean hit( Dealer dealer ) {
        while( true ) {
            Console.INSTANCE.printMessage( PLAY_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equalsIgnoreCase( HIT ) ) {
                return true;
            } else if( response.equalsIgnoreCase( STAND ) ) {
                return false;
            }
            // se chegarmos até aqui, faz laço até obtermos uma entrada
            // significativa
        }
    }

    protected boolean doubleDown( Dealer dealer ) {
        while( true ) {
            Console.INSTANCE.printMessage( DD_MSG );
            String response = Console.INSTANCE.readInput( DEFAULT );
            if( response.equalsIgnoreCase( NO ) ) {
                return false;
            } else if( response.equalsIgnoreCase( YES ) ) {
                return true;
            }
            // se chegarmos até aqui, faz laço até obtermos uma entrada significativa
        }
    }
}
```

**LISTAGEM E.20** CommandLinePlayer.java (*continuação*)

```
protected void bet() {
    while( true ) {
        Console.INSTANCE.printMessage( BET_MSG );
        String response = Console.INSTANCE.readInput( DEFAULT );
        if( response.equals( BET_10 ) ) {
            getBank().place10Bet();
            return;
        }
        if( response.equals( BET_50 ) ) {
            getBank().place50Bet();
            return;
        }
        if( response.equals( BET_100 ) ) {
            getBank().place100Bet();
            return;
        }
        // se chegarmos até aqui, faz laço até obtermos uma entrada
        // significativa
    }
}
```

**LISTAGEM E.21** FlipPlayer.java

```
package blackjack.players;

import blackjack.core.*;

public class FlipPlayer extends BettingPlayer {

    private boolean hit = false;
    private boolean should_hit_once = false;

    public FlipPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit( Dealer dealer ) {
        if( should_hit_once && !hit ) {
            hit = true;
            return true;
        }
    }
}
```

E

**LISTAGEM E.21** FlipPlayer.java (*continuação*)

```
    return false;
}

public void reset() {
    super.reset();
    hit = false;
    should_hit_once = !should_hit_once;
}

public void bet() {
    getBank().place10Bet();
}

public boolean doubleDown( Dealer dealer ) {
    return false;
}
}
```

---

**LISTAGEM E.22** KnowledgeablePlayer.java

```
package blackjack.players;

import blackjack.core.*;

public class KnowledgeablePlayer extends BettingPlayer {

    public KnowledgeablePlayer( String name, Hand hand, Bank bank) {
        super( name, hand, bank );
    }

    public boolean doubleDown( Dealer dealer ) {
        int total = getHand().total();
        if( total == 10 || total == 11 ) {
            return true;
        }
        return false;
    }

    public boolean hit( Dealer dealer ) {

        int total = getHand().total();
        Card card = dealer.getUpCard();

        //nunca distribui mais cartas, independente de qual, se total > 15
        if( total > 15 ) {
```

**LISTAGEM E.22** KnowledgeablePlayer.java (*continuação*)

```
        return false;
    }

    //sempre distribui mais cartas para 11 e menos
    if( total <= 11 ) {
        return true;
    }

    //isso deixa 11, 12, 13, 14
    //baseia a decisão na banca

    if( card.getRank().getRank() > 7 ) {
        return true;
    }

    return false;

}

public void bet() {
    getBank().place10Bet();
}
}
```

**LISTAGEM E.23** OneHitPlayer.java

```
package blackjack.players;

import blackjack.core.*;

public class OneHitPlayer extends BettingPlayer {

    private boolean has_hit = false;

    public OneHitPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit( Dealer dealer ) {
        if( !has_hit ) {
            has_hit = true;
            return true;
        }
        return false;
    }
}
```

E

**LISTAGEM E.23** OneHitPlayer.java (*continuação*)

```
}

public void reset() {
    super.reset();
    has_hit = false;
}

public void bet() {
    getBank().place10Bet();
}

public boolean doubleDown( Dealer dealer ) {
    return false;
}
}
```

---

**LISTAGEM E.24** OptimalPlayer.java

```
package blackjack.players;

import blackjack.core.*;

public class OptimalPlayer extends BettingPlayer {

    public OptimalPlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean doubleDown( Dealer dealer ) {
        int total = getHand().total();
        Card card = dealer.getUpCard();
        if( total == 11 ) {
            return true;
        }
        if( total == 10 ) {
            if( card.getRank().getRank() != Rank.TEN.getRank() &&
                card.getRank() != Rank.ACE ) {
                return true;
            }
            return false;
        }
        if( total == 9 ) {
            if( card.getRank() == Rank.TWO ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR ||
                card.getRank() == Rank.FIVE ||
                card.getRank() == Rank.SIX ) {
                    return true;
                }
            return false;
        }
        if( total == 8 ) {
            if( card.getRank() == Rank.TWO ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR ||
                card.getRank() == Rank.FIVE ||
                card.getRank() == Rank.SIX ||
                card.getRank() == Rank.SEVEN ) {
                    return true;
                }
            return false;
        }
        if( total == 7 ) {
            if( card.getRank() == Rank.TWO ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR ||
                card.getRank() == Rank.FIVE ||
                card.getRank() == Rank.SIX ) {
                    return true;
                }
            return false;
        }
        if( total == 6 ) {
            if( card.getRank() == Rank.TWO ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR ||
                card.getRank() == Rank.FIVE ) {
                    return true;
                }
            return false;
        }
        if( total == 5 ) {
            if( card.getRank() == Rank.TWO ||
                card.getRank() == Rank.THREE ||
                card.getRank() == Rank.FOUR ) {
                    return true;
                }
            return false;
        }
        if( total == 4 ) {
            if( card.getRank() == Rank.TWO ||
                card.getRank() == Rank.THREE ) {
                    return true;
                }
            return false;
        }
        if( total == 3 ) {
            if( card.getRank() == Rank.TWO ) {
                return true;
            }
            return false;
        }
        if( total == 2 ) {
            if( card.getRank() == Rank.TWO ) {
                return true;
            }
            return false;
        }
        if( total == 1 ) {
            if( card.getRank() == Rank.TWO ) {
                return true;
            }
            return false;
        }
        return false;
    }
}
```

**LISTAGEM E.24** OptimalPlayer.java (*conmtinuação*)

```
        card.getRank() == Rank.FOUR ||
        card.getRank() == Rank.FIVE ||
        card.getRank() == Rank.SIX ) {
            return true;
        }
        return false;
    }
    return false;
}

public boolean hit( Dealer dealer ) {

    int total = getHand().total();
    Card card = dealer.getUpCard();

    if( total >= 17 ) {
        return false;
    }

    if( total == 16 ) {
        if( card.getRank() == Rank.SEVEN ||
            card.getRank() == Rank.EIGHT ||
            card.getRank() == Rank.NINE ) {
                return true;
            } else {
                return false;
            }
    }
    if( total == 13 || total == 14 || total == 15 ) {
        if( card.getRank() == Rank.TWO ||
            card.getRank() == Rank.THREE ||
            card.getRank() == Rank.FOUR ||
            card.getRank() == Rank.FIVE ||
            card.getRank() == Rank.SIX ) {
                return false;
            } else {
                return true;
            }
    }
    if( total == 12 ) {
        if( card.getRank() == Rank.FOUR ||
            card.getRank() == Rank.FIVE ||
            card.getRank() == Rank.SIX ) {
                return false;
            } else {
```

E

**LISTAGEM E.24** OptimalPlayer.java (*conmtinuação*)

```
        return true;
    }
}
return true;
}

public void bet() {
    getBank().place10Bet();
}
}
```

---

**LISTAGEM E.25** SafePlayer.java

```
package blackjack.players;

import blackjack.core.*;

public class SafePlayer extends BettingPlayer {

    public SafePlayer( String name, Hand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public boolean hit( Dealer dealer ) {
        return false;
    }

    public boolean doubleDown( Dealer dealer ) {
        return false;
    }

    public void bet() {
        getBank().place10Bet();
    }
}
```

---

**LISTAGEM E.26** SmartPlayer.java

```
package blackjack.players;

import blackjack.core.*;

public class SmartPlayer extends BettingPlayer {
```

**LISTAGEM E.26** SmartPlayer.java (*continuação*)

