



Modelo do elevador (no CD)

H.1 Introdução

Depois de ter lido as seções “Pensando em objetos”, você deve ter uma compreensão considerável do processo de projeto e dos diagramas de UML que dizem respeito à nossa simulação. Este apêndice apresenta o código de todas as 10 classes que, em conjunto, representam o modelo e terminam a discussão de seu funcionamento. Discutimos cada classe separadamente e em detalhes.

H.2 A classe `ElevatorModel`

Como discutido na Seção 13.17, a classe `ElevatorModel` (Fig. H.1) reúne os objetos que formam o modelo da simulação do elevador. O `ElevatorModel` serve para enviar eventos do modelo para a visão. O `ElevatorModel` também instancia novas `Persons` e permite que cada `Floor` obtenha uma referência para o `ElevatorShaft`.

```
1 // ElevatorModel.java
2 // Modelo da simulação do Elevator com ElevatorShaft e dois Floors
3 package com.deitel.jhtp4.elevator.model;
4
5 // Pacotes do núcleo de Java
6 import java.util.*;
7
8 // Pacotes Deitel
9 import com.deitel.jhtp4.elevator.event.*;
10 import com.deitel.jhtp4.elevator.ElevatorConstants;
11
12 public class ElevatorModel implements ElevatorModelListener,
13     ElevatorConstants {
14
15     // declara arquitetura de dois Floors na simulação
16     private Floor firstFloor;
17     private Floor secondFloor;
18
19     // ElevatorShaft na simulação
20     private ElevatorShaft elevatorShaft;
```

Fig. H.1 Classe `ElevatorModel` representa o modelo em nossa simulação de elevador (parte 1 de 6).

```

21
22 // objetos que esperam por eventos do ElevatorModel
23 private Set personMoveListeners;
24 private DoorListener doorListener;
25 private ButtonListener buttonListener;
26 private LightListener lightListener;
27 private BellListener bellListener;
28 private ElevatorMoveListener elevatorMoveListener;
29
30 // número cumulativo de pessoas na simulação
31 private int numberOfPeople = 0;
32
33 // construtor instancia ElevatorShaft e Floors
34 public ElevatorModel()
35 {
36     // instancia objetos firstFloor e secondFloor
37     firstFloor = new Floor( FIRST_FLOOR_NAME );
38     secondFloor = new Floor( SECOND_FLOOR_NAME );
39
40     // instancia objeto ElevatorShaft
41     elevatorShaft =
42         new ElevatorShaft( firstFloor, secondFloor );
43
44     // dá referência a elevatorShaft para primeiro e segundo Floors
45     firstFloor.setElevatorShaft( elevatorShaft );
46     secondFloor.setElevatorShaft( elevatorShaft );
47
48     // registra-se para eventos de ElevatorShaft
49     elevatorShaft.setDoorListener( this );
50     elevatorShaft.setButtonListener( this );
51     elevatorShaft.addElevatorMoveListener( this );
52     elevatorShaft.setLightListener( this );
53     elevatorShaft.setBellListener( this );
54
55     // instancia Set para objetos ElevatorMoveListener
56     personMoveListeners = new HashSet( 1 );
57 } // fim do construtor ElevatorModel
58
59 // devolve Floor com o nome dado
60 private Floor getFloor( String name )
61 {
62     if ( name.equals( FIRST_FLOOR_NAME ) )
63         return firstFloor;
64     else
65
66         if ( name.equals( SECOND_FLOOR_NAME ) )
67             return secondFloor;
68         else
69             return null;
70 } // fim do método getFloor
71
72 // adiciona Person a ElevatorSimulator
73 public void addPerson( String floorName )
74 {
75     // instancia nova Person e a coloca no Floor
76     Person person =
77         new Person( numberOfPeople, getFloor( floorName ) );
78
79

```

Fig. H.1 Classe `ElevatorModel` representa o modelo em nossa simulação de elevador (parte 2 de 6).

```

80     person.setName( Integer.toString( numberOfPeople ) );
81
82     // registra ouvinte (listener) para eventos de Person
83     person.setPersonMoveListener( this );
84
85     // dispara a thread de Person
86     person.start();
87
88     // incrementa o número de objetos Person na simulação
89     numberOfPeople++;
90
91 } // fim do método addPerson
92
93 // invocado quando o Elevator partiu do Floor
94 public void elevatorDeparted(
95     ElevatorMoveEvent moveEvent )
96 {
97     elevatorMoveListener.elevatorDeparted( moveEvent );
98 }
99
100 // invocado quando o Elevator chegou no Floor de destino
101 public void elevatorArrived(
102     ElevatorMoveEvent moveEvent )
103 {
104     elevatorMoveListener.elevatorArrived( moveEvent );
105 }
106
107 // envia PersonMoveEvent para ouvinte, dependendo do tipo de evento
108 private void sendPersonMoveEvent(
109     int eventType, PersonMoveEvent event )
110 {
111     Iterator iterator = personMoveListeners.iterator();
112
113     while ( iterator.hasNext() ) {
114
115         PersonMoveListener listener =
116             ( PersonMoveListener ) iterator.next();
117
118         // envia Event para ouvinte "this", dependendo do eventType
119         switch ( eventType ) {
120
121             // Person foi criada
122             case Person.PERSON_CREATED:
123                 listener.personCreated( event );
124                 break;
125
126             // Person chegou no Elevator
127             case Person.PERSON_ARRIVED:
128                 listener.personArrived( event );
129                 break;
130
131             // Person entrou no Elevator
132             case Person.PERSON_ENTERING_ELEVATOR:
133                 listener.personEntered( event );
134                 break;
135
136             // Person pressionou o objeto Button
137             case Person.PERSON_PRESSING_BUTTON:

```

Fig. H.1 Classe `ElevatorModel` representa o modelo em nossa simulação de elevador (parte 3 de 6).

```

138         listener.personPressedButton( event );
139         break;
140
141         // Person saiu do Elevator
142         case Person.PERSON_EXITING_ELEVATOR:
143             listener.personDeparted( event );
144             break;
145
146         // Person saiu da simulação
147         case Person.PERSON_EXITED:
148             listener.personExited( event );
149             break;
150
151         default:
152             break;
153     }
154 }
155 } // fim do método sendPersonMoveEvent
156
157 // invocado quando a Person foi criada no modelo
158 public void personCreated( PersonMoveEvent moveEvent )
159 {
160     sendPersonMoveEvent( Person.PERSON_CREATED, moveEvent );
161 }
162
163 // invocado quando a Person chegou no Button do Floor
164 public void personArrived( PersonMoveEvent moveEvent )
165 {
166     sendPersonMoveEvent( Person.PERSON_ARRIVED, moveEvent );
167 }
168
169 // invocado quando a Person pressionou o Button
170 public void personPressedButton( PersonMoveEvent moveEvent )
171 {
172     sendPersonMoveEvent( Person.PERSON_PRESSING_BUTTON,
173         moveEvent );
174 }
175
176 // invocado quando a Person entrou no Elevator
177 public void personEntered( PersonMoveEvent moveEvent )
178 {
179     sendPersonMoveEvent( Person.PERSON_ENTERING_ELEVATOR,
180         moveEvent );
181 }
182
183 // invocado quando a Person saiu do Elevator
184 public void personDeparted( PersonMoveEvent moveEvent )
185 {
186     sendPersonMoveEvent( Person.PERSON_EXITING_ELEVATOR,
187         moveEvent );
188 }
189
190 // invocado quando a Person saiu da simulação
191 public void personExited( PersonMoveEvent moveEvent )
192 {
193     sendPersonMoveEvent( Person.PERSON_EXITED, moveEvent );
194 }
195
196 // invocado quando a Door abriu

```

Fig. H.1 Classe `ElevatorModel` representa o modelo em nossa simulação de elevador (parte 4 de 6).

```

197     public void doorOpened( DoorEvent doorEvent )
198     {
199         doorListener.doorOpened( doorEvent );
200     }
201
202     // invocado quando a Door fechou
203     public void doorClosed( DoorEvent doorEvent )
204     {
205         doorListener.doorClosed( doorEvent );
206     }
207
208     // invocado quando o Button foi pressionado
209     public void buttonPressed( ButtonEvent buttonEvent )
210     {
211         buttonListener.buttonPressed( buttonEvent );
212     }
213
214     // invocado quando o Button foi desligado
215     public void buttonReset( ButtonEvent buttonEvent )
216     {
217         buttonListener.buttonReset( buttonEvent );
218     }
219
220     // invocado quando a Bell tocou
221     public void bellRang( BellEvent bellEvent )
222     {
223         bellListener.bellRang( bellEvent );
224     }
225
226     // invocado quando a Light acendeu
227     public void lightTurnedOn( LightEvent lightEvent )
228     {
229         lightListener.lightTurnedOn( lightEvent );
230     }
231
232     // invocado quando a Light apagou
233     public void lightTurnedOff( LightEvent lightEvent )
234     {
235         lightListener.lightTurnedOff( lightEvent );
236     }
237
238     // configura ouvinte (listener) para ElevatorModelListener
239     public void setElevatorModelListener(
240         ElevatorModelListener listener )
241     {
242         // ElevatorModelListener estende todas as interfaces abaixo
243         addPersonMoveListener( listener );
244         setElevatorMoveListener( listener );
245         setDoorListener( listener );
246         setButtonListener( listener );
247         setLightListener( listener );
248         setBellListener( listener );
249     }
250
251     // configura ouvinte (listener) para PersonMoveEvents
252     public void addPersonMoveListener(
253         PersonMoveListener listener )
254     {
255         personMoveListeners.add( listener );
256     }

```

Fig. H.1 Classe `ElevatorModel` representa o modelo em nossa simulação de elevador (parte 5 de 6).

```

257
258 // configura ouvinte para DoorEvents
259 public void setDoorListener( DoorListener listener )
260 {
261     doorListener = listener;
262 }
263
264 // configura ouvinte para ButtonEvents
265 public void setButtonListener( ButtonListener listener )
266 {
267     buttonListener = listener;
268 }
269
270 // adiciona ouvinte para ElevatorMoveEvents
271 public void setElevatorMoveListener(
272     ElevatorMoveListener listener )
273 {
274     elevatorMoveListener = listener;
275 }
276
277 // configura ouvinte para LightEvents
278 public void setLightListener( LightListener listener )
279 {
280     lightListener = listener;
281 }
282
283 // configura ouvinte para BellEvents
284 public void setBellListener( BellListener listener )
285 {
286     bellListener = listener;
287 }
288 }

```

Fig. H.1 Classe **ElevatorModel** representa o modelo em nossa simulação de elevador (parte 6 de 6).

O diagrama de classes da Fig.15.21 mostra que a classe **ElevatorModel** contém uma instância da classe **ElevatorShaft** e duas instâncias da classe **Floor**, de modo que **ElevatorModel** declara o objeto **elevatorShaft** (linha 20) e os objetos **firstFloor** e **secondFloor** (linhas 16 e 17). As linhas 37 a 46 instanciam estes objetos e dão a cada objeto **Floor** uma referência para o objeto **ElevatorShaft**. A Fig. 15.21 também mostra que a classe **ElevatorModel** cria objetos **Person**. De acordo com a Fig. 15.21, a classe **ElevatorModel** contém o método **addPerson** (linhas 75 a 91), que cria e coloca uma **Person** no **Floor** especificado. A linha 86 do método **addPerson** dispara a *thread* da **Person** e a linha 89 incrementa o número acumulado de objetos **Person** na simulação.