```
public SmartPlayer( String name, Hand hand, Bank bank ) {
    super( name, hand, bank );
}

public boolean hit( Dealer dealer ) {
    if( getHand().total() > 11 ) {
        return false;
    }
    return true;
}

public void bet() {
    getBank().place10Bet();
}

public boolean doubleDown( Dealer dealer ) {
    return false;
}
}
```

## blackjack.ui

blackjack.ui contém código da UI comum (Listagem E.27).

**LISTAGEM E.27** Console.java

```
package blackjack.ui;

import blackjack.core.*;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class Console implements PlayerListener {

    // console singleton
    public final static Console INSTANCE = new Console();

    private BufferedReader in =
        new BufferedReader( new InputStreamReader( System.in ) );

    public void printMessage( String message ) {
        System.out.println( message );
    }
}
```

**LISTAGEM E.27** Console.java (*continuação*)

```
}

public String readInput( String default_input ) {
    String response;
    try {
        return in.readLine();
    } catch (IOException ioe) {
        return default_input;
    }
}

public void playerChanged( Player player ) {
    printMessage( player.toString() );
}

public void playerBusted( Player player ) {
    printMessage( player.toString() + "BUSTED!" );
}

public void playerBlackjack( Player player ) {
    printMessage( player.toString() + "BLACKJACK!" );
}

public void playerStanding( Player player ){
    printMessage( player.toString() + "STANDING " );
}

public void playerWon( Player player ) {
    printMessage( player.toString() + "WINNER!" );
}

public void playerLost( Player player ) {
    printMessage( player.toString() + "LOSER!" );
}

public void playerStandoff( Player player ) {
    printMessage( player.toString() + "STANDOFF" );
}

//privado para evitar instanciação
private Console() {}

}
```

---

**blackjack.ui.mvc**

blackjack.ui.mvc contém o código mvc (listagens E.28 a E.34).

**LISTAGEM E.28** CardView.java

```
package blackjack.ui.mvc;

import blackjack.core.*;
import javax.swing.*;
import java.awt.*;

public class CardView extends JLabel {

    private ImageIcon icon;

    public CardView( VCard card ) {
        getImage( card.getImage() );
        setIcon( icon );
        setBackground( Color.white );
        setOpaque( true );
    }

    private void getImage( String name ) {
        java.net.URL url = this.getClass().getResource( name );
        icon = new ImageIcon( url );
    }
}
```

**LISTAGEM E.29** GUIPlayer.java

```
package blackjack.ui.mvc;

import blackjack.core.*;

public class GUIPlayer extends BettingPlayer {

    private Dealer dealer;

    public GUIPlayer( String name, Hand hand, Bank bank) {
        super( name, hand, bank );
    }

    public boolean hit( Dealer dealer ) {
        return true;
    }

    public void bet() {
        // não faz nada, isto não será chamado
        // em vez disso, o jogador humano pressiona um botão da GUI
    }
}
```

E

**LISTAGEM E.29 GUIPlayer.java (continuação)**

```
}
```

```
// esses métodos de aposta serão chamados pelo controlador da GUI
// para cada um: faz a aposta correta, muda o estado, permite que a
// banca saiba que o jogador terminou de apostar
public void place10Bet() {
    getBank().place10Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

public void place50Bet() {
    getBank().place50Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

public void place100Bet() {
    getBank().place100Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

// a aposta em dobro é um pouco diferente, pois o jogador precisa
// responder aos eventos de Hand quando uma carta é adicionada na mão
// de modo que configura o estado como DoublingDown e depois o executa
protected boolean doubleDown( Dealer d ) {
    setCurrentState( getDoublingDownState() );
    getCurrentState().execute( dealer );
    return true;
}

// takeCard será chamado pelo controlador da GUI quando o jogador
// decidir receber mais cartas
public void takeCard() {
    dealer.hit( this );
}

// stand será chamado pelo controlador da GUI quando o jogador optar
// por parar, quando a parada mudar de estado, deixa o mundo saber, e depois
// diz à banca
public void stand() {
    setCurrentState( getStandingState() );
    notifyStanding();
    getCurrentState().execute( dealer );
```

**LISTAGEM E.29** GUIPlayer.java (*continuação*)

```
}

// você precisa sobrepor play para que ele armazene a banca para
// uso posterior
public void play( Dealer dealer ) {
    this.dealer = dealer;
    super.play( dealer );
}

// o seguinte trata dos estados
protected PlayerState getPlayingState() {
    return new Playing();
}

protected PlayerState getBettingState() {
    return new Betting();
}

private class Playing implements PlayerState {

    public void handPlayable() {
        // não faz nada
    }

    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        getCurrentState().execute( dealer );
    }

    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
        getCurrentState().execute( dealer );
    }

    public void handChanged() {
        notifyChanged();
    }

    public void execute( Dealer dealer ) {
        // não faz nada aqui, as ações virão da GUI, que é
        // externa ao estado, mas quando eventos vierem, certifica-se de
        // forçar a transição de estado imediatamente
    }
}
```

E

**LISTAGEM E.29** GUIPlayer.java (*continuação*)

```
}

private class Betting implements PlayerState {
    public void handChanged() {
        // impossível no estado de estouro
    }
    public void handPlayable() {
        // impossível no estado de estouro
    }
    public void handBlackjack() {
        // impossível no estado de estouro
    }
    public void handBusted() {
        // impossível no estado de estouro
    }
    public void execute( Dealer dealer ) {
        // não faz nada aqui, as ações virão da GUI, que é
        // externa ao estado, pois nenhum evento aparece como parte da
        // aposta, o estado precisará ser mudado externamente para este estado
    }
}
}
```

---

**LISTAGEM E.30** OptionView.java

```
package blackjack.ui.mvc;

import blackjack.core.*;
import javax.swing.*;
import java.awt.*;

public class OptionView extends JPanel {

    public static final String NEW_GAME      = "new";
    public static final String QUIT          = "quit";
    public static final String HIT           = "hit";
    public static final String STAND         = "stand";
    public static final String BET_10        = "BET10";
    public static final String BET_50        = "BET50";
    public static final String BET_100       = "BET100";
    public static final String DOUBLE_DOWN   = "dd";

    private JButton bet_10   = new JButton( "$10" );
    private JButton bet_50   = new JButton( "$50" );
    private JButton bet_100  = new JButton( "$100" );
```

**LISTAGEM E.30** OptionView.java (*continuação*)

```
private JButton deal = new JButton( "New Game" );
private JButton quit = new JButton( "Quit" );
private JButton hit = new JButton( "Hit" );
private JButton stand = new JButton( "Stand" );
private JButton ddown = new JButton( "Double Down" );
private BlackjackDealer dealer;
private GUIPlayer player;

private static final Color FOREST_GREEN = new Color( 35, 142, 35 );

public OptionView( GUIPlayer player, BlackjackDealer dealer ) {
    super( new BorderLayout() );
    this.player = player;
    this.dealer = dealer;
    attachController( makeController() );
    buildGUI();
}

public void attachController( OptionViewController controller ){
    deal.addActionListener( controller );
    quit.addActionListener( controller );
    hit.addActionListener( controller );
    stand.addActionListener( controller );
    bet_10.addActionListener( controller );
    bet_50.addActionListener( controller );
    bet_100.addActionListener( controller );
    ddown.addActionListener( controller );
}

public void enableDoubleDown( boolean enable ) {
    ddown.setEnabled( enable );
}

public void enableBettingControls( boolean enable ) {
    bet_10.setEnabled( enable );
    bet_50.setEnabled( enable );
    bet_100.setEnabled( enable );
}

public void enablePlayerControls( boolean enable ) {
    hit.setEnabled( enable );
    stand.setEnabled( enable );
}

public void enableGameControls( boolean enable ) {
```

E

**LISTAGEM E.30** OptionView.java (*continuação*)

```
    deal.setEnabled( enable );
    quit.setEnabled( enable );
}

protected OptionViewController makeController() {
    return new OptionViewController( player, dealer, this );
}

private void buildGUI() {
    JPanel betting_controls = new JPanel();
    JPanel game_controls = new JPanel();
    add( betting_controls, BorderLayout.NORTH );
    add( game_controls, BorderLayout.SOUTH );
    betting_controls.setBackground( FOREST_GREEN );
    game_controls.setBackground( FOREST_GREEN );
    ddown.setActionCommand( DOUBLE_DOWN );
    deal.setActionCommand( NEW_GAME );
    quit.setActionCommand( QUIT );
    hit.setActionCommand( HIT );
    stand.setActionCommand( STAND );
    bet_10.setActionCommand( BET_10 );
    bet_50.setActionCommand( BET_50 );
    bet_100.setActionCommand( BET_100 );
    betting_controls.add( bet_10 );
    betting_controls.add( bet_50 );
    betting_controls.add( bet_100 );
    game_controls.add( ddown );
    game_controls.add( hit );
    game_controls.add( stand );
    game_controls.add( deal );
    game_controls.add( quit );
    enableBettingControls( false );
    enablePlayerControls( false );
    enableDoubleDown( false );
}
}
```

---

**LISTAGEM E.31** OptionViewController.java

```
package blackjack.ui.mvc;

import blackjack.core.*;
import java.awt.event.*;
```

**LISAGEM E.31** OptionViewController.java (*continuação*)

```
public class OptionViewController implements ActionListener, PlayerListener {  
  
    private GUIPlayer model;  
    private OptionView view;  
    private BlackjackDealer dealer;  
  
    public OptionViewController( GUIPlayer model, BlackjackDealer dealer,  
OptionView view ) {  
        this.model = model;  
        model.addListener( this );  
        this.dealer = dealer;  
        this.view = view;  
        view.enablePlayerControls( false );  
    }  
  
    public void actionPerformed( ActionEvent event ) {  
        if( event.getActionCommand().equals( OptionView.QUIT ) ) {  
            System.exit( 0 );  
        } else if( event.getActionCommand().equals( OptionView.HIT ) ) {  
            view.enableDoubleDown( false );  
            model.takeCard();  
        } else if( event.getActionCommand().equals( OptionView.STAND ) ) {  
            view.enableDoubleDown( false );  
            model.stand();  
        } else if( event.getActionCommand().equals( OptionView.NEW_GAME ) ) {  
            view.enableDoubleDown( false );  
            view.enableGameControls( false );  
            view.enablePlayerControls( false );  
            view.enableBettingControls( true );  
            dealer.newGame();  
        } else if( event.getActionCommand().equals( OptionView.BET_10 ) ) {  
            view.enableBettingControls( false );  
            view.enablePlayerControls( true );  
            view.enableDoubleDown( true );  
            model.place10Bet();  
        } else if( event.getActionCommand().equals( OptionView.BET_50 ) ) {  
            view.enableBettingControls( false );  
            view.enablePlayerControls( true );  
            view.enableDoubleDown( true );  
            model.place50Bet();  
        } else if( event.getActionCommand().equals( OptionView.BET_100 ) ) {  
            view.enableBettingControls( false );  
            view.enablePlayerControls( true );  
            view.enableDoubleDown( true );  
            model.place100Bet();  
        }  
    }  
}
```

E

**LISTAGEM E.31 OptionViewController.java (*continuação*)**

```
    } else if( event.getActionCommand().equals( OptionView.DOUBLE_DOWN ) ) {
        view.enableBettingControls( false );
        view.enablePlayerControls( false );
        view.enableDoubleDown( false );
        view.enableGameControls( true );
        model.doubleDown( dealer );
    }

}

public void playerChanged( Player player ) {}

public void playerBusted( Player player ) {
    view.enablePlayerControls( false );
    view.enableDoubleDown( false );
    view.enableGameControls( true );
}

public void playerBlackjack( Player player ) {
    view.enablePlayerControls( false );
    view.enableDoubleDown( false );
    view.enableGameControls( true );
}

public void playerStanding( Player player ) {
    view.enablePlayerControls( false );
    view.enableGameControls( true );
}

public void playerWon( Player player ) {
    view.enablePlayerControls( false );
    view.enableGameControls( true );
}

public void playerLost( Player player ) {
    view.enablePlayerControls( false );
    view.enableDoubleDown( false );
    view.enableGameControls( true );
}

public void playerStandoff( Player player ) {
    view.enablePlayerControls( false );
    view.enableGameControls( true );
}
```

---

**LISAGEM E.32** PlayerView.java

```
package blackjack.ui.mvc;

import blackjack.core.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.util.Iterator;

public class PlayerView extends JPanel implements PlayerListener {

    private JPanel cards = new JPanel(new FlowLayout(FlowLayout.LEFT));
    private TitledBorder border;

    public PlayerView( Player player ) {
        super( new BorderLayout() );
        buildUI( player );
        player.addListener( this );
    }

    public void playerChanged( Player player ) {
        border.setTitle( player.getName() );
        cards.removeAll();
        Hand hand = player.getHand();
        Iterator i = hand.getCards();
        while( i.hasNext() ) {
            VCard vcard = (Vcard) i.next();
            JLabel card = new CardView( vcard );
            cards.add( card );
        }
        revalidate();
        repaint();
    }

    public void playerBusted( Player player ) {
        border.setTitle( player.getName() + " BUSTED!" );
        cards.repaint();
    }

    public void playerBlackjack( Player player ){
        border.setTitle( player.getName() + " BLACKJACK!" );
        cards.repaint();
    }

    public void playerStanding( Player player ) {
        border.setTitle( player.getName() + " STANDING " );
        cards.repaint();
    }
}
```

E

**LISTAGEM E.32 PlayerView.java (*continuação*)**

---

```
}

public void playerWon( Player player ) {
    border.setTitle( player.getName() + " WINNER!" );
    cards.repaint();
}

public void playerLost( Player player ) {
    border.setTitle( player.getName() + " LOSER!" );
    cards.repaint();
}

public void playerStandoff( Player player ) {
    border.setTitle( player.getName() + " STANDOFF!" );
    cards.repaint();
}

private void buildUI( Player player ) {
    add( cards, BorderLayout.NORTH );
    border = new TitledBorder( player.getName() );
    cards.setBorder( border );
    cards.setBackground( new Color( 35, 142, 35 ) );
    border.setTitleColor( Color.black );
}
}
```

---

**LISTAGEM E.33 VCard.java**

---

```
package blackjack.ui.mvc;

import blackjack.core.*;

public class VCard extends Card {

    private String image;

    public Vcard( Suit suit, Rank rank, String image ) {
        super( suit, rank );
        this.image = image;
    }

    public String getImage() {
        if( isFaceUp() ) {
            return image;
```

**LISTAGEM E.33** VCard.java (*continuação*)

```
    } else {
        return "/blackjack/ui/bitmaps/empty_pile.xbm";
    }
}
```

**LISTAGEM E.34** VDeck.java

```
package blackjack.ui.mvc;

import blackjack.core.*;
import java.util.Iterator;

public class VDeck extends Deck {

    protected void buildCards() {

        // Isso é horrível, mas é melhor do que os laços e estruturas
        // condicionais alternativas
        Card [] deck = new Card [52];
        setDeck( deck );

        deck [0] = new Vcard( Suit.HEARTS, Rank.TWO,
        "/blackjack/ui/bitmaps/h2" );
        deck [1] = new Vcard( Suit.HEARTS, Rank.THREE,
        "/blackjack/ui/bitmaps/h3" );
        deck [2] = new Vcard( Suit.HEARTS, Rank.FOUR,
        "/blackjack/ui/bitmaps/h4" );
        deck [3] = new Vcard( Suit.HEARTS, Rank.FIVE,
        "/blackjack/ui/bitmaps/h5" );
        deck [4] = new Vcard( Suit.HEARTS, Rank.SIX,
        "/blackjack/ui/bitmaps/h6" );
        deck [5] = new Vcard( Suit.HEARTS, Rank.SEVEN,
        "/blackjack/ui/bitmaps/h7" );
        deck [6] = new Vcard( Suit.HEARTS, Rank.EIGHT,
        "/blackjack/ui/bitmaps/h8" );
        deck [7] = new Vcard( Suit.HEARTS, Rank.NINE,
        "/blackjack/ui/bitmaps/h9" );
        deck [8] = new Vcard( Suit.HEARTS, Rank.TEN,
        "/blackjack/ui/bitmaps/h10" );
        deck [9] = new Vcard( Suit.HEARTS, Rank.JACK,
        "/blackjack/ui/bitmaps/h11" );
        deck [10] = new Vcard( Suit.HEARTS, Rank.QUEEN,
        "/blackjack/ui/bitmaps/h12" );
```

E

**LISTAGEM E.34 VDeck.java (continuação)**