Como foi mencionado antes, a classe **ElevatorModel** envia eventos a partir do modelo para a visão. A declaração de classe (linhas 12 e 13) e as linhas 49 a 53 revelam que o **ElevatorModel** espera diversos tipos de eventos do **ElevatorShaft** – é assim que o **ElevatorModel** recebe eventos dos objetos que compõem o modelo. Especificamente, a classe **ElevatorModel** implementa a interface **ElevatorModelListener**, que implementa todas as interfaces do pacote **event**. As linhas 23 a 28 declaram os objetos *listener* para os quais o **ElevatorModel** envia os eventos que ele recebe do **ElevatorShaft**. Como foi mencionado na Seção 13.17, o **ElevatorFrame** (o aplicativo) registra a **ElevatorView** como um ouvinte (*listener*) para eventos do **ElevatorModel** – é assim que o **ElevatorModel** envia eventos do modelo para a visão.

As linhas 94 a 236 da classe **ElevatorModel** implementam todos os métodos da interface **ElevatorModelListener** e as linhas 239 a 288 fornecem os métodos **addListener** para registrar um ouvinte (neste caso, o ouvinte é a **ElevatorView**) para todos os eventos. Na verdade, dois terços da classe se dedicam a “borbulhar” mensagens do modelo para a visão.

Apresentamos um diagrama de classes que mostrou as realizações do modelo do elevador na Fig. 11.27. Alteramos este diagrama para se adaptar ao fato de que a classe **ElevatorModel** implementa todas as interfaces através da interface **ElevatorModelListener**. Além disso, a classe **ElevatorShaft** precisa implementar mais interfaces para receber eventos da classe **Elevator**, de modo que **ElevatorShaft** possa enviar estes eventos para o **ElevatorModel**. Apresentamos o diagrama de classes mostrando todas as realizações para o modelo na Fig. H.2 e na Fig. H.3 – a Fig. H.2 mostra o relacionamento entre as interfaces *listener* e a interface **ElevatorModelListener** (poderíamos ter criado um diagrama de classes mostrando os dois tipos de relacionamento, mas o diagrama teria ficado muito confuso).

A Fig. H.4 descreve o conteúdo do diagrama de classes que mostra as realizações do modelo do elevador. As mudanças significativas são que a classe **ElevatorShaft** agora implementa a interface **ElevatorMoveListener**, e **ElevatorModel** implementa a interface **ElevatorModelListener**, que implementa todas as interfaces.

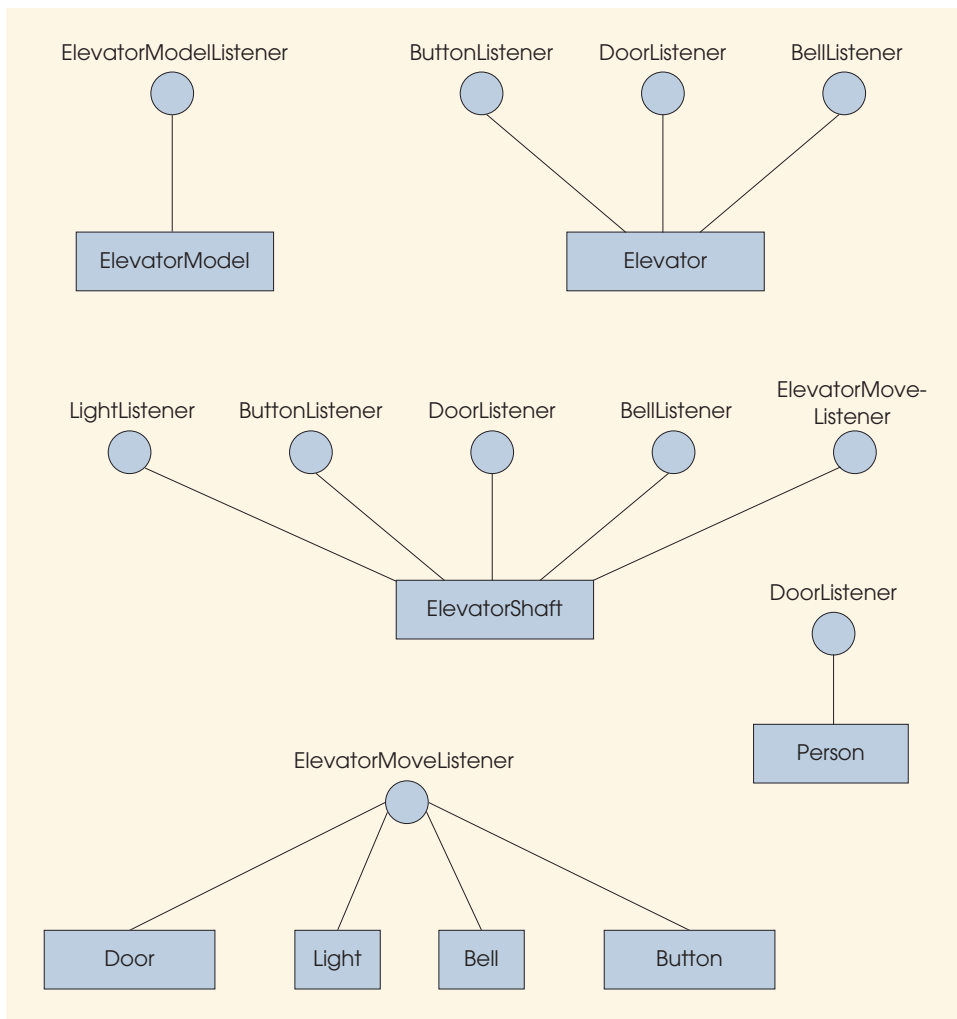


Fig. H.2 Diagramas de classes mostrando as realizações no modelo do elevador (parte 1 de 2).

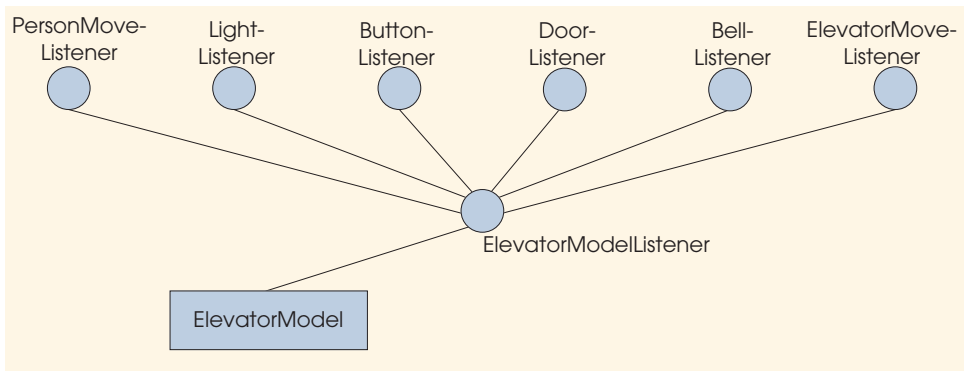


Fig. H.3 Diagramas de classes mostrando as realizações no modelo do elevador (parte 2 de 2).

Classe	implementa <i>Listener</i>
ElevatorModel	ElevatorModelListener
ElevatorModelListener	PersonMoveListener
	ElevatorMoveListener
	ButtonListener
	DoorListener
	BellListener
	LightListener
Door, Light, Bell, Button	ElevatorMoveListener
Elevator	ButtonListener
	DoorListener
	BellListener
ElevatorShaft	LightListener
	ButtonListener
	DoorListener
	BellListener
	ElevatorMoveListener
Person	DoorListener

Fig. H.4 Classes e interfaces *listener* implementadas da Fig. H.2.

H.3 As classes Location e Floor

Precisamos representar a posição da **Person** na simulação. A **Person** poderia ter um atributo **int** descrevendo em que **Floor** a **Person** está caminhando; entretanto, a **Person** não está em nenhum dos **Floors** quando está andando no **Elevator**. Como descrito na Seção 9.23 de “Pensando em objetos”, nossa solução é que a **Person** mantenha uma referência **Location**, que faz referência para um **Floor** ou para o **Elevator**, dependendo de onde está a **Person**. Para implementar esta característica, as classes **Floor** e **Elevator** estendem a superclasse abstrata **Location** (Fig. H.5).

```

1  // Location.java
2  // Superclasse abstrata que representa um lugar na simulação
3  package com.deitel.jhttp4.elevator.model;
4
5  // Pacotes Deitel
6  import com.deitel.jhttp4.elevator.event.*;
7
8  public abstract class Location {
9
10     // nome da Location
11     private String locationName;
12
13     // configura nome da Location
14     protected void setLocationName( String name )
15     {
16         locationName = name;
17     }
18
19     // devolve nome da Location
20     public String getLocationName()
21     {
22         return locationName;
23     }
24
25     // devolve Button na Location
26     public abstract Button getButton();
27
28     // devolve objeto Door na Location
29     public abstract Door getDoor();
30 }

```

Fig. H.5 Superclasse abstrata `Location` que representa um lugar na simulação.

`Location` contém o `String locationName` (linha 11), que pode conter os valores `firstFloor`, `secondFloor` ou `elevator` para descrever os três lugares que `Person` pode ocupar. As linhas 26 e 29 declaram os métodos abstratos `getButton` e `getDoor`, respectivamente. Usando estes métodos, o `Floor` devolve referências para objetos associados com aquele `Floor` e o `Elevator` devolve referências para os objetos associados com o `Elevator`. A referência `Location` permite que uma `Person` pressione um `Button` e saiba quando uma `Door` se abriu. Por exemplo, se desejamos que uma `Person` pressione um `Button`, escrevemos

```
person.getLocation().getButton().pressButton();
```

Portanto, nosso uso de uma superclasse abstrata fornece uma maneira alternativa para os objetos em nosso modelo interagirem. A Fig. H.6 apresenta a classe `Floor`, uma subclasse da classe `Location`. O construtor `Floor` (linhas 15 a 18) recebe como o argumento `String` o valor `firstFloor` ou `secondFloor` para identificar o `Floor`. A linha 17 invoca o método `setLocationName` para atribuir o valor deste `String` ao atributo `locationName`, herdado da superclasse `Location`.

```

1  // Floor.java
2  // Representa um Floor localizado próximo a um ElevatorShaft
3  package com.deitel.jhttp4.elevator.model;
4
5  // Pacotes Deitel
6  import com.deitel.jhttp4.elevator.ElevatorConstants;
7
8  public class Floor extends Location
9  {
10     implements ElevatorConstants {

```

Fig. H.6 A classe `Floor` – uma subclasse de `Location` – representa um `Floor` através do qual a `Person` caminha para o `Elevator` (parte 1 de 2).

```

10
11 // referência para objeto ElevatorShaft
12 private ElevatorShaft elevatorShaft;
13
14 // construtor Floor configura nome do Floor
15 public Floor( String name )
16 {
17     setLocationName( name );
18 }
19
20 // obtém Button do primeiro ou segundo Floor, usando nome da Location
21 public Button getButton()
22 {
23     if ( getLocationName().equals( FIRST_FLOOR_NAME ) )
24         return getElevatorShaft().getFirstFloorButton();
25     else
26
27         if ( getLocationName().equals( SECOND_FLOOR_NAME ) )
28             return getElevatorShaft().getSecondFloorButton();
29     else
30
31         return null;
32 } // fim do método getButton
33
34 // obtém Door do primeiro ou segundo Floor, usando nome da Location
35 public Door getDoor()
36 {
37     if ( getLocationName().equals( FIRST_FLOOR_NAME ) )
38         return getElevatorShaft().getFirstFloorDoor();
39     else
40
41         if ( getLocationName().equals( SECOND_FLOOR_NAME ) )
42             return getElevatorShaft().getSecondFloorDoor();
43     else
44
45         return null;
46 } // fim do método getDoor
47
48 // obtém referência para ElevatorShaft
49 public ElevatorShaft getElevatorShaft()
50 {
51     return elevatorShaft;
52 }
53
54 // configura referência para ElevatorShaft
55 public void setElevatorShaft( ElevatorShaft shaft )
56 {
57     elevatorShaft = shaft;
58 }
59
60 }
61

```

Fig. H.6 A classe **Floor** – uma subclasse de **Location** – representa um **Floor** através do qual a **Person** caminha para o **Elevator** (parte 2 de 2).