```
deck [11] = new Vcard( Suit.HEARTS, Rank.KING,
"/blackjack/ui/bitmaps/h13" );
    deck [12] = new Vcard( Suit.HEARTS, Rank.ACE,
"/blackjack/ui/bitmaps/h1" );
    deck [13] = new Vcard( Suit.DIAMONDS, Rank.TWO,
"/blackjack/ui/bitmaps/d2" );
    deck [14] = new Vcard( Suit.DIAMONDS, Rank.THREE,
"/blackjack/ui/bitmaps/d3" );
    deck [15] = new Vcard( Suit.DIAMONDS, Rank.FOUR,
"/blackjack/ui/bitmaps/d4" );
    deck [16] = new Vcard( Suit.DIAMONDS, Rank.FIVE,
"/blackjack/ui/bitmaps/d5" );
    deck [17] = new Vcard( Suit.DIAMONDS, Rank.SIX,
"/blackjack/ui/bitmaps/d6" );
    deck [18] = new Vcard( Suit.DIAMONDS, Rank.SEVEN,
"/blackjack/ui/bitmaps/d7" );
    deck [19] = new Vcard( Suit.DIAMONDS, Rank.EIGHT,
"/blackjack/ui/bitmaps/d8" );
    deck [20] = new Vcard( Suit.DIAMONDS, Rank.NINE,
"/blackjack/ui/bitmaps/d9" );
    deck [21] = new Vcard( Suit.DIAMONDS, Rank.TEN,
"/blackjack/ui/bitmaps/d10" );
    deck [22] = new Vcard( Suit.DIAMONDS, Rank.JACK,
"/blackjack/ui/bitmaps/d11" );
    deck [23] = new Vcard( Suit.DIAMONDS, Rank.QUEEN,
"/blackjack/ui/bitmaps/d12" );
    deck [24] = new Vcard( Suit.DIAMONDS, Rank.KING,
"/blackjack/ui/bitmaps/d13" );
    deck [25] = new Vcard( Suit.DIAMONDS, Rank.ACE,
"/blackjack/ui/bitmaps/d1" );
    deck [26] = new Vcard( Suit.SPADES, Rank.TWO,
"/blackjack/ui/bitmaps/s2" );
    deck [27] = new Vcard( Suit.SPADES, Rank.THREE,
"/blackjack/ui/bitmaps/s3" );
    deck [28] = new Vcard( Suit.SPADES, Rank.FOUR,
"/blackjack/ui/bitmaps/s4" );
    deck [29] = new Vcard( Suit.SPADES, Rank.FIVE,
"/blackjack/ui/bitmaps/s5" );
    deck [30] = new Vcard( Suit.SPADES, Rank.SIX,
"/blackjack/ui/bitmaps/s6" );
    deck [31] = new Vcard( Suit.SPADES, Rank.SEVEN,
"/blackjack/ui/bitmaps/s7" );
    deck [32] = new Vcard( Suit.SPADES, Rank.EIGHT,
"/blackjack/ui/bitmaps/s8" );
    deck [33] = new Vcard( Suit.SPADES, Rank.NINE,
```

**LISTAGEM E.34** VDeck.java

```
"/blackjack/ui/bitmaps/s9" );
    deck [34] = new Vcard( Suit.SPADES, Rank.TEN,
"/blackjack/ui/bitmaps/s10" );
    deck [35] = new Vcard( Suit.SPADES, Rank.JACK,
"/blackjack/ui/bitmaps/s11" );
    deck [36] = new Vcard( Suit.SPADES, Rank.QUEEN,
"/blackjack/ui/bitmaps/s12" );
    deck [37] = new Vcard( Suit.SPADES, Rank.KING,
"/blackjack/ui/bitmaps/s13" );
    deck [38] = new Vcard( Suit.SPADES, Rank.ACE,
"/blackjack/ui/bitmaps/s1" );
    deck [39] = new Vcard( Suit.CLUBS, Rank.TWO,
"/blackjack/ui/bitmaps/c2" );
    deck [40] = new Vcard( Suit.CLUBS, Rank.THREE,
"/blackjack/ui/bitmaps/c3" );
    deck [41] = new Vcard( Suit.CLUBS, Rank.FOUR,
"/blackjack/ui/bitmaps/c4" );
    deck [42] = new Vcard( Suit.CLUBS, Rank.FIVE,
"/blackjack/ui/bitmaps/c5" );
    deck [43] = new Vcard( Suit.CLUBS, Rank.SIX,
"/blackjack/ui/bitmaps/c6" );
    deck [44] = new Vcard( Suit.CLUBS, Rank.SEVEN,
"/blackjack/ui/bitmaps/c7" );
    deck [45] = new Vcard( Suit.CLUBS, Rank.EIGHT,
"/blackjack/ui/bitmaps/c8" );
    deck [46] = new Vcard( Suit.CLUBS, Rank.NINE,
"/blackjack/ui/bitmaps/c9" );
    deck [47] = new Vcard( Suit.CLUBS, Rank.TEN,
"/blackjack/ui/bitmaps/c10" );
    deck [48] = new Vcard( Suit.CLUBS, Rank.JACK,
"/blackjack/ui/bitmaps/c11" );
    deck [49] = new Vcard( Suit.CLUBS, Rank.QUEEN,
"/blackjack/ui/bitmaps/c12" );
    deck [50] = new Vcard( Suit.CLUBS, Rank.KING,
"/blackjack/ui/bitmaps/c13" );
    deck [51] = new Vcard( Suit.CLUBS, Rank.ACE,
"/blackjack/ui/bitmaps/c1" );
}
```

E

**blackjack.ui.pac**

blackjack.ui.pac contém o código pac (listagens E.35 a E.42).

**Listagem E.35** Displayable.java

```
package blackjack.ui.pac;

import javax.swing.JComponent;

public interface Displayable {
    public JComponent view();
}
```

---

**Listagem E.36** GUIPlayer.java

```
package blackjack.ui.pac;

import blackjack.core.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class GUIPlayer extends VBettingPlayer implements Displayable {

    private BlackjackDealer dealer;
    private JPanel view;

    public GUIPlayer( String name, VHand hand, Bank bank, VblackjackDealer
➥dealer ) {
        super( name, hand, bank );
        this.dealer = dealer;
    }

    public boolean hit( Dealer dealer ) {
        return true;
    }

    public void bet() {
        // não faz nada, isto não será chamado
        // em vez disso, o jogador humano pressiona um botão da GUI
    }

    // esses métodos de aposta serão chamados pelo controlador da GUI
    // para cada um: faz a aposta correta, muda o estado, permite que a
    // banca saiba que o jogador terminou de apostar
    public void place10Bet() {
        getBank().place10Bet();
        setCurrentState( getWaitingState() );
        dealer.doneBetting( this );
    }
}
```

**LISTAGEM E.36 GUIPlayer.java (continuação)**

```
}

public void place50Bet() {
    getBank().place50Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

public void place100Bet() {
    getBank().place100Bet();
    setCurrentState( getWaitingState() );
    dealer.doneBetting( this );
}

// a aposta em dobro é um pouco diferente, pois o jogador precisa
// responder aos eventos de Hand quando uma carta é adicionada na mão
// de modo que configura o estado como DoublingDown e depois o executa
protected boolean doubleDown( Dealer d ) {
    setCurrentState( getDoublingDownState() );
    getCurrentState().execute( dealer );
    return true;
}

// takeCard será chamado pelo controlador da GUI quando o jogador
// decidir receber mais cartas
public void takeCard() {
    dealer.hit( this );
}

// stand será chamado pelo controlador da GUI quando o jogador optar
// por parar, quando a parada mudar de estado, deixa o mundo saber, e depois
// diz à banca
public void stand() {
    setCurrentState( getStandingState() );
    notifyStanding();
    getCurrentState().execute( dealer );
}

public JComponent view() {
    if( view == null ) {
        view = new JPanel( new BorderLayout() );
        JComponent pv = super.view();
        GUIView cv = new GUIView();
        addListener( cv );
        view.add( pv, BorderLayout.CENTER );
```

E

**LISTAGEM E.36 GUIPlayer.java (continuação)**

---

```
        view.add( cv, BorderLayout.SOUTH );
    }
    return view;
}

// o seguinte trata dos estados
protected PlayerState getPlayingState() {
    return new Playing();
}

protected PlayerState getBettingState() {
    return new Betting();
}

private class Playing implements PlayerState {

    public void handPlayable() {
        // não faz nada
    }

    public void handBlackjack() {
        setCurrentState( getBlackjackState() );
        notifyBlackjack();
        getCurrentState().execute( dealer );
    }

    public void handBusted() {
        setCurrentState( getBustedState() );
        notifyBusted();
        getCurrentState().execute( dealer );
    }

    public void handChanged() {
        notifyChanged();
    }

    public void execute( Dealer dealer ) {
        // não faz nada aqui, as ações virão da GUI, que é
        // externa ao estado, mas quando eventos vierem, certifica-se de
        // forçar a transição de estado imediatamente
    }
}
private class Betting implements PlayerState {
    public void handChanged() {
        // impossível no estado de estouro
```

**LISTAGEM E.36 GUIPlayer.java (continuação)**

```
    }

    public void handPlayable() {
        // impossível no estado de estouro
    }

    public void handBlackjack() {
        // impossível no estado de estouro
    }

    public void handBusted() {
        // impossível no estado de estouro
    }

    public void execute( Dealer dealer ) {
        // não faz nada aqui, as ações virão da GUI, que é
        // externa ao estado, pois nenhum evento aparece como parte da
        // aposta, o estado precisará ser mudado externamente para este estado
    }
}

private class GUIView extends JPanel implements PlayerListener, ActionListener
{

    private JButton bet_10  = new JButton( "$10" );
    private JButton bet_50  = new JButton( "$50" );
    private JButton bet_100 = new JButton( "$100" );
    private JButton deal   = new JButton( "New Game" );
    private JButton quit   = new JButton( "Quit" );
    private JButton hit    = new JButton( "Hit" );
    private JButton stand  = new JButton( "Stand" );
    private JButton ddown  = new JButton( "Double Down" );

    private final String NEW_GAME = "new";
    private final String QUIT     = "quit";
    private final String HIT      = "hit";
    private final String STAND    = "stand";
    private final String BET_10   = "BET10";
    private final String BET_50   = "BET50";
    private final String BET_100  = "BET100";
    private final String D_DOWN   = "DDown";

    private final Color FOREST_GREEN = new Color( 35, 142, 35 );

    public GUIView() {
        super( new BorderLayout() );
        GUIPlayer.this.addListener( this );
        buildGUI();
    }
}
```

E

**LISTAGEM E.36** GUIPlayer.java (*continuação*)

```
private void buildGUI() {
    JPanel betting_controls = new JPanel();
    JPanel game_controls   = new JPanel();

    add( betting_controls, BorderLayout.NORTH );
    add( game_controls, BorderLayout.SOUTH );

    betting_controls.setBackground( FOREST_GREEN );
    game_controls.setBackground( FOREST_GREEN );
    deal.setActionCommand( NEW_GAME );
    deal.addActionListener( this );
    quit.setActionCommand( QUIT );
    quit.addActionListener( this );
    hit.setActionCommand( HIT );
    hit.addActionListener( this );
    stand.setActionCommand( STAND );
    stand.addActionListener( this );
    bet_10.setActionCommand( BET_10 );
    bet_10.addActionListener( this );
    bet_50.setActionCommand( BET_50 );
    bet_50.addActionListener( this );
    bet_100.setActionCommand( BET_100 );
    bet_100.addActionListener( this );
    ddown.setActionCommand( D_DOWN );
    ddown.addActionListener( this );
    betting_controls.add( bet_10 );
    betting_controls.add( bet_50 );
    betting_controls.add( bet_100 );
    game_controls.add( ddown );
    game_controls.add( hit );
    game_controls.add( stand );
    game_controls.add( deal );
    game_controls.add( quit );
    enableBettingControls( false );
    enablePlayerControls( false );
    enableDoubleDown( false );
}

private void enableBettingControls( boolean enable ) {
    bet_10.setEnabled( enable );
    bet_50.setEnabled( enable );
    bet_100.setEnabled( enable );
}

private void enablePlayerControls( boolean enable ) {
```

**LISTAGEM E.36** GUIPlayer.java (*continuação*)

```
    hit.setEnabled( enable );
    stand.setEnabled( enable );
}

private void enableGameControls( boolean enable ) {
    deal.setEnabled( enable );
    quit.setEnabled( enable );
}

private void enableDoubleDown( boolean enable ) {
    ddown.setEnabled( enable );
}

public void actionPerformed( ActionEvent event ) {
    if( event.getActionCommand().equals( QUIT ) ) {
        System.exit( 0 );
    }else if( event.getActionCommand().equals( HIT ) ) {
        enableDoubleDown( false );
        takeCard();
    }else if( event.getActionCommand().equals( STAND ) ) {
        enableDoubleDown( false );
        stand();
    }else if( event.getActionCommand().equals( NEW_GAME ) ) {
        enableDoubleDown( false );
        enableGameControls( false );
        enablePlayerControls( false );
        enableBettingControls( true );
        dealer.newGame();
    }else if( event.getActionCommand().equals( BET_10 ) ) {
        enableDoubleDown( true );
        enableBettingControls( false );
        enablePlayerControls( true );
        place10Bet();
    }else if( event.getActionCommand().equals( BET_50 ) ) {
        enableDoubleDown( true );
        enableBettingControls( false );
        enablePlayerControls( true );
        place50Bet();
    }else if( event.getActionCommand().equals( BET_100 ) ) {
        enableDoubleDown( true );
        enableBettingControls( false );
        enablePlayerControls( true );
        place100Bet();
    }else if( event.getActionCommand().equals( D_DOWN ) ) {
        enablePlayerControls( false );
    }
}
```

E

**LISTAGEM E.36** GUIPlayer.java (*continuação*)

```
        enableDoubleDown( false );
        doubleDown( dealer );
    }

    public void playerChanged( Player player ) {}

    public void playerBusted( Player player ) {
        enablePlayerControls( false );
        enableGameControls( true );
    }

    public void playerBlackjack( Player player ) {
        enableDoubleDown( false );
        enablePlayerControls( false );
        enableGameControls( true );
    }

    public void playerStanding( Player player ) {
        enablePlayerControls( false );
        enableGameControls( true );
    }

    public void playerWon( Player player ) {
        enablePlayerControls( false );
        enableGameControls( true );
    }

    public void playerLost( Player player ) {
        enableDoubleDown( false );
        enablePlayerControls( false );
        enableGameControls( true );
    }

    public void playerStandoff( Player player ) {
        enablePlayerControls( false );
        enableGameControls( true );
    }
}
```

**LISTAGEM E.37** VBettingPlayer.java

```
package blackjack.ui.pac;

import blackjack.core.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;

public abstract class VBettingPlayer extends BettingPlayer implements
➥Displayable {

    private BettingView view;

    public VBettingPlayer( String name, VHand hand, Bank bank ) {
        super( name, hand, bank );
    }

    public JComponent view() {
        if( view == null ) {
            view = new BettingView( (VHand)getHand() );
            addListener( view );
        }
        return view;
    }

    // Note que tudo que essa classe faz é recuperar o modo de visualização de
    Hand, adiciona esse modo
    // em si mesmo e atualiza a borda, conforme for necessário. Note o que essa
    classe não faz:
    // atualiza as cartas quando elas mudam. Do ponto de vista deste modo de
    visualização, a
    //atualização da carta acontece automaticamente, pois VHand atualiza sua
    exibição nos
    //bastidores

    private class BettingView extends JPanel implements PlayerListener {

        private TitledBorder border;

        public BettingView( VHand hand ) {
            super( new FlowLayout( FlowLayout.LEFT ) );
            buildGUI( hand.view() );
        }

        public void playerChanged( Player player ) {
            String name = VBettingPlayer.this.getName();
```

E

**LISTAGEM E.37** VBettingPlayer.java (*continuação*)