A classe **Floor** fornece os métodos concretos **getButton** (linhas 21 a 33) e **getDoor** (linhas 36 a 48). Os métodos **getButton** e **getDoor** devolvem uma referência para **Button** e para **Door** no primeiro ou segundo **Floor**, dependendo de qual **Floor** devolve a referência. Finalmente, o método **getElevatorShaft** (linhas 51 a 54) devolve uma referência para o **ElevatorShaft**. Veremos mais tarde como uma **Person** usa este método, combinado com a referência **Location** daquela **Person**, para entrar no **Elevator**.

H.4 A classe Door

As **Doors** são uma parte essencial do modelo, porque elas informam a uma **Person** quando entrar no **Elevador** ou sair dele – sem as **Doors**, nenhuma **Person** andaria no **Elevador**. Os diagramas de colaborações das Figs. 7.20, 10.25 e 15.18 apresentaram as colaborações entre as **Doors**, o **Elevador** e a **Person**. Agora, fornecemos um *walkthrough* da classe **Door**.

```

1  // Door.java
2  // Envia DoorEvents para DoorListeners quando aberta ou fechada
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes do núcleo de Java
6  import java.util.*;
7
8  // Pacotes Deitel
9  import com.deitel.jhtp4.elevator.event.*;
10
11 public class Door implements ElevatorMoveListener {
12
13     // representa se a Door está aberta ou fechada
14     private boolean open = false;
15
16     // tempo antes de a Door fechar automaticamente
17     public static final int AUTOMATIC_CLOSE_DELAY = 3000;
18
19     // Set de DoorListeners
20     private Set doorListeners;
21
22     // lugar onde a Door abriu ou fechou
23     private Location doorLocation;
24
25     // construtor Door instancia Set para DoorListeners
26     public Door()
27     {
28         doorListeners = new HashSet( 1 );
29     }
30
31     // adiciona doorListener
32     public void addDoorListener( DoorListener listener )
33     {
34         // impede que outros objetos modifiquem doorListeners
35         synchronized( doorListeners )
36         {
37             doorListeners.add( listener );
38         }
39     }
40
41     // remove doorListener
42     public void removeDoorListener( DoorListener listener )
43     {
44         // impede que outros objetos modifiquem doorListeners
45         synchronized( doorListeners )
46         {
47             doorListeners.remove( listener );
48         }
49     }
50
51     // abre Door e envia objetos DoorEvent para todos os ouvintes (listener)

```

Fig. H.7 A classe **Door**, que representa uma **Door** no modelo, informa aos ouvintes quando uma **Door** abriu ou fechou (parte 1 de 3).

```

52     public void openDoor( Location location )
53     {
54         if ( !open ) {
55
56             open = true;
57
58             // obtém iterador do Set
59             Iterator iterator;
60             synchronized( doorListeners )
61             {
62                 iterator = new HashSet( doorListeners ).iterator();
63             }
64
65             // obtém próximo DoorListener
66             while ( iterator.hasNext() ) {
67                 DoorListener doorListener =
68                     ( DoorListener ) iterator.next();
69
70                 // envia evento doorOpened para DoorListener "this"
71                 doorListener.doorOpened(
72                     new DoorEvent( this, location ) );
73             }
74
75             doorLocation = location;
76
77             // declara Thread que assegura o fechamento automático da Door
78             Thread closeThread = new Thread(
79                 new Runnable() {
80
81                     public void run()
82                     {
83                         // fecha Door se aberta por mais de três segundos
84                         try {
85                             Thread.sleep( AUTOMATIC_CLOSE_DELAY );
86                             closeDoor( doorLocation );
87                         }
88
89                         // trata exceção se interrompida
90                         catch ( InterruptedException exception ) {
91                             exception.printStackTrace();
92                         }
93                     }
94                 } // fim da classe interna anônima
95             );
96
97             closeThread.start();
98         }
99     } // fim do método openDoor
100
101     // fecha Door e envia objetos DoorEvent para todos os ouvintes
102     public void closeDoor( Location location )
103     {
104         if ( open ) {
105
106             open = false;
107
108             // obtém iterador do Set
109             Iterator iterator;
110             synchronized( doorListeners )

```

Fig. H.7 A classe `Door`, que representa uma `Door` no modelo, informa aos ouvintes quando uma `Door` abriu ou fechou (parte 2 de 3).

```

111         {
112             iterator = new HashSet( doorListeners ).iterator();
113         }
114
115         // obtém próximo DoorListener
116         while ( iterator.hasNext() ) {
117             DoorListener doorListener =
118                 ( DoorListener ) iterator.next();
119
120             // envia evento doorClosed para DoorListener "this"
121             doorListener.doorClosed(
122                 new DoorEvent( this, location ) );
123         }
124     }
125 } // fim do método closeDoor
126
127 // devolve se a Door está aberta ou fechada
128 public boolean isDoorOpen()
129 {
130     return open;
131 }
132
133 // invocado depois que o Elevator partiu
134 public void elevatorDeparted( ElevatorMoveEvent moveEvent ) {}
135
136 // invocado quando o Elevator chegou
137 public void elevatorArrived( ElevatorMoveEvent moveEvent )
138 {
139     openDoor( moveEvent.getLocation() );
140 }
141 }

```

Fig. H.7 A classe **Door**, que representa uma **Door** no modelo, informa aos ouvintes quando uma **Door** abriu ou fechou (parte 3 de 3).

A Fig. 15.12 indica que a classe **Door** contém um atributo **boolean open** (linha 14) para representar o estado da **Door** (aberta ou fechada). A Fig. 15.12 também indica que a classe **Door** contém os métodos **openDoor** (linhas 52 a 99) e **closeDoor** (linhas 102 a 125). O método **openDoor** envia um evento **doorOpened** para todos os **DoorListeners** registrados (a **Door** passa um objeto **DoorEvent** para o método **doorOpened** de cada **DoorListener** registrado), e o método **closeDoor** envia um evento **doorClosed** para todos os **DoorListeners** registrados. O conjunto **doorListeners** (linha 20) armazena todos os **DoorListeners** registrados. O **DoorListener** que queira receber **DoorEvents** de uma **Door** deve invocar o método **addDoorListener** (linhas 32 a 39); aqueles ouvintes que não queiram mais ser um **DoorListener** para aquela **Door** devem invocar **removeDoorListener** (linhas 42 a 49).

A linha 56 do método **openDoor** abre a **Door** ajustando **open** para **true**. As linhas 66 a 73 percorrem o conjunto **doorListeners** e enviam para cada objeto um evento **doorOpened**. As linhas 60 a 63 usam um bloco **synchronized** para obter o **Iterator** do **Set doorListeners**, porque uma **Person**, a qualquer momento, pode adicionar ou remover a si mesma deste **Set**. Se o método **openDoor** estiver percorrendo **doorListeners** quando uma **Person** adiciona ou remove a si mesma de **doorListeners**, a JVM dispara uma **ConcurrentModificationException** – a **Person** está modificando o **Set** enquanto o método **openDoor** está percorrendo o mesmo **Set**. O método **openDoor** evita esta situação com o bloco **synchronized**.

O método **openDoor** recebe como argumento uma referência **Location** para o **Floor** em que aquela **Door** deve abrir. As linhas 71 e 72 enviam um **DoorEvent** usando a referência **Location** para todos os **DoorListeners** registrados. O método **closeDoor** ajusta **open** para **false** (fechando assim a **Door**) e invoca o método **doorClosed** para todos os **DoorListeners** registrados.

Decidimos na Seção 15.12 tornar a classe **Door** ativa, de modo que a **Door** fecha a si mesma três segundos após ter sido aberta. As linhas 78 a 95 do método **openDoor** instanciam uma *thread* que trata desta responsabili-

dade. O método `run` (linhas 81 a 93) coloca esta *thread* para dormir por três segundos e depois fecha a `Door`. A linha 97 do método `openDoor` dispara a *thread*.

Finalmente, de acordo com a Fig. H.2, `Door` implementa `ElevatorMoveListener`. Em nossa simulação, o `Elevator` invoca o método `elevatorDeparted` (linha 134) quando o `Elevator` partiu e invoca o método `elevatorArrived` (linhas 137 a 140) quando o `Elevator` chegou. O método `elevatorArrived` chama o método `openDoor` – a `Door` se abre quando o `Elevator` chegou. O método `elevatorDeparted` não executa nenhuma ação. À primeira vista, você pode ficar se perguntando porque este método não chama `closeDoor`. A razão é que uma `Door` deve fechar antes de o `Elevator` ter partido, de modo que os passageiros não fiquem feridos – em nosso modelo, o `Elevator` chama o método `closeDoor` antes de chamar `elevatorDeparted`.

H.5 A classe Button

`Buttons` (Fig. H.8) também são importantes para o modelo, porque eles sinalizam ao `Elevator` para se mover entre os `Floors`. A Fig. 15.12 indica que a classe `Button` contém o atributo `boolean pressed` (linha 14) para representar o estado do `Button` (pressionado ou desligado). A Fig. 15.12 também indica que a classe `Button` contém os métodos `pressButton` (linhas 23 a 29) e `resetButton` (linhas 32 a 38).

```

1  // Button.java
2  // Envia ButtonEvents para ButtonListeners quando acessada
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes Deitel
6  import com.deitel.jhtp4.elevator.event.*;
7
8  public class Button implements ElevatorMoveListener {
9
10     // ButtonListener
11     private ButtonListener buttonListener = null;
12
13     // representa se o Button está pressionado
14     private boolean pressed = false;
15
16     // configura o ouvinte
17     public void setButtonListener( ButtonListener listener )
18     {
19         buttonListener = listener;
20     }
21
22     // pressiona Button e envia ButtonEvent
23     public void pressButton( Location location )
24     {
25         pressed = true;
26
27         buttonListener.buttonPressed(
28             new ButtonEvent( this, location ) );
29     }
30
31     // desliga Button e envia ButtonEvent
32     public void resetButton( Location location )
33     {
34         pressed = false;
35
36         buttonListener.buttonReset(
37             new ButtonEvent( this, location ) );
38     }

```

Fig. H.8 A classe `Button`, que representa um `Button` no modelo, informa aos ouvintes quando o `Button` foi pressionado ou desligado (parte 1 de 2).

```

39
40     // devolve se o botão está pressionado
41     public boolean isButtonPressed()
42     {
43         return pressed;
44     }
45
46     // invocado quando o Elevator partiu
47     public void elevatorDeparted( ElevatorMoveEvent moveEvent ) {}
48
49     // invocado quando o Elevator chegou
50     public void elevatorArrived( ElevatorMoveEvent moveEvent )
51     {
52         resetButton( moveEvent.getLocation() );
53     }
54 }

```

Fig. H.8 A classe **Button**, que representa um **Button** no modelo, informa aos ouvintes quando o **Button** foi pressionado ou desligado (parte 2 de 2).

O método **pressButton** envia um evento **buttonPressed** para o **ButtonListener** registrado (linha 11) e o método **resetButton** envia um evento **buttonReset** para o **ButtonListener**. O método **setButtonListener** (linhas 17 a 20) permite a um objeto receber **ButtonEvents** registrando a si mesmo como o **ButtonListener**.

A linha 25 do método **pressButton** ajusta o atributo **pressed** para **true** e as linhas 27 e 28 passam um **ButtonEvent** para o método **buttonPressed** do **buttonListener**. A linha 34 do método **resetButton** ajusta o atributo **pressed** para **false** e as linhas 36 e 37 passam um **ButtonEvent** para o método **buttonReset** do **buttonListener**.

Finalmente, de acordo com a Fig. H.2, a classe **Button** implementa a interface **ElevatorMoveListener**. O método **elevatorArrived** (linhas 50 a 53) chama o método **resetButton** para desligar o **Button**.