```
        border.setTitle( name );
        repaint();
    }

    public void playerBusted( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " BUSTED!" );
        repaint();
    }

    public void playerBlackjack( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " BLACKJACK!" );
        repaint();
    }

    public void playerStanding( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " STANDING " );
        repaint();
    }

    public void playerWon( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " WINNER!" );
        repaint();
    }

    public void playerLost( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " LOSER!" );
        repaint();
    }

    public void playerStandoff( Player player ) {
        String name = VBettingPlayer.this.getName();
        border.setTitle( name + " STANDOFF!" );
        repaint();
    }

    private void buildGUI( JComponent hand ){
        border = new TitledBorder( VBettingPlayer.this.getName() );
        setBorder( border );
        setBackground( new Color( 35, 142, 35 ) );
        border.setTitleColor( Color.black );
```

**LISTAGEM E.37** VBettingPlayer.java (*continuação*)

```
    add( hand );
}
}
```

**LISTAGEM E.38** VBlackjackDealer.java

```
package blackjack.ui.pac;

import blackjack.core.*;
import blackjack.core.threaded.*;
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;

public class VBlackjackDealer extends ThreadedBlackjackDealer implements
Displayable {
    private DealerView view;

    public VBlackjackDealer( String name, VHand hand, Deckpile cards ) {
        super( name, hand, cards );
    }

    public JComponent view() {
        if( view == null ) {
            view = new DealerView( (VHand)getHand() );
            addListener( view );
        }
        return view;
    }

    // Note que tudo que essa classe faz é recuperar o modo de visualização de
    Hand, adiciona esse modo
    // em si mesmo e atualiza a borda conforme for necessário .Note o que essa
    classe não faz:
    // atualiza as cartas quando elas mudam. Do ponto de vista desse modo de
    visualização,
    // a atualização da carta acontece automaticamente, pois VHand atualiza sua
    exibição
    // nos bastidores
    private class DealerView extends JPanel implements PlayerListener {

        private TitledBorder border;
```

E

**LISTAGEM E.38 VBlackjackDealer.java (continuação)**

```
public DealerView( VHand hand ) {
    super( new FlowLayout( FlowLayout.LEFT ) );
    String name = VBlackjackDealer.this.getName();
    border = new TitledBorder( name );
    setBorder( border );
    setBackground( new Color( 35, 142, 35 ) );
    border.setTitleColor( Color.black );
    add( hand.view() );
    repaint();
}

public void playerChanged( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name );
    repaint();
}

public void playerBusted( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " BUSTED!" );
    repaint();
}

public void playerBlackjack( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " BLACKJACK!" );
    repaint();
}

public void playerStanding( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " STANDING " );
    repaint();
}

public void playerWon( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " WINNER!" );
    repaint();
}

public void playerLost( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " LOSER!" );
    repaint();
}
```

**LISTAGEM E.38** VBlackjackDealer.java (*continuação*)

```
}

public void playerStandoff( Player player ) {
    String name = VBlackjackDealer.this.getName();
    border.setTitle( name + " STANDOFF!" );
    repaint();
}
}
```

**LISTAGEM E.39** VCard.java

```
package blackjack.ui.pac;

import blackjack.core.*;
import javax.swing.*;
import java.awt.*;

public class VCard extends Card implements Displayable {

    private String image;
    private CardView view;

    public Vcard( Suit suit, Rank rank, String image ) {
        super( suit, rank );
        this.image = image;
        view = new CardView( getImage() );
    }

    public void setFaceUp( boolean up ) {
        super.setFaceUp( up );
        view.changed();
    }

    public JComponent view() {
        return view;
    }

    private String getImage() {
        if( isFaceUp() ) {
            return image;
        } else {
            return "/blackjack/ui/bitmaps/empty_pile.xbm";
        }
    }
}
```

E

**LISTAGEM E.39** VCard.java (*continuação*)

```
    }
}

private class CardView extends JLabel {

    public CardView( String image ) {
        setImage( image );
        setBackground( Color.white );
        setOpaque( true );
    }

    public void changed(){
        setImage( getImage() );
    }

    private void setImage( String image ) {
        java.net.URL url = this.getClass().getResource( image );
        ImageIcon icon = new ImageIcon( url );
        setIcon( icon );
    }
}
}
```

---

**LISTAGEM E.40** VDeck.java

```
package blackjack.ui.pac;

import blackjack.core.*;
import java.util.Iterator;

public class VDeck extends Deck {

    protected void buildCards() {

        // Isso é horrível, mas é melhor do que os laços e as estruturas
        condicionais alternativas
        Card [] deck = new Card[52];
        setDeck( deck );

        deck [0] = new Vcard( Suit.HEARTS, Rank.TWO,
"/blackjack/ui/bitmaps/h2" );
        deck [1] = new VCard(Suit.HEARTS, Rank.THREE,
"/blackjack/ui/bitmaps/h3" );
        deck [2] = new Vcard( Suit.HEARTS, Rank.FOUR,
```

**LISTAGEM E.40** VDeck.java (*continuação*)

```
"/blackjack/ui/bitmaps/h4" );
    deck [3] = new Vcard( Suit.HEARTS, Rank.FIVE,
"/blackjack/ui/bitmaps/h5" );
    deck [4] = new Vcard( Suit.HEARTS, Rank.SIX,
"/blackjack/ui/bitmaps/h6" );
    deck [5] = new Vcard( Suit.HEARTS, Rank.SEVEN,
"/blackjack/ui/bitmaps/h7" );
    deck [6] = new Vcard( Suit.HEARTS, Rank.EIGHT,
"/blackjack/ui/bitmaps/h8" );
    deck [7] = new Vcard( Suit.HEARTS, Rank.NINE,
"/blackjack/ui/bitmaps/h9" );
    deck [8] = new Vcard( Suit.HEARTS, Rank.TEN,
"/blackjack/ui/bitmaps/h10" );
    deck [9] = new Vcard( Suit.HEARTS, Rank.JACK,
"/blackjack/ui/bitmaps/h11" );
    deck [10] = new Vcard( Suit.HEARTS, Rank.QUEEN,
"/blackjack/ui/bitmaps/h12" );
    deck [11] = new Vcard( Suit.HEARTS, Rank.KING,
"/blackjack/ui/bitmaps/h13" );
    deck [12] = new Vcard( Suit.HEARTS, Rank.ACE,
"/blackjack/ui/bitmaps/h1" );
    deck [13] = new Vcard( Suit.DIAMONDS, Rank.TWO,
"/blackjack/ui/bitmaps/d2" );
    deck [14] = new Vcard( Suit.DIAMONDS, Rank.THREE,
"/blackjack/ui/bitmaps/d3" );
    deck [15] = new Vcard( Suit.DIAMONDS, Rank.FOUR,
"/blackjack/ui/bitmaps/d4" );
    deck [16] = new Vcard( Suit.DIAMONDS, Rank.FIVE,
"/blackjack/ui/bitmaps/d5" );
    deck [17] = new Vcard( Suit.DIAMONDS, Rank.SIX,
"/blackjack/ui/bitmaps/d6" );
    deck [18] = new Vcard( Suit.DIAMONDS, Rank.SEVEN,
"/blackjack/ui/bitmaps/d7" );
    deck [19] = new Vcard( Suit.DIAMONDS, Rank.EIGHT,
"/blackjack/ui/bitmaps/d8" );
    deck [20] = new Vcard( Suit.DIAMONDS, Rank.NINE,
"/blackjack/ui/bitmaps/d9" );
    deck [21] = new Vcard( Suit.DIAMONDS, Rank.TEN,
"/blackjack/ui/bitmaps/d10" );
    deck [22] = new Vcard( Suit.DIAMONDS, Rank.JACK,
"/blackjack/ui/bitmaps/d11" );
    deck [23] = new Vcard( Suit.DIAMONDS, Rank.QUEEN,
"/blackjack/ui/bitmaps/d12" );
    deck [24] = new Vcard( Suit.DIAMONDS, Rank.KING,
"/blackjack/ui/bitmaps/d13" );
```

E

**LISTAGEM E.40 VDeck.java (continuação)**

```
deck [25] = new Vcard( Suit.DIAMONDS, Rank.ACE,
"/blackjack/ui/bitmaps/d1" );
deck [26] = new Vcard( Suit.SPADES, Rank.TWO,
"/blackjack/ui/bitmaps/s2" );
deck [27] = new Vcard( Suit.SPADES, Rank.THREE,
"/blackjack/ui/bitmaps/s3" );
deck [28] = new Vcard( Suit.SPADES, Rank.FOUR,
"/blackjack/ui/bitmaps/s4" );
deck [29] = new VCard(Suit.SPADES, Rank.FIVE,
"/blackjack/ui/bitmaps/s5" );
deck [30] = new Vcard( Suit.SPADES, Rank.SIX,
"/blackjack/ui/bitmaps/s6" );
deck [31] = new Vcard( Suit.SPADES, Rank.SEVEN,
"/blackjack/ui/bitmaps/s7" );
deck [32] = new Vcard( Suit.SPADES, Rank.EIGHT,
"/blackjack/ui/bitmaps/s8" );
deck [33] = new Vcard( Suit.SPADES, Rank.NINE,
"/blackjack/ui/bitmaps/s9" );
deck [34] = new Vcard( Suit.SPADES, Rank.TEN,
"/blackjack/ui/bitmaps/s10" );
deck [35] = new Vcard( Suit.SPADES, Rank.JACK,
"/blackjack/ui/bitmaps/s11" );
deck [36] = new Vcard( Suit.SPADES, Rank.QUEEN,
"/blackjack/ui/bitmaps/s12" );
deck [37] = new Vcard( Suit.SPADES, Rank.KING,
"/blackjack/ui/bitmaps/s13" );
deck [38] = new Vcard( Suit.SPADES, Rank.ACE,
"/blackjack/ui/bitmaps/s1" );
deck [39] = new Vcard( Suit.CLUBS, Rank.TWO,
"/blackjack/ui/bitmaps/c2" );
deck [40] = new Vcard( Suit.CLUBS, Rank.THREE,
"/blackjack/ui/bitmaps/c3" );
deck [41] = new Vcard( Suit.CLUBS, Rank.FOUR,
"/blackjack/ui/bitmaps/c4" );
deck [42] = new Vcard( Suit.CLUBS, Rank.FIVE,
"/blackjack/ui/bitmaps/c5" );
deck [43] = new Vcard( Suit.CLUBS, Rank.SIX,
"/blackjack/ui/bitmaps/c6" );
deck [44] = new Vcard( Suit.CLUBS, Rank.SEVEN,
"/blackjack/ui/bitmaps/c7" );
deck [45] = new Vcard( Suit.CLUBS, Rank.EIGHT,
"/blackjack/ui/bitmaps/c8" );
deck [46] = new Vcard( Suit.CLUBS, Rank.NINE,
"/blackjack/ui/bitmaps/c9" );
deck [47] = new Vcard( Suit.CLUBS, Rank.TEN,
```

**LISTAGEM E.40** VDeck.java (*continuação*)

```
"/blackjack/ui/bitmaps/c10" );
    deck [48] = new Vcard( Suit.CLUBS, Rank.JACK,
"/blackjack/ui/bitmaps/c11" );
    deck [49] = new Vcard( Suit.CLUBS, Rank.QUEEN,
"/blackjack/ui/bitmaps/c12" );
    deck [50] = new Vcard( Suit.CLUBS, Rank.KING,
"/blackjack/ui/bitmaps/c13" );
    deck [51] = new Vcard( Suit.CLUBS, Rank.ACE,
"/blackjack/ui/bitmaps/c1" );
}
}
```

**LISTAGEM E.41** VHand.java

```
package blackjack.ui.pac;

import blackjack.core.*;
import java.awt.*;
import javax.swing.*;
import java.util.Iterator;

public class VHand extends Hand implements Displayable {

    private HandView view = new HandView();

    public JComponent view() {
        return view;
    }

    // você precisa sobrepor addCard e reconfigurar para que quando a mão mudar, a
    // alteração se propague no modo de visualização
    public void addCard( Card card ) {
        super.addCard( card );
        view.changed();
    }

    public void reset() {
        super.reset();
        view.changed();
    }

    private class HandView extends JPanel {
        public HandView() {
            super( new FlowLayout( FlowLayout.LEFT ) );
        }
    }
}
```

E

**LISTAGEM E.41** VHand.java (*continuação*)

```
        setBackground( new Color( 35, 142, 35 ) );
    }

    public void changed() {
        removeAll();
        Iterator i = getCards();
        while( i.hasNext() ) {
            VCard card = ( Vcard ) i.next();
            add( card.view() );
        }
        revalidate();
    }
}
```

---

**LISTAGEM E.42** VPlayerFactory.java

```
package blackjack.ui.pac;

import blackjack.core.*;

public class VPlayerFactory {

    private VBlackjackDealer dealer;
    private GUIPlayer human;
    private Deckpile pile;

    public VBlackjackDealer getDealer() {
        // cria e retorna apenas um
        if( dealer == null ) {
            VHand dealer_hand = getHand();
            Deckpile cards = getCards();
            dealer = new VBlackjackDealer( "Dealer", dealer_hand, cards );
        }
        return dealer;
    }

    public GUIPlayer getHuman() {
        // cria e retorna apenas um
        if( human == null ) {
            VHand human_hand = getHand();
            Bank bank = new Bank( 1000 );
            human = new GUIPlayer( "Human", human_hand, bank, getDealer() );
        }
    }
}
```

**LISTAGEM E.42** VPlayerFactory.java (*continuação*)

```
    return human;
}

public Deckpile getCards() {
    // cria e retorna apenas um
    if( pile == null ) {
        pile = new Deckpile();
        for( int i = 0; i < 4 ; i ++ ) {
            pile.shuffle();
            Deck deck = new VDeck();
            deck.addToStack( pile );
            pile.shuffle();
        }
    }
    return pile;
}