H.6 A classe **ElevatorShaft**

A classe **ElevatorShaft** (Fig. H.9) representa o poço de elevador no qual o **Elevator** se desloca no modelo. A maioria dos métodos na classe **ElevatorShaft** acessa variáveis **private**, espera por mensagens do **Elevator** e envia eventos “borbulhando para cima” para o **ElevatorModel**, que os envia para a **ElevatorView**. De acordo com o diagrama de classes da Fig. 15.21, a classe **ElevatorShaft** contém um objeto **Elevator**, dois objetos **Button**, dois objetos **Door** em dois objetos **Light**. Os objetos **Button**, **Door** e **Light** se referem aos botões, às portas e às luzes em cada **Floor**. A linha 15 declara o objeto **Elevator** – **elevator**. As linhas 18 e 19 declaram os **Buttons** **firstFloorButton** e **secondFloorButton**. As linhas 22 e 23 declaram as **Doors** **firstFloorDoor** e **secondFloorDoor**. As linhas 26 e 27 declaram as **Lights** **firstFloorLight** e **secondFloorLight**. As linhas 169 a 208 fornecem métodos para acessar referências para estes objetos.

```

1  // ElevatorShaft.java
2  // Representa o poço do elevador, que contém o elevador
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes do núcleo de Java
6  import java.util.*;
7
8  // Pacotes Deitel
9  import com.deitel.jhtp4.elevator.event.*;
10

```

Fig. H.9 A classe **ElevatorShaft**, que representa o **ElevatorShaft**, que envia eventos do **Elevator** para o **ElevatorModel** (parte 1 de 6).

```

11 public class ElevatorShaft implements ElevatorMoveListener,
12    LightListener, BellListener {
13
14     // Elevator
15     private Elevator elevator;
16
17     // Buttons nos Floors
18     private Button firstFloorButton;
19     private Button secondFloorButton;
20
21     // Doors nos Floors
22     private Door firstFloorDoor;
23     private Door secondFloorDoor;
24
25     // Lights nos Floors
26     private Light firstFloorLight;
27     private Light secondFloorLight;
28
29     // ouvintes
30     private DoorListener doorListener;
31     private ButtonListener buttonListener;
32     private LightListener lightListener;
33     private BellListener bellListener;
34     private Set elevatorMoveListeners;
35
36     // construtor inicializa componentes agregados
37     public ElevatorShaft( Floor firstFloor, Floor secondFloor )
38     {
39         // instancia Set para ElevatorMoveListeners
40         elevatorMoveListeners = new HashSet( 1 );
41
42         // classe interna anônima espera por ButtonEvents
43         ButtonListener floorButtonListener =
44             new ButtonListener() {
45
46             // chamado quando o Button do Floor foi pressionado
47             public void buttonPressed( ButtonEvent buttonEvent )
48             {
49                 // pede para elevador se mover para location
50                 Location location = buttonEvent.getLocation();
51                 buttonListener.buttonPressed( buttonEvent );
52                 elevator.requestElevator( location );
53             }
54
55             // chamado quando o Button do Floor foi desligado
56             public void buttonReset( ButtonEvent buttonEvent )
57             {
58                 buttonListener.buttonReset( buttonEvent );
59             }
60         }; // fim da classe interna anônima
61
62         // instancia Buttons do Floor
63         firstFloorButton = new Button();
64         secondFloorButton = new Button();
65
66         // registra ButtonListener anônimo com Buttons do Floor
67         firstFloorButton.setButtonListener(
68             floorButtonListener );
69         secondFloorButton.setButtonListener(

```

Fig. H.9 A classe `ElevatorShaft`, que representa o `ElevatorShaft`, que envia eventos do `Elevator` para o `ElevatorModel` (parte 2 de 6).

```

70         floorButtonListener );
71
72         // Floor Buttons esperam por ElevatorMoveEvents
73         addElevatorMoveListener( firstFloorButton );
74         addElevatorMoveListener( secondFloorButton );
75
76         // classe interna anônima espera por DoorEvents
77         DoorListener floorDoorListener = new DoorListener() {
78
79             // chamado quando a Door do Floor abriu
80             public void doorOpened( DoorEvent doorEvent )
81             {
82                 // passa evento adiante, para doorListener
83                 doorListener.doorOpened( doorEvent );
84             }
85
86             // chamado quando a Door do Floor fechou
87             public void doorClosed( DoorEvent doorEvent )
88             {
89                 // passa evento adiante, para doorListener
90                 doorListener.doorClosed( doorEvent );
91             }
92         }; // fim da classe interna anônima
93
94         // instancia Floor Doors
95         firstFloorDoor = new Door();
96         secondFloorDoor = new Door();
97
98         // registra DoorListener anônimo com as Doors dos Floors
99         firstFloorDoor.addDoorListener( floorDoorListener );
100        secondFloorDoor.addDoorListener( floorDoorListener );
101
102        // instancia Lights, então espera por LightEvents
103        firstFloorLight = new Light();
104        addElevatorMoveListener( firstFloorLight );
105        firstFloorLight.setLightListener( this );
106
107        secondFloorLight = new Light();
108        addElevatorMoveListener( secondFloorLight );
109        secondFloorLight.setLightListener( this );
110
111        // instancia objeto Elevator
112        elevator = new Elevator( firstFloor, secondFloor );
113
114        // se registra ElevatorMoveEvents do elevador
115        elevator.addElevatorMoveListener( this );
116
117        // espera por BellEvents do elevador
118        elevator.setBellListener( this );
119
120        // classe interna anônima espera por ButtonEvents
121        // do elevador
122        elevator.setButtonListener(
123            new ButtonListener() {
124
125                // invocado quando o botão foi pressionado
126                public void buttonPressed( ButtonEvent buttonEvent )
127                {

```

Fig. H.9 A classe `ElevatorShaft`, que representa o `ElevatorShaft`, que envia eventos do `Elevator` para o `ElevatorModel` (parte 3 de 6).

```

128         // envia evento para ouvinte
129         buttonListener.buttonPressed( buttonEvent );
130     }
131
132     // invocado quando o botão foi desligado
133     public void buttonReset( ButtonEvent buttonEvent )
134     {
135         // envia evento para ouvinte
136         buttonListener.buttonReset(
137             new ButtonEvent( this, elevator ) );
138     }
139 } // fim da classe interna anônima
140 );
141
142 // classe interna anônima espera por DoorEvents
143 // do elevador
144 elevator.setDoorListener(
145     new DoorListener() {
146
147         // invocado quando a porta abriu
148         public void doorOpened( DoorEvent doorEvent )
149         {
150             // envia evento para ouvinte
151             doorListener.doorOpened( doorEvent );
152         }
153
154         // invocado quando a porta fechou
155         public void doorClosed( DoorEvent doorEvent )
156         {
157             // envia evento para ouvinte
158             doorListener.doorClosed( doorEvent );
159         }
160     } // fim da classe interna anônima
161 );
162
163 // dispara Thread do Elevator
164 elevator.start();
165
166 } // fim do construtor ElevatorShaft
167
168 // obtém Elevator
169 public Elevator getElevator()
170 {
171     return elevator;
172 }
173
174 // obtém Door no primeiro Floor
175 public Door getFirstFloorDoor()
176 {
177     return firstFloorDoor;
178 }
179
180 // obtém Door no segundo Floor
181 public Door getSecondFloorDoor()
182 {
183     return secondFloorDoor;
184 }
185

```

Fig. H.9 A classe `ElevatorShaft`, que representa o `ElevatorShaft`, que envia eventos do `Elevator` para o `ElevatorModel` (parte 4 de 6).

```

186 // obtém Button no primeiro Floor
187 public Button getFirstFloorButton()
188 {
189     return firstFloorButton;
190 }
191
192 // obtém Button no segundo Floor
193 public Button getSecondFloorButton()
194 {
195     return secondFloorButton;
196 }
197
198 // obtém Light no primeiro Floor
199 public Light getFirstFloorLight()
200 {
201     return firstFloorLight;
202 }
203
204 // obtém Light no segundo Floor
205 public Light getSecondFloorLight()
206 {
207     return secondFloorLight;
208 }
209
210 // invocado quando a Bell toca
211 public void bellRang( BellEvent bellEvent )
212 {
213     bellListener.bellRang( bellEvent );
214 }
215
216 // invocado quando a Light acende
217 public void lightTurnedOn( LightEvent lightEvent )
218 {
219     lightListener.lightTurnedOn( lightEvent );
220 }
221
222 // invocado quando a Light apaga
223 public void lightTurnedOff( LightEvent lightEvent )
224 {
225     lightListener.lightTurnedOff( lightEvent );
226 }
227
228 // invocado quando o Elevator parte
229 public void elevatorDeparted( ElevatorMoveEvent moveEvent )
230 {
231     Iterator iterator = elevatorMoveListeners.iterator();
232
233     // percorre o Set de ouvintes de ElevatorMoveEvent
234     while ( iterator.hasNext() ) {
235
236         // obtém ElevatorMoveListener respectivos do Set
237         ElevatorMoveListener listener =
238             ( ElevatorMoveListener ) iterator.next();
239
240         // envia ElevatorMoveEvent para ouvinte "this"
241         listener.elevatorDeparted( moveEvent );
242     }
243 } // fim do método elevatorDeparted
244

```

Fig. H.9 A classe `ElevatorShaft`, que representa o `ElevatorShaft`, que envia eventos do `Elevator` para o `ElevatorModel` (parte 5 de 6).

```

245 // invocado quando o Elevator chega
246 public void elevatorArrived( ElevatorMoveEvent moveEvent )
247 {
248     // obtém iterador do Set
249     Iterator iterator = elevatorMoveListeners.iterator();
250
251     // obtém próximo DoorListener
252     while ( iterator.hasNext() ) {
253
254         // obtém próximo ElevatorMoveListener do Set
255         ElevatorMoveListener listener =
256             ( ElevatorMoveListener ) iterator.next();
257
258         // envia ElevatorMoveEvent para ouvinte "this"
259         listener.elevatorArrived( moveEvent );
260
261     } // fim do laço while
262 } // fim do método elevatorArrived
263
264 // configura ouvinte para DoorEvents
265 public void setDoorListener( DoorListener listener )
266 {
267     doorListener = listener;
268 }
269
270 // configura ouvinte para ButtonEvents
271 public void setButtonListener( ButtonListener listener )
272 {
273     buttonListener = listener;
274 }
275
276 // adiciona ouvinte para ElevatorMoveEvents
277 public void addElevatorMoveListener(
278     ElevatorMoveListener listener )
279 {
280     elevatorMoveListeners.add( listener );
281 }
282
283 // configura ouvinte para LightEvents
284 public void setLightListener( LightListener listener )
285 {
286     lightListener = listener;
287 }
288
289 // configura ouvinte para BellEvents
290 public void setBellListener( BellListener listener )
291 {
292     bellListener = listener;
293 }
294 }

```

Fig. H.9 A classe `ElevatorShaft`, que representa o `ElevatorShaft`, que envia eventos do `Elevator` para o `ElevatorModel` (parte 6 de 6).

A principal responsabilidade do `ElevatorShaft` é receber eventos de outros objetos e depois enviar estes eventos para o `ElevatorModel` (o `ElevatorModel` envia os eventos para a `ElevatorView`, que exibe o modelo em funcionamento). O `ElevatorShaft` contém referências para diversos objetos *listener* diferentes, como `DoorListener`, `ButtonListener`, `LightListener`, `BellListener` e diversos `ElevatorMoveListeners`. As linhas 30 a 34 declaram estes objetos *listeners* – a linha 34 declara um `Set` para guardar vários `ElevatorMoveListeners`, porque os `Buttons`, as `Doors`, as `Lights` e o `ElevatorModel` são todos `Ele-`

vatorMoveListeners. As linhas 265 a 293 fornecem métodos que permitem aos objetos registrar a si mesmos para vários eventos.

O construtor (linhas 37 a 166) instancia objetos *listener* de diversas classes internas – estes objetos *listener* recebem eventos de outros objetos e depois enviam novamente os eventos para os objetos *listener* definidos nas linhas 30 a 34. Por exemplo, as linhas 43 a 60 declaram um objeto **ButtonListener** chamado **floorButtonListener**, que contém a lógica para quando um **Button** em um **Floor** foi pressionado ou desligado. As linhas 63 a 70 instanciam **firstFloorButton** e **secondFloorButton**, depois registram **floorButtonListener** como **ButtonListener** para os dois objetos **Button**. Quando qualquer **Button** foi pressionado, aquele **Button** invoca o método **buttonPressed** (linhas 47 a 53) do **floorButtonListener**. Quando qualquer **Button** foi desligado, aquele **Button** invoca o método **buttonReset** de **floorButtonListener** (linhas 56 a 59). O método **buttonPressed** chama o **Elevator** invocando o método **requestElevator** de **Elevator** – o método **buttonPressed** passa uma referência **Location** do **Floor** que gerou o **ButtonEvent**. Tanto o método **buttonPressed** quanto **buttonReset** enviam o **ButtonEvent** para o **ButtonListener** definido na linha 31 (que, neste caso, é **ElevatorModel**).

As linhas 77 a 92 declaram um objeto **DoorListener** chamado **floorDoorListener**, que contém a lógica para quando uma **Door** em um **Floor** abriu ou fechou. As linhas 93 a 100 instanciam **firstFloorDoor** e **secondFloorDoor**, depois registram **floorDoorListener** como **DoorListener** para os dois objetos **Door**. Quando qualquer uma das **Doors** abriu, essa **Door** chama o método **doorOpened** (linhas 80 a 84) do **floorDoorListener**. Quando qualquer uma das **Doors** fechou, ela chama o método **doorClosed** (linhas 87 a 91) do **floorDoorListener**. Os dois métodos enviam o **DoorEvent** para o **DoorListener** declarado na linha 30 (que, neste caso, é **ElevatorModel**).

As linhas 112 a 115 instanciam o **Elevator** e registram no **Elevator** o **ElevatorShaft** como um **ElevatorMoveListener**. Quando o **Elevator** partiu, ele invoca o método **elevatorDeparted** (linhas 229 a 243), que informa a todos os objetos em **elevatorMoveListeners** sobre a partida. Quando o **Elevator** chegou, ele invoca o método **ElevatorArrived** (linhas 246 a 262), que informa a todos os objetos em **elevatorMoveListeners** sobre a chegada.

H.7 As classes **Light** e **Bell**

A classe **Light** (Fig. H.10) representa as **Lights** nos **Floors** no modelo. Os objetos da classe **Light** ajudam a decorar a visão enviando eventos para a **ElevatorView** através da técnica de “borbulhar para cima” descrita anteriormente. Em nossa simulação, a **ElevatorView** liga e desliga a **Light** na visão quando recebe um evento **lightTurnedOn** ou **lightTurnedOff**, respectivamente.

```

1  // Light.java
2  // Light acende ou apaga uma luz
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes Deitel
6  import com.deitel.jhtp4.elevator.event.*;
7
8  public class Light implements ElevatorMoveListener {
9
10     // estado da Light (acesa/apagada)
11     private boolean lightOn;
12
13     // tempo antes que a Light apague automaticamente (três segundos)
14     public static final int AUTOMATIC_TURNOFF_DELAY = 3000;
15
16     // LightListener espera que a Light acenda/apague
17     private LightListener lightListener;
18
19     // lugar onde a Light acendeu ou apagou
20     private Location lightLocation;
```

Fig. H.10 A classe **Light** representa uma **Light** no **Floor** no modelo (parte 1 de 3).

```

21
22 // configura LightListener
23 public void setLightListener( LightListener listener )
24 {
25     lightListener = listener;
26 }
27
28 // acende Light
29 public void turnOnLight( Location location )
30 {
31     if ( !lightOn ) {
32
33         lightOn = true;
34
35         // envia LightEvent para LightListener
36         lightListener.lightTurnedOn(
37             new LightEvent( this, location ) );
38
39         lightLocation = location;
40
41         // declara Thread que assegura apagamento automático da Light
42         Thread thread = new Thread(
43             new Runnable() {
44
45                 public void run()
46                 {
47                     // desliga Light se ligada por mais de três segundos
48                     try {
49                         Thread.sleep( AUTOMATIC_TURNOFF_DELAY );
50                         turnOffLight( lightLocation );
51                     }
52
53                     // trata exceção se interrompida
54                     catch ( InterruptedException exception ) {
55                         exception.printStackTrace();
56                     }
57                 }
58             } // fim da classe interna anônima
59         );
60
61         thread.start();
62     }
63 } // fim do método turnOnLight
64
65 // desliga Light
66 public void turnOffLight( Location location )
67 {
68     if ( lightOn ) {
69
70         lightOn = false;
71
72         // envia LightEvent para LightListener
73         lightListener.lightTurnedOff(
74             new LightEvent( this, location ) );
75     }
76 } // fim do método turnOffLight
77
78 // devolve se a Light está acesa ou apagada
79 public boolean isLightOn()
80 {

```

Fig. H.10 A classe Light representa uma Light no Floor no modelo (parte 2 de 3).

```

81     return lightOn;
82 }
83
84 // invocado quando o Elevator partiu
85 public void elevatorDeparted(
86     ElevatorMoveEvent moveEvent )
87 {
88     turnOffLight( moveEvent.getLocation() );
89 }
90
91 // invocado quando o Elevator chegou
92 public void elevatorArrived(
93     ElevatorMoveEvent moveEvent )
94 {
95     turnOnLight( moveEvent.getLocation() );
96 }
97 }

```

Fig. H.10 A classe **Light** representa uma **Light** no **Floor** no modelo (parte 3 de 3).

De acordo com a Fig. 15.21, a classe **Light** contém o atributo **lightOn** (linha 11), que representa o estado da **Light** (ligada ou desligada). Além disso, a Fig. 15.21 especifica que a classe **Light** contém os métodos **turnOnLight** (linhas 29 a 63) e **turnOffLight** (linhas 66 a 76). A linha 33 do método **turnOnLight** ajusta o atributo **lightOn** para **true** e as linhas 36 e 37 chamam o método **lightTurnedOn** do **lightListener** (linha 17). Em nosso modelo, o **ElevatorShaft** é o **lightListener** – o **ElevatorShaft** recebe eventos da **Light** e os envia para o **ElevatorModel**, que os envia para a **ElevatorView**. O **ElevatorShaft** usa o método **setLightListener** (linhas 23 a 26) para se registrar para **LightEvents**. O método **turnOffLight** ajusta o atributo **lightOn** para **false**, depois chama o método **lightTurnedOff** do **lightListener**.

Decidimos na Seção 15.12 tornar a classe **Light** ativa, de modo que a **Light** desliga a si mesma três segundos após ter sido acesa. As linhas 42 a 59 do método **turnOnLight** instanciam uma *thread* que trata desta responsabilidade. O método **run** (linhas 45 a 57) coloca esta *thread* para dormir por três segundos, depois desliga a **Light**. A linha 61 do método **turnOnLight** dispara a *thread*.

De acordo com a Fig. H.2, a classe **Light** implementa a interface **ElevatorMoveListener**. As linhas 85 a 89 e as linhas 92 a 96 listam os métodos **elevatorDeparted** e **elevatorArrived**, respectivamente. Em nosso modelo, a **Light** apaga quando o **Elevator** partiu e a **Light** acende quando o **Elevator** chegou.

A classe **Bell** (Fig. H.11) representa a campainha no modelo e envia um evento **bellRang** para um **BellListener** quando a **Bell** toca. Este evento, em algum momento, “borbulha para cima” para a **ElevatorView**. A **ElevatorView** reproduz um clipe de áudio de uma campainha tocando quando recebe um evento **bellRang**.

```

1  // Bell.java
2  // Representa Bell na simulação
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes Deitel
6  import com.deitel.jhtp4.elevator.event.*;
7
8  public class Bell implements ElevatorMoveListener {
9
10     // BellListener espera por objeto BellEvent
11     private BellListener bellListener;
12

```

Fig. H.11 A classe **Bell** representa a **Bell** no modelo (parte 1 de 2).

```

13 // toca a campainha e envia objeto BellEvent para o ouvinte
14 private void ringBell( Location location )
15 {
16     if ( bellListener != null )
17         bellListener.bellRang(
18             new BellEvent( this, location ) );
19 }
20
21 // configura BellListener
22 public void setBellListener( BellListener listener )
23 {
24     bellListener = listener;
25 }
26
27 // invocado quando o Elevator partiu
28 public void elevatorDeparted( ElevatorMoveEvent moveEvent ) {}
29
30 // invocado quando o Elevator chegou
31 public void elevatorArrived( ElevatorMoveEvent moveEvent )
32 {
33     ringBell( moveEvent.getLocation() );
34 }
35 }

```

Fig. H.11 A classe **Bell** representa a **Bell** no modelo (parte 2 de 2).

De acordo com a Fig. 15.21, a classe **Bell** não contém atributos, porque a **Bell** não muda de estado. Entretanto, a Fig. 15.21 especifica que a classe **Bell** contém o método **ringBell** (linhas 14 a 19), que toca a **Bell** invocando o método **bellRang** do **BellListener** **bellListener** (linha 11). Em nossa simulação, o **Elevator** é o **bellListener** – o **Elevator** recebe o evento da **Bell**, depois envia o evento para o **ElevatorShaft**, que envia o evento para o **ElevatorModel**, que envia o evento para a **ElevatorView**. A **ElevatorView** então toca um clipe de áudio de uma campainha tocando. O **Elevator** usa o método **setBellListener** (linhas 22 a 24) para se registrar para receber **BellEvents** da **Bell**.

De acordo com a Fig. H.2, a classe **Bell** implementa a interface **ElevatorMoveListener**. A linha 28 e as linhas 31 a 34 listam os métodos **elevatorDeparted** e **elevatorArrived**, respectivamente. Em nossa simulação, a **Bell** toca quando o **Elevator** chegou.

H.8 A classe **Elevator**

A classe **Elevator** (Fig. H.12) representa o carro do elevador que se movimenta entre os dois **Floors** no **ElevatorShaft** enquanto está carregando uma **Person**. De acordo com o diagrama de classes da Fig. 15.21, a classe **Elevator** contém um objeto de cada uma das classes **Button**, **Door** e **Bell** – as linhas 37 a 39 declaram os objetos **elevatorButton**, **elevatorDoor** e **bell**. Como discutido na Seção 9.23, a classe **Elevator** estende a superclasse **Location**, porque o **Elevator** é um lugar que a **Person** pode ocupar. A classe **Elevator** implementa os métodos **getButton** (linhas 232 a 235) e **getDoor** (linhas 238 a 241) fornecidos pela classe **Location**. O método **getButton** devolve o **elevatorButton** e o método **getDoor** devolve o **elevatorDoor**. De acordo com a Fig. 15.21, também devemos incluir dois objetos **Location** – um chamado **currentFloor** (linha 22), que representa o **Floor** atual que está sendo atendido, e outro chamado **destinationFloor** (linha 25), que representa o **Floor** no qual o **Elevator** vai chegar. Além disso, a Fig. 15.21 especifica que a classe **Elevator** exige a variável **boolean moving** (linha 19), que descreve se o **Elevator** está se movendo ou está ocioso, e a variável **boolean summoned** (linha 28), que descreve se o **Elevator** foi chamado. Além disso, a classe **Elevator** usa a constante **int TRAVEL_TIME** (linha 44), que indica o tempo de viagem de cinco segundos entre **Floors**.

```

1  // Elevator.java
2  // Se movimentação entre Floors no ElevatorShaft
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes do núcleo de Java
6  import java.util.*;
7
8  // Pacotes Deitel
9  import com.deitel.jhtp4.elevator.event.*;
10 import com.deitel.jhtp4.elevator.ElevatorConstants;
11
12 public class Elevator extends Location implements Runnable,
13     BellListener, ElevatorConstants {
14
15     // gerencia a thread do Elevator
16     private boolean elevatorRunning = false;
17
18     // descreve o estado do Elevator (parado ou em movimento)
19     private boolean moving = false;
20
21     // Floor atual
22     private Location currentFloorLocation;
23
24     // Floor de destino
25     private Location destinationFloorLocation;
26
27     // Elevator precisa atender outro Floor
28     private boolean summoned;
29
30     // objetos listener
31     private Set elevatorMoveListeners;
32     private ButtonListener elevatorButtonListener;
33     private DoorListener elevatorDoorListener;
34     private BellListener bellListener;
35
36     // Door, Button e Bell no Elevator
37     private Door elevatorDoor;
38     private Button elevatorButton;
39     private Bell bell;
40
41     public static final int ONE_SECOND = 1000;
42
43     // tempo necessário para se mover entre Floors (cinco segundos)
44     private static final int TRAVEL_TIME = 5 * ONE_SECOND;
45
46     // tempo máximo de viagem para o Elevator (20 minutos)
47     private static final int MAX_TRAVEL_TIME =
48         20 * 60 * ONE_SECOND;
49
50     // thread do Elevator para tratar de movimento assíncrono
51     private Thread thread;
52
53     // construtor cria variáveis; registra-se para ButtonEvents
54     public Elevator( Floor firstFloor, Floor secondFloor )
55     {
56         setLocationName( ELEVATOR_NAME );
57
58         // instancia Door, Button e Bell do Elevator
59         elevatorDoor = new Door();