private VHand getHand() {
    return new VHand();
}
}
```

PÁGINA EM BRANCO

# Índice Remissivo

## Símbolos

\* (asterisco), agregações de objeto, 186

## A

### Abstração, 26-27

- exemplo de jogo de cartas, 56
- exemplo de, 26-29
- implementação eficiente de, 41-42
- regras para, 28-29
- situações apropriadas para, 29

### Abstração, alternativas, 93

Abstract Data Type. *Veja ADT*

### Acesso

- privado, 25
- protegido, 25
- público, 25
- níveis de, 25

### Assessores, 11

- classes, 575

### ACM Special Interest Group on Computer-Human Interaction (SIGCHI), 290

### Adaptadores, 242

- de classe, 244-245
- de objeto, 244-245

### Agentes, objetos, 231

### Agregações, UML, 585

### Ambiente

- configuração, SDK, 555-556
- variáveis, PATH, 556

### Analisadores

- documentos XML, padrão de projeto
- Abstract Factory e, 266
- exemplo de código, 266-267

### Análise

- recursos, 587

*Veja também AOO*

### Análise de caso de uso, 199

- casos de uso resultantes, 204-205
- combinando casos de uso, 203-204
- definindo a seqüência de eventos, 205-207
- diagramas, 207
  - de atividade, 212
  - de colaboração, 211
  - de interação, 209
  - de seqüência, 210-211
- dicas para escrever, 207
- dividindo casos de uso, 203-204
- entrada do usuário e, 200
- identificando atores, 200-201
- jogo de cartas vinte-e-um
  - aposta, 414-417
  - GUI, 433-435
  - regras, 384-388
- listando casos de uso, 201-203
- lógica booleana da, 476

### Ancestrais, 88

### Aninhando instruções if/else, 337

### AOO (Análise Orientada a Objeto)

- modelo de caso de uso, 199
- casos de uso resultantes, 204
- combinando casos, 203-204
- definindo seqüência de eventos, 205-207
- diagramas de atividade, 212
- diagramas de caso de uso, 207-208
- diagramas de colaboração, 211
- diagramas de interação, 209-210
- diagramas de seqüência, 210-211
- dividindo a lista de casos, 203
- dividindo casos, 203
- gerando lista preliminar, 201-203
- identificando atores, 200-201

- modelo de domínio, diagramas de atividade, 212  
objetivo, 352  
panorama, 197-198  
protótipos, 214  
sistema, 198
- APIs (interfaces de programa aplicativo), 24**
- Argumentos**  
conversão, 137  
ferramentas de terceiros, adaptadores, 244-245  
*Veja também* parâmetros, 133-134
- Arquivo**  
Product.java, 573-574  
SimpleHelloWorld.java, 564
- Arquivos, criando arquivos .jar, 562-563**
- Associações (objetos), UML, 584**
- Asteriscos (\*), associações de objeto, 185**
- Atividades, diagrama de estados, 390-391**
- Atores, 200**  
análise de caso de uso, 352-353  
jogo de cartas vinte-e-um, 353  
modelando, 207
- Atributos, 8**  
especialização, 87-89  
herança, 77-78  
recursivos, 84-85  
sobrepostos, 79-83  
controle de acesso, 83  
visibilidade, UML, 581-582
- Atualizando**  
documentação, 339  
interfaces, 35  
projeto de UI desacoplada e, 290-291  
bibliotecas de terceiros, padrão de projeto  
Abstract Factory, 264
- Autotransições, diagramas de estado, 390-391**
- B**
- Bibliotecas**  
ferramentas de terceiros
- adaptadores, 244-245  
atualizando, 264  
Swing, 293
- Blackjack.exe, interfaces com o usuário, 621-627**
- Blackjack.java, 376-377**
- BlackjackDealer.java, 374-375**
- C**
- C++, recursos, 588**
- Cabeçalhos (métodos e classes), documentação, 338-339**
- Camada controladora (MVC), 301**  
combinando com camada de modelo, 302-303  
implementando, 301-302
- Camada de modelo (MVC), 294**  
implementando, 295-296  
reunindo com camada de controle, 302-303
- Camada de modo de visualização (MVC), 297**
- Capacidade de conexão, 91-92**  
implementando, 297-300
- Cartões CRC (Class Responsibility Collaboration), 224-225**  
aplicando, 225-226  
exemplo, 226-229  
GUIs, jogo de cartas vinte-e-um, 437  
limitações, 230-231
- Casos de teste, 316**  
nomeando, 322  
teste de caixa branca, 316  
teste de caixa preta, 316
- Casos de uso**  
aposta, 414-416  
atores, 352  
jogo de cartas vinte-e-um, 353, 356-360  
Banca efetua jogo, 387-388  
Deal Cards, 384-387

- Cenários (análise de caso de uso),** 205  
**Chamadas de método,** 8  
**Chaves ({} ) no código-fonte,** 564  
**Chaves, especificando,** 51  
**Classe Bank (jogo de cartas vinte-e-um)**  
    exemplo de código, 159  
    implementação, 421-422  
    projetando, 417  
**Classe**  
    BettingPlayer, implementação, 423-424  
    BlackjackDealer, implementação, 426-427  
    BlackjackGUI, método setUp(), 448-449  
    CardView, implementando, 443  
    CountedObject, exemplo de código, 65  
    Dealer  
    DoubleKey  
        construtores, 52  
        exemplo de código, 50-51  
        exemplo de código, 61  
    FlipPlayer, implementando, 475-476  
    GUIPlayer, implementando, 445-448, 465  
    Hand, receptor para, 395-399  
    HumanPlayer, implementação, 424-426  
    OneHitPlayer, implementando, 476  
    OptionView, implementando, 445  
    OptionViewController, implementando, 445  
    Player  
        herança, refazendo a hierarquia, 419-420  
        interface PlayerListener, 403-404  
        interface PlayerState, 399-403  
    PlayerView, implementando, 444-445  
    progenitora, 77-79  
        exemplo de código de herança, 99-100  
        palavra-chave super, 84  
    SafePlayer  
        adicionando na GUI, 472-473  
        criando, 471-472  
    SmartPlayer, implementando, 477  
    Stack, herança  
        exemplo de código da solução, 118-119  
        exemplo de problema, 117-118  
        exposição do problema, 118  
    State, jogo de cartas vinte-e-um, 393  
    VBettingPlayer, implementando, 462-463  
    VBlackjackDealer, implementando, 464-465  
    VCard, implementando, 441-442, 460-461  
    VDeck, implementando, 442  
    VHand, implementando, 461-462  
**Classes abstratas**  
    exemplo de código, 102-103, 126-128  
    métodos, definindo, 104  
    requisitos do exemplo, 105  
    UML, 582  
        notação, 182  
**Classes anônimas, criando,** 324  
**Classes filhas,** 77-78  
    especialização, 87-88  
    exemplo de código de herança, 100-101  
    novos métodos, 84  
    palavra-chave super, 84  
    polimorfismo, exemplo de código, 147-148  
    requisitos do exemplo, 99-101  
**Classes folha,** 89  
**Classes internas,** 577-579  
    anônimas, 577-579  
**Classes-raiz,** 89  
    anônimas, 577-579  
**Classes,** 8  
    abstratas  
        exemplo de código, 102-103  
        notação UML, 181-182  
        requisitos do exemplo, 105-106  
    ADT (Abstract Data Type), TAD, implementação eficiente de, 47  
    agregações, UML, 585  
    ancestral, 88  
    anônimas, 577-579  
        criando, 324-325  
    assessores, 575  
    associações de objeto, UML, 584  
    atributos, 8  
    Bank, 416-417  
        implementação, 421  
    BaseLog, exemplo de código, 127-128  
    BettingPlayer, implementação, 423-424

- Blackjack  
diagrama de atualização, 420-421  
método `setUp()`, 448-449
- BlackjackDealer, implementação, 426-427  
cabeçalhos, documentação e, 338-339
- CardView, implementando, 443
- cartões CRC (Class Responsibility Collaboration), 224-225  
aplicando, 225-226  
exemplo, 226-227  
limitações, 230-231
- compilando, 561-562
- composição, 76  
    UML, 585
- comuns  
    pacote blackjack.core, 592-619, 618-619
- considerações de projeto, testes de unidade e, 321-322
- convenções de atribuição de nomes, 338
- CountedObject, exemplo de código, 65
- criando, estado de jogo e, 393, 395
- Dealer, implementação, 424-425
- diagramas  
    de colaboração, UML, 586  
    de seqüência, UML, 586
- DoubleKey, construtores, 52
- exemplo de código, 9-10  
    Bank, 159-160  
    BankAccount, 111-112  
    CheckingAccount, 114-115  
    OverdraftAccount, 116  
    SavingsAccount, 112-113  
    TimeMaturityAccount, 113-114
- exemplo de conta  
    código para, 55  
    requisitos, 5343
- exemplo de jogo de cartas, código para, 57-62
- exemplos de código, DoubleKey.java, 50
- filhas, 77
- especialização, 87
- exemplo de código de herança, 100-101
- novos métodos, 84
- palavra-chave super, 84
- polimorfismo (exemplo de código), 147-148
- requisitos do exemplo, 100
- FlipPlayer, implementando, 475
- folha, 89
- GUIPlayer, implementando, 445-448, 465
- Hand, receptor para, 395-398
- Hashtable, conversão e, 157
- herança, princípios básicos, 77-78
- hierarquia de herança, 77
- HumanPlayer, implementação, 424-425
- instanciação, evitando múltipla, 268
- internas, 577-579
- Java, 564-565, 571-572  
    criando, 572-573
- jogo de cartas vinte-e-um, 360-364
- criando jogadores não-humanos, 471-472
- implementação, 365-380
- modelo, 365
- métodos  
    construtores, 574-575  
    de configuração, 575  
    de obtenção, 575
- modelando, 188
- documentando código, 179
- notação de seleção UML, 181
- notação UML avançada, 181
- notação UML, 179-180
- relacionamentos de agregação, 186-187
- relacionamentos de associação, 184-185
- relacionamentos de composição, 187-188
- relacionamentos de dependência, 183-184
- relacionamentos de generalização, 188-189
- relacionamentos, 183
- modificadores, 575
- multiplicidade, UML, 584-585
- número de responsabilidades (problemas de projeto), 229-230
- objetos, 7-8
- OneHitPlayer, implementando, 476
- OptionView, implementando, 445