```

Fig. H.12 A classe `Elevator` representa o Elevador se movendo entre dois Floors, operando assincronamente com outros objetos (parte 1 de 7).

```

60     elevatorButton = new Button();
61     bell = new Bell();
62
63     // registra Elevator para BellEvents
64     bell.setBellListener( this );
65
66     // instancia Set de ouvintes
67     elevatorMoveListeners = new HashSet( 1 );
68
69     // inicia Elevator no primeiro Floor
70     currentFloorLocation = firstFloor;
71     destinationFloorLocation = secondFloor;
72
73     // registra elevatorButton para ElevatorMoveEvents
74     addElevatorMoveListener( elevatorButton );
75
76     // registra elevatorDoor para ElevatorMoveEvents
77     addElevatorMoveListener( elevatorDoor );
78
79     // registra bell para ElevatorMoveEvents
80     addElevatorMoveListener( bell );
81
82     // classe interna anônima espera por ButtonEvents
83     // de elevatorButton
84     elevatorButton.setButtonListener(
85         new ButtonListener() {
86
87             // invocado quando elevatorButton foi pressionado
88             public void buttonPressed( ButtonEvent buttonEvent )
89             {
90                 // envia ButtonEvent para ouvinte
91                 elevatorButtonListener.buttonPressed(
92                     buttonEvent );
93
94                 // começa a movimentar Elevator para o Floor de destino
95                 setMoving( true );
96             }
97
98             // invocado quando elevatorButton foi desligado
99             public void buttonReset( ButtonEvent buttonEvent )
100             {
101                 // envia ButtonEvent para ouvinte
102                 elevatorButtonListener.buttonReset(
103                     buttonEvent );
104             }
105         } // fim da classe interna anônima
106     );
107
108     // classe interna anônima espera por DoorEvents
109     // de elevatorDoor
110     elevatorDoor.addDoorListener(
111         new DoorListener() {
112
113             // invocado quando a elevatorDoor abriu
114             public void doorOpened( DoorEvent doorEvent )
115             {
116                 // obtém Location associada com o DoorEvent
117                 Location location = doorEvent.getLocation();
118                 String locationName = location.getLocationName();

```

Fig. H.12 A classe `Elevator` representa o Elevator se movendo entre dois Floors, operando assincronamente com outros objetos (parte 2 de 7).

```

119
120         // abre Door no Floor de Location
121         if ( !locationName.equals( ELEVATOR_NAME ) )
122             location.getDoor().openDoor( location );
123
124         // envia DoorEvent para ouvinte
125         elevatorDoorListener.doorOpened( new DoorEvent(
126             doorEvent.getSource(), Elevator.this ));
127     }
128
129     // invocado quando a elevatorDoor fechou
130     public void doorClosed( DoorEvent doorEvent )
131     {
132         // obtém Location associada com o DoorEvent
133         Location location = doorEvent.getLocation();
134         String locationName = location.getLocationName();
135
136         // fecha Door no Floor de Location
137         if ( !locationName.equals( ELEVATOR_NAME ) )
138             location.getDoor().closeDoor( location );
139
140         // envia DoorEvent para ouvinte
141         elevatorDoorListener.doorClosed( new DoorEvent(
142             doorEvent.getSource(), Elevator.this ));
143     }
144 } // fim da classe interna anônima
145 );
146 } // fim do construtor Elevator
147
148 // troca Location do Floor atual com a Location do Floor oposto
149 private void changeFloors()
150 {
151     Location location = currentFloorLocation;
152     currentFloorLocation = destinationFloorLocation;
153     destinationFloorLocation = location;
154 }
155
156 // dispara thread do Elevator
157 public void start()
158 {
159     if ( thread == null )
160         thread = new Thread( this );
161
162     elevatorRunning = true;
163     thread.start();
164 }
165
166 // pára a thread do Elevator; o método run termina
167 public void stopElevator()
168 {
169     elevatorRunning = false;
170 }
171
172 // método run da thread do Elevator
173 public void run()
174 {
175     while ( isElevatorRunning() ) {
176
177         // permanece ociosa até ser despertada

```

Fig. H.12 A classe `Elevator` representa o Elevador se movendo entre dois Floors, operando assincronamente com outros objetos (parte 3 de 7).

```

178         while ( !isMoving() )
179             pauseThread( 10 );
180
181         // fecha elevatorDoor
182         getDoor().closeDoor( currentFloorLocation );
183
184         // fechar a Door leva um segundo
185         pauseThread( ONE_SECOND );
186
187         // dispara evento elevatorDeparted
188         sendDepartureEvent( currentFloorLocation );
189
190         // Elevator precisa de cinco segundos para se mover
191         pauseThread( TRAVEL_TIME );
192
193         // pára Elevator
194         setMoving( false );
195
196         // permuta Locations dos Floors
197         changeFloors();
198
199         // dispara evento elevatorArrived
200         sendArrivalEvent( currentFloorLocation );
201
202     } // fim do laço while
203
204 } // fim do método run
205
206 // invocado quando a Person anda no Elevator entre Floors
207 public synchronized void ride()
208 {
209     try {
210         Thread.sleep( MAX_TRAVEL_TIME );
211     }
212     catch ( InterruptedException exception ) {
213         // método doorOpened em Person interrompe o método sleep;
214         // Person terminou de andar no Elevator
215     }
216 }
217
218 // faz a thread simultânea parar por milissegundo
219 private void pauseThread( int milliseconds )
220 {
221     try {
222         Thread.sleep( milliseconds );
223     }
224
225     // trata se interrompida enquanto adormecida
226     catch ( InterruptedException exception ) {
227         exception.printStackTrace();
228     }
229 } // fim do método pauseThread
230
231 // devolve Button no Elevator
232 public Button getButton()
233 {
234     return elevatorButton;
235 }
236

```

Fig. H.12 A classe `Elevator` representa o Elevator se movendo entre dois Floors, operando assincronamente com outros objetos (parte 4 de 7).

```

237 // devolve Door no Elevator
238 public Door getDoor()
239 {
240     return elevatorDoor;
241 }
242
243 // configura se Elevator deve se mover
244 public void setMoving( boolean elevatorMoving )
245 {
246     moving = elevatorMoving;
247 }
248
249 // O Elevator está se movendo?
250 public boolean isMoving()
251 {
252     return moving;
253 }
254
255 // a thread do Elevator está sendo executada?
256 private boolean isElevatorRunning()
257 {
258     return elevatorRunning;
259 }
260
261 // registra ElevatorMoveListener para ElevatorMoveEvents
262 public void addElevatorMoveListener(
263     ElevatorMoveListener listener )
264 {
265     elevatorMoveListeners.add( listener );
266 }
267
268 // registra ButtonListener para ButtonEvents
269 public void setButtonListener( ButtonListener listener )
270 {
271     elevatorButtonListener = listener;
272 }
273
274 // registra DoorListener para DoorEvents
275 public void setDoorListener( DoorListener listener )
276 {
277     elevatorDoorListener = listener;
278 }
279
280 // registra BellListener para BellEvents
281 public void setBellListener( BellListener listener )
282 {
283     bellListener = listener;
284 }
285
286 // notifica todos os ElevatorMoveListeners da chegada
287 private void sendArrivalEvent( Location location )
288 {
289     // obtém iterador do Set
290     Iterator iterator = elevatorMoveListeners.iterator();
291
292     // obtém próximo DoorListener
293     while ( iterator.hasNext() ) {
294

```

Fig. H.12 A classe `Elevator` representa o Elevador se movendo entre dois Floors, operando assincronamente com outros objetos (parte 5 de 7).

```

295         // obtém próximo ElevatorMoveListener do Set
296         ElevatorMoveListener listener =
297             ( ElevatorMoveListener ) iterator.next();
298
299         // envia evento para ouvinte
300         listener.elevatorArrived( new
301             ElevatorMoveEvent( this, location ) );
302
303     } // fim do laço while
304
305     // atende chamada pendente, se existir uma
306     if ( summoned ) {
307         pauseThread( Door.AUTOMATIC_CLOSE_DELAY );
308         setMoving( true ); // começa a movimentar o Elevator
309     }
310
311     summoned = false; // chamada foi atendida
312
313 } // fim do método sendArrivalEvent
314
315 // notificar todos os ElevatorMoveListeners da partida
316 private void sendDepartureEvent( Location location )
317 {
318     // obtém iterador do Set
319     Iterator iterator = elevatorMoveListeners.iterator();
320
321     // obtém próximo DoorListener
322     while ( iterator.hasNext() ) {
323
324         // obtém próximo ElevatorMoveListener do Set
325         ElevatorMoveListener listener =
326             ( ElevatorMoveListener ) iterator.next();
327
328         // envia ElevatorMoveEvent para ouvinte "this"
329         listener.elevatorDeparted( new ElevatorMoveEvent(
330             this, currentFloorLocation ) );
331     } // fim do laço while
332 } // fim do método sendDepartureEvent
333
334 // chama o Elevator
335 public void requestElevator( Location location )
336 {
337     // se o Elevator estiver parado
338     if ( !isMoving() ) {
339
340         // se o Elevator estiver no mesmo Floor da chamada
341         if ( location == currentFloorLocation )
342
343             // Elevator já chegou; envia evento de chegada
344             sendArrivalEvent( currentFloorLocation );
345
346         // se o Elevator estiver no Floor oposto ao da chamada
347         else {
348
349             if ( getDoor().isDoorOpen() )
350                 pauseThread( Door.AUTOMATIC_CLOSE_DELAY );
351             setMoving( true ); // se move para outro Floor
352         }
353     }

```

Fig. H.12 A classe `Elevator` representa o Elevator se movendo entre dois Floors, operando assincronamente com outros objetos (parte 6 de 7).

```

354     }
355     else // se o Elevator estiver se movendo
356
357         // se o Elevator partiu do mesmo Floor que a chamada
358         if ( location == currentFloorLocation )
359             summoned = true;
360
361         // se o Elevator estiver se movendo para o Floor da chamada,
362         // continua se movendo
363
364     } // fim do método requestElevator
365
366     // invocado quando a campainha tocou
367     public void bellRang( BellEvent bellEvent )
368     {
369         // envia evento para bellListener
370         if ( bellListener != null )
371             bellListener.bellRang( bellEvent );
372     }
373 }

```

Fig. H.12 A classe **Elevator** representa o Elevador se movendo entre dois Floors, operando assincronamente com outros objetos (parte 7 de 7).

De acordo com a Fig. H.12, a classe **Elevator** implementa as interfaces **ButtonListener**, **DoorListener** e **BellListener** e, portanto, pode esperar por **ButtonEvents**, **DoorEvents** e **BellEvents**. A classe **Elevator** deve enviar estes eventos para um ouvinte (neste caso, o **ElevatorShaft**), de modo que estes eventos possam “borbulhar para cima” para a **ElevatorView**. A classe **Elevator** contém um **ButtonListener** chamado **elevatorButtonListener** (linha 32), um **DoorListener** chamado **elevatorDoorListener** (linha 33) e um **BellListener** chamado **bellListener** (linha 34). As linhas 269 a 284 listam os métodos **setButtonListener**, **setDoorListener** e **setBellListener**, que permitem que um objeto – como **ElevatorShaft** – se registre como um ouvinte para estes eventos.

A classe **Elevator** contém um **ButtonListener** anônimo (linhas 84 a 106) que se registra para **ButtonEvents** do **elevatorButton**. Quando uma **Person** pressionou o **elevatorButton**, o **ButtonListener** chama o método **buttonPressed** (linhas 88 a 96) do **ButtonListener**. As linhas 91 e 92 deste método chamam o método **buttonPressed** do **elevatorButtonListener** e a linha 95 informa ao **Elevator** para se mover com o método **setMoving**. Quando o **Button** foi desligado, o **ButtonListener** chama o método **buttonReset** (linhas 99 a 104) do **ButtonListener**. As linhas 102 e 103 deste método chamam o método **buttonReset** do **elevatorButtonListener**.

A classe **Elevator** contém um **DoorListener** anônimo (linhas 110 a 145) que se registra para **DoorEvents** da **elevatorDoor**. Quando a **elevatorDoor** abriu, o **DoorListener** chama o método **doorOpened** (linhas 114 a 127) deste **DoorListener**. As linhas 121 e 122 abrem a **Door** no **Floor** que gerou o evento e as linhas 125 e 126 chamam o método **doorOpened** do **elevatorDoorListener**. O método **doorOpened** garante que a **Door** no **Floor** abra antes de o passageiro sair do **Elevator**. Quando a **elevatorDoor** fechou, o **DoorListener** chama o método **doorClosed** (linhas 130 a 143) do **DoorListener**. As linhas 137 e 138 fecham a **Door** no **Floor** que gerou o evento e as linhas 141 e 142 chamam o método **doorClosed** do **elevatorDoorListener**.

A classe **Elevator** age como uma *thread* porque ela implementa a interface **Runnable**. O método **run** (linhas 173 a 204) trata do deslocamento entre os **Floors**. O método começa com o **Elevator** permanecendo ocioso em um laço **while** (linhas 178 e 179). A saída do laço acontece quando o método **buttonPressed** no **ButtonListener** anônimo chama o método **setMoving**.

Quando o **Elevator** sai do laço, o **Elevator** fecha a **elevatorDoor** (linha 182) e depois chama o método **private sendDepartureEvent** (linhas 316 a 333), para informar a todos os ouvintes – o **elevatorButton**, a **elevatorDoor**, a **bell** e o **ElevatorShaft** – da partida do **Elevator**. A classe **Elevator** contém o **Set elevatorMoveListeners** (linha 31), que armazena todos os **ElevatorMoveListeners** registrados. Os objetos que desejam receber **ElevatorMoveEvents** do **Elevator** devem chamar o método **addElevatorMoveListener** (linhas 262 a 266), que anexam aqueles objetos a **elevatorMoveListeners**.

O método `sendDepartureEvent` invoca o método `elevatorDeparted` de cada objeto *listener* no `Set elevatorMoveListeners`.

A linha 191 do método `run` permite ao `Elevator` deslocar-se para o `Floor` chamando o método `pauseThread` (linhas 219 a 229) – este simula o deslocamento invocando o método `sleep` da classe `Thread`. O `Elevator` pára de se movimentar quando sua *thread* desperta, após cinco segundos. A linha 197 chama o método `private changeFloors` (linhas 149 a 154), que permuta a `currentFloorLocation` e a `destinationFloorLocation`. A linha 200 chama o método `private sendArrivalEvent` (linhas 287 a 313), que invoca o método `elevatorArrived` de todos os ouvintes no `Set elevatorMoveListeners`. As linhas 306 a 309 do método `sendArrivalEvent` atendem a qualquer pedido pendente (por exemplo, a `Person` pressionou o `Button` no `Floor` do qual o `Elevator` partiu). Se existe um pedido pendente, a linha 308 invoca o método `setMoving` para mover o `Elevator` para o `Floor` oposto.

O método `requestElevator` (linhas 336 a 364) chama o `Elevator` e gera um pedido pendente. Em nosso modelo, o `ButtonListener` definido na classe interna do `ElevatorShaft` chama este método quando um `Button` em qualquer `Floor` foi pressionado. O diagrama de atividades da Fig. 5.30 especifica a lógica para o método `requestElevator`. Se o `Elevator` estiver ocioso e no mesmo `Floor` que o `Floor` da chamada, a linha 345 chama o método `sendArrivalEvent`, porque o `Elevator` já chegou. Se o `Elevator` estiver ocioso, mas no `Floor` oposto ao `Floor` da chamada, a linha 352 move o `Elevator` para o `Floor` oposto. Se o `Elevator` estiver se movendo para o `Floor` que gerou a chamada, o `Elevator` deve continuar se movendo para aquele `Floor`. Se o `Elevator` estiver se movendo, mas saindo do `Floor` que gerou a chamada, o `Elevator` deve se lembrar de voltar àquele `Floor` (linhas 358 e 359).

Finalmente, como mencionado na Seção 15.12, a classe `Elevator` contém o método `synchronized ride` (linhas 207 a 216). A `Person` chama este método para garantir exclusividade com o `Elevator`. O método `ride` assegura que dois objetos `Person` não podem ocupar o `Elevator` ao mesmo tempo. Quando um objeto `Person` invoca o método `ride`, esse objeto obtém um monitor sobre o objeto `Elevator`. Outros objetos não podem acessar o `Elevator` até que aquela `Person` libere o monitor quando sai do método `ride`.

H.9 A classe `Person`

A classe `Person` (Fig. H.13) representa uma `Person` que caminha através dos `Floors` e anda no `Elevator` em nossa simulação. De acordo com o diagrama de classes da Fig. 15.21, a classe `Person` contém um objeto da classe `Location` (linha 20) que representa o lugar em que a `Person` atual está no modelo (ou em um `Floor` ou no `Elevator`). Além disso, a Fig. 15.21 especifica que a `Person` exige o atributo `int ID` (linha 14) como um identificador único e o atributo `boolean moving` (linha 17), que indica se a pessoa está caminhando através do `Floor` ou esperando que uma `Door` se abra.

```

1  // Person.java
2  // Person andando no elevador
3  package com.deitel.jhtp4.elevator.model;
4
5  // Pacotes do núcleo de Java
6  import java.util.*;
7
8  // Pacotes Deitel
9  import com.deitel.jhtp4.elevator.event.*;
10
11 public class Person extends Thread implements DoorListener {
12
13     // número de identificação
14     private int ID = -1;
15
16     // representa se a Person está se movendo ou esperando
17     private boolean moving;
18
19     // referência para Location (no Floor ou no Elevator)

```

Fig. H.13 A classe `Person` representa a `Person` que anda no `Elevator`. A `Person` opera assincronamente com outros objetos (parte 1 de 6).

```

20     private Location location;
21
22     // objeto listener para PersonMoveEvents
23     private PersonMoveListener personMoveListener;
24
25     // tempo em milissegundos para caminhar até Button no Floor
26     private static final int TIME_TO_WALK = 3000;
27
28     // tempo máximo que a Person irá esperar pelo Elevator (10 minutos)
29     private static final int TIME_WAITING = 10 * 60 * 1000;
30
31     // tipos de mensagens que Person pode enviar
32     public static final int PERSON_CREATED = 1;
33     public static final int PERSON_ARRIVED = 2;
34     public static final int PERSON_ENTERING_ELEVATOR = 3;
35     public static final int PERSON_PRESSING_BUTTON = 4;
36     public static final int PERSON_EXITING_ELEVATOR = 5;
37     public static final int PERSON_EXITED = 6;
38
39     // construtor Person configura Location inicial
40     public Person( int identifier, Location initialLocation )
41     {
42         super();
43
44         ID = identifier;    // atribui identificador único
45         location = initialLocation;    // configura Location do Floor
46         moving = true;    // começa a se mover em direção ao Button no Floor
47     }
48
49     // configura ouvinte para PersonMoveEvents
50     public void setPersonMoveListener(
51         PersonMoveListener listener )
52     {
53         personMoveListener = listener;
54     }
55
56     // invocado quando Door abriu
57     public void doorOpened( DoorEvent doorEvent )
58     {
59         // configura Person no Floor em que a Door abriu
60         setLocation( doorEvent.getLocation() );
61
62         // interrompe o método sleep da Person no método run
63         // e no método ride de Elevator
64         interrupt();
65     }
66
67     // invocado quando a Door fechou
68     public void doorClosed( DoorEvent doorEvent ) {}
69
70     // configura Location da Person
71     private void setLocation( Location newLocation )
72     {
73         location = newLocation;
74     }
75
76     // obtém Location atual
77     private Location getLocation()
78     {

```

Fig. H.13 A classe `Person` representa a `Person` que anda no `Elevator`. A `Person` opera assincronamente com outros objetos (parte 2 de 6).

```

79     return location;
80 }
81
82 // obtém identificador
83 public int getID()
84 {
85     return ID;
86 }
87
88 // configura se a Person deve se mover
89 public void setMoving( boolean personMoving )
90 {
91     moving = personMoving;
92 }
93
94 // obtém se a Person deve se mover
95 public boolean isMoving()
96 {
97     return moving;
98 }
99
100 // Person anda no Elevator ou espera por ele
101 public void run()
102 {
103     sendPersonMoveEvent( PERSON_CREATED );
104
105     // caminha para o Elevator
106     pauseThread( TIME_TO_WALK );
107     setMoving( false );
108
109     // Person chegou no Button do Floor
110     sendPersonMoveEvent( PERSON_ARRIVED );
111
112     // obtém Door atual no Floor
113     Door currentFloorDoor = location.getDoor();
114
115     // determina se a Door no Floor está aberta
116     try {
117
118         boolean doorOpen = currentFloorDoor.isDoorOpen();
119
120         // se a Door no Floor estiver fechada
121         if ( !doorOpen ) {
122
123             // pressiona Button do Floor
124             sendPersonMoveEvent( PERSON_PRESSING_BUTTON );
125             pauseThread( 1000 );
126
127             // registra-se para evento doorOpen da Door do Floor
128             currentFloorDoor.addDoorListener( this );
129
130             // pressiona Button do Floor para chamar Elevator
131             Button floorButton = getLocation().getButton();
132             floorButton.pressButton( getLocation() );
133
134             // espera que a Door do Floor se abra
135             sleep( TIME_WAITING );
136
137             // cancela registro com a Door do Floor se demorou demais

```

Fig. H.13 A classe `Person` representa a `Person` que anda no `Elevator`. A `Person` opera assincronamente com outros objetos (parte 3 de 6).

```

138         currentFloorDoor.removeDoorListener( this );
139     }
140
141     // se a Door no Floor estiver aberta, anda no Elevator
142     else
143         enterAndRideElevator();
144 }
145
146 // trata de exceção quando interrompida durante espera
147 catch ( InterruptedException interruptedException ) {
148
149     // Person cancela registro para evento doorOpen do Floor
150     currentFloorDoor.removeDoorListener( this );
151
152     // entra e anda no Elevator quando a Door no Floor se abre
153     pauseThread( 1000 );
154     enterAndRideElevator();
155 }
156
157 // esperar que a Door do Elevator se abra leva um segundo
158 pauseThread( 1000 );
159
160 // começa a se afastar do Elevator
161 setMoving( true );
162
163 // Person sai do Elevator
164 sendPersonMoveEvent( PERSON_EXITING_ELEVATOR );
165
166 // afastar-se do elevador leva cinco segundos
167 pauseThread( 2 * TIME_TO_WALK );
168
169 // Person sai da simulação
170 sendPersonMoveEvent( PERSON_EXITED );
171
172 } // fim do método run
173
174 // Person entra no Elevator
175 private void enterAndRideElevator()
176 {
177     // Person entra no Elevator
178     sendPersonMoveEvent( PERSON_ENTERING_ELEVATOR );
179
180     // configura Location da Person para Elevator
181     Floor floorLocation = ( Floor ) getLocation();
182     setLocation(
183         floorLocation.getElevatorShaft().getElevator() );
184
185     // Person leva um segundo para entrar no Elevator
186     pauseThread( 1000 );
187
188     // registra-se para evento doorOpen da Door do Elevator
189     Door elevatorDoor = getLocation().getDoor();
190     elevatorDoor.addDoorListener( this );
191
192     // pressionar o Button do Elevator leva um segundo
193     sendPersonMoveEvent( PERSON_PRESSING_BUTTON );
194     pauseThread( 1000 );
195
196     // obtém Button do Elevator

```

Fig. H.13 A classe **Person** representa a **Person** que anda no **Elevator**. A **Person** opera assincronamente com outros objetos (parte 4 de 6).

```

197     Button elevatorButton = getLocation().getButton();
198
199     // pressiona Button do Elevator
200     elevatorButton.pressButton( location );
201
202     // fechamento da Door leva um segundo
203     pauseThread( 1000 );
204
205     // anda no Elevator
206     Elevator elevator = ( Elevator ) getLocation();
207     elevator.ride();
208
209     // Person terminou de andar no Elevator
210
211     // cancela registro para evento doorOpen do Elevator
212     elevatorDoor.removeDoorListener( this );
213
214 } // fim do método enterAndRideElevator
215
216 // faz thread parar pelo número de milissegundos
217 private void pauseThread( int milliseconds )
218 {
219     try {
220         sleep( milliseconds );
221     }
222
223     // trata exceção se interrompida quando em pausa
224     catch ( InterruptedException interruptedException ) {
225         interruptedException.printStackTrace();
226     }
227 } // fim do método pauseThread
228
229 // envia PersonMoveEvent para listener, dependendo do tipo de evento
230 private void sendPersonMoveEvent( int eventType )
231 {
232     // cria novo evento
233     PersonMoveEvent event =
234         new PersonMoveEvent( this, getLocation(), getID() );
235
236     // envia evento para ouvinte "this", dependendo do eventType
237     switch ( eventType ) {
238
239         // Person foi criada
240         case PERSON_CREATED:
241             personMoveListener.personCreated( event );
242             break;
243
244         // Person chegou no Elevator
245         case PERSON_ARRIVED:
246             personMoveListener.personArrived( event );
247             break;
248
249         // Person entrou no Elevator
250         case PERSON_ENTERING_ELEVATOR:
251             personMoveListener.personEntered( event );
252             break;
253
254         // Person pressionou o objeto Button
255         case PERSON_PRESSING_BUTTON:

```

Fig. H.13 A classe `Person` representa a `Person` que anda no `Elevator`. A `Person` opera assincronamente com outros objetos (parte 5 de 6).

```

256         personMoveListener.personPressedButton( event );
257         break;
258
259         // Person saiu do Elevator
260         case PERSON_EXITING_ELEVATOR:
261             personMoveListener.personDeparted( event );
262             break;
263
264         // Person saiu da simulação
265         case PERSON_EXITED:
266             personMoveListener.personExited( event );
267             break;
268
269         default:
270             break;
271     }
272 } // fim do método sendPersonMoveEvent
273 }

```

Fig. H.13 A classe **Person** representa a **Person** que anda no **Elevator**. A **Person** opera assincronamente com outros objetos (parte 6 de 6).

A classe **Person** é uma subclasse da classe **Thread**. A **Person** executa todas as ações, como caminhar através de **Floors** e andar no **Elevator**, no método **run** (linhas 101 a 172). O método **run** representa o tempo de vida de uma **Person** descrito no diagrama de seqüência da Fig. 15.20. A classe **Person** contém um objeto **PersonMoveListener** (linha 23) para o qual a **Person** envia **PersonMoveEvents**. Em nossa simulação, o **ElevatorModel** usa o método **setPersonMoveListener** (linhas 50 a 54) para registrar a si mesmo como o **PersonMoveListener**. O **ElevatorModel**, ao receber um **PersonMoveEvent**, envia o evento para a **ElevatorView** – portanto, a **ElevatorView** “sabe” quando uma **Person** executou certas ações discutidas em seguida.

Existem diversos tipos de ações que uma **Person** executa durante seu tempo de vida, de modo que existem diversos tipos de **PersonMoveEvents** que a **Person** pode enviar para o **personMoveListener**. As linhas 32 a 37 definem uma série de constantes na qual cada constante representa um tipo de **PersonMoveEvent** único. A **Person** envia eventos para **personMoveListener** quando

- a **Person** foi criada;
- a **Person** chega no **Elevator**;
- a **Person** entra no **Elevator**;
- a **Person** pressiona um **Button** (ou no **Elevator** ou em um **Floor**);
- a **Person** sai do **Elevator**;
- a **Person** sai da simulação.

Quando a **Person** decide enviar um evento para seu **PersonMoveListener**, ela chama o método **private sendPersonMoveEvent** e passa a constante desejada como parâmetro. Este método envia o evento associado com a constante. Por exemplo, a linha 124 chama

```
sendPersonMoveEvent( PERSON_PRESSED_BUTTON );
```

quando a **Person** pressiona um **Button** em um **Floor**. No método **run**, a **Person** caminha para o **Elevator**, depois envia um evento **personArrived** quando chega no **Elevator**. Usamos o diagrama de atividades da Fig.

5.29 para determinar a próxima ação da **Person**. Se a **Door** no **Floor** estiver fechada (linha 121), a **Person** precisa esperar que aquela **Door** se abra. Especificamente, a linha 128 registra a **Person** como um **DoorListener** para aquela **Door**, e as linhas 131 e 132 permitem que a **Person** pressione o **Button** naquele **Floor**. A **Person** espera que a **Door** abra chamando o método **sleep** (linha 135) da superclasse **Thread** de **Person**. Quando a **Door** abre, ela informa à **Person** em **doorOpened** (linhas 57 a 65). A linha 64 do método **doorOpened** interrompe a **thread** da **Person**, que termina o método **sleep** da linha 135. O método **interrupt** dispara uma **InterruptedException** recebida por um bloco **catch** (linhas 147 a 155). A linha 150 deste bloco **catch** cancela o registro da **Person** com a **Door** no **Floor** e a linha 154 chama o método **private enterAndRideElevator** imediatamente.

A linha 178 do método **enterAndRideElevator** envia um evento **personEntered** para **personMoveListener**, indicando que a **Person** está entrando no **Elevator**. Quando a **Person** entra no **Elevator**, a **Location** da **Person** faz referência ao **Elevator** (linhas 181 a 183). Quando a **Person** entrou no **Elevator**, as linhas 189 e 190 registram a **Person** como um **DoorListener** com a **Door** no **Elevator**. As linhas 197 a 200 permitem que a **Person** pressione o **Button** no **Elevator** e envie um evento **PressedButton** para o **personMoveListener**. As linhas 206 e 207 invocam o método **synchronized ride** no **Elevator**, assegurando que outras **Persons** não possam ocupar o **Elevator**. Quando chega o **Elevator**, ele envia um evento **elevatorArrived** para a **Door** no **Elevator**, que abre aquela **Door** e invoca o método **doorOpened** da **Person**. O método **doorOpened**, como mencionado anteriormente, interrompe a **thread** da **Person** – neste caso, o método **interrupt** termina o adormecimento no método **ride** e permite que a **Person** saia do **Elevator** (permitindo que uma **Person** que está esperando pelo **Elevator** entre). O método **enterAndRideElevator** retorna, e as linhas 161 a 170 do método **run** fazem a **Person** sair do **Elevator** e sair da simulação pouco tempo depois.

H.10 Diagramas de componentes revisitados

Na Seção 13.17, apresentamos o diagrama de componentes para a simulação do elevador. Em nossa simulação, cada classe no modelo importa o pacote **event** – mostramos os componentes do pacote **event** na Fig. G.15. A Fig. H.14 apresenta o diagrama de componentes para o pacote **model**. Cada componente no pacote **model** mapeia uma classe do diagrama de classes da Fig. 15.21 – o pacote **model** agrega o pacote **event**.

H.11 Conclusão

Isto conclui a discussão do modelo de nosso estudo de caso. Esperamos que você tenha gostado do processo de projeto de nossa simulação de elevador com a UML, junto com a apresentação dos fundamentos da orientação a objetos e dos tópicos específicos de Java, como tratamento de eventos e *multithreading*. Usando os conceitos discutidos neste estudo de caso, você deve agora ser capaz de enfrentar até mesmo sistemas maiores. Incentivamo-lo a ler o Apêndice I, que implementa a **ElevatorView**, que transforma o **ElevatorModel** que projetamos em um programa vibrante e interativo, com muitos gráficos, muita animação e muito som.

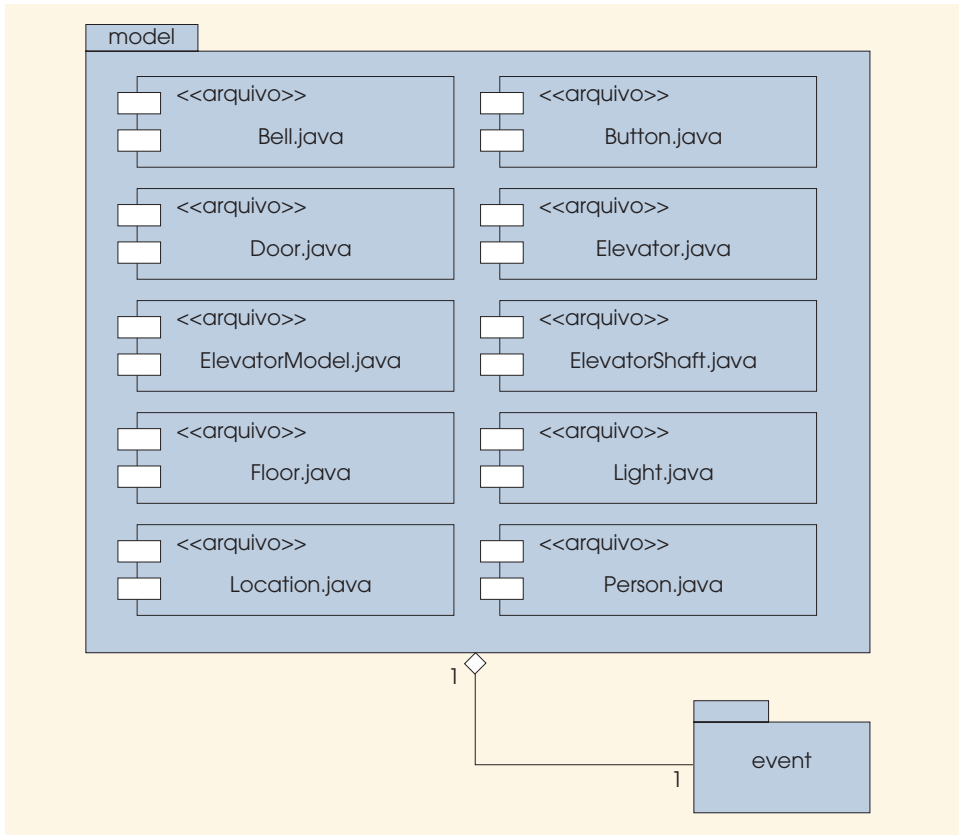


Fig. H.14 Diagrama de componentes para o pacote `model1`.