in the ABM; submodels are often almost completely independent of each other and can be designed and tested independently. The submodels are listed in the schedule, and now we must describe them in full detail. To make the ABM reproducible, we must describe all equations, logical rules, and algorithms that constitute the submodels. But no one will be impressed if all we do is write down these details; we also need to document *why* we formulated the submodels as we did. What literature did we use, what assumptions did we make, where did we get parameter values, and how did we test and calibrate the submodel? Under what conditions is the submodel more or less reliable? In publications, we might present only the description and leave the full justification of the model assumptions to an appendix or separate report.

Let us now see the ODD description of our first real ABM. The example we chose is extremely simple—but it was also the basis of three interesting journal articles. Therefore, we learn immediately that, while some ABMs take months or even years to become productive, quite often we can learn a lot, even about the real world, from very simple ABMs.

3.4 Our First Example: Virtual Corridors of Butterflies

Many animals *disperse*—leave their home location and move long distances for purposes such as mating—at some point in their life. Dispersing animals respond to the landscape, avoiding some features and being attracted to others. These behavioral responses to the landscape can channel their movement into pathways referred to as *corridors*. Corridors are conceived of as linear elements in the landscape that facilitate dispersal; examples include hedgerows, fences, and vegetation along roads. However, our perception of corridors certainly is limited because we can't see the landscape through the animals' eyes. Could it be that some places where we see high numbers of dispersing animals are "virtual corridors"—they do not have especially beneficial features (unlike, for example, a hedgerow, which provides concealment from predators) but instead they emerge from more subtle characteristics of the landscape and from how the animals decide where to go?

To demonstrate the concept of virtual corridors, Pe'er et al. (2005) chose an extremely simple system in which it is easy to observe real individuals: mate-finding by butterflies. In many butterfly species, males and unmated females apply the "hilltopping" strategy: they simply move uphill until they are concentrated on hilltops where they can meet and mate. The model developed by Pe'er et al. (2005) is a good example of how simple a model can be and still capture the essence of a system with regard to a certain question. Because the model is so simple, we skip the heuristic, brainstorming phase of formulating models and jump directly to the model formulation using the ODD protocol.

Purpose

The model was designed to explore questions about virtual corridors. Under what conditions do the interactions of butterfly hilltopping behavior and landscape topography lead to the emergence of virtual corridors, that is, relatively narrow paths along which many butterflies move? How does variability in the butterflies' tendency to move uphill affect the emergence of virtual corridors?

Entities, State Variables, and Scales

The model has two kinds of entities: butterflies and square patches of land. The patches make up a square grid landscape of 150 × 150 patches, and each patch has one state variable: its elevation. Butterflies are characterized only by their location, described as the patch they are

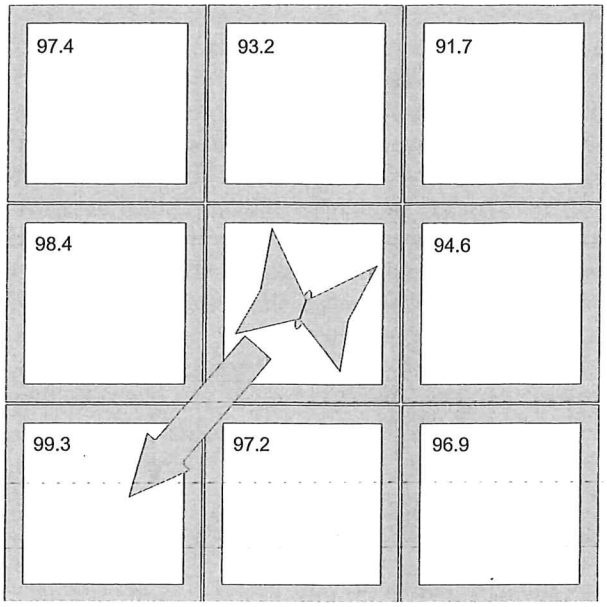

Figure 3.2
The butterfly movement action. A butterfly moves to one of its eight neighboring patches. Patches have an elevation variable, displayed here in their upper left corner. If a random number is less than q, the butterfly moves to the neighbor patch with highest elevation, indicated by the arrow. Otherwise, the butterfly chooses one of the eight neighbor patches randomly.

Input Data

The environment is assumed to be constant, so the model has no input data.

Submodels

The movement submodel defines exactly how butterflies decide whether to move uphill or randomly. First, to "move uphill" is defined specifically as moving to the neighbor patch that has the highest elevation; if two patches have the same elevation, one is chosen randomly. "Move randomly" is defined as moving to one of the neighboring patches, with equal probability of choosing any patch. "Neighbor patches" are the eight patches surrounding the butterfly's current patch. The decision of whether to move uphill or randomly is controlled by the parameter q, which ranges from 0.0 to 1.0 (q is a global variable: all butterflies use the same value). On each time step, each butterfly draws a random number from a uniform distribution between 0.0 and 1.0. If this random number is less than q, the butterfly moves uphill; otherwise, the butterfly moves randomly (figure 3.2).

This formulation in ODD format contains all the information needed to implement the model on a computer. As you learn NetLogo, you will see that the elements of ODD correspond to the typical elements of a NetLogo program. NetLogo has, for example, specific dialogs and primitives for setting the spatial scale and keeping track of time steps, defining and initializing the model's entities, scheduling actions, and observing results. Consequently, in chapter 4 we will be able to use this ODD description of the hilltopping butterfly model as the blueprint for your first real NetLogo program.

3.5 Summary and Conclusions

Describing a model on paper is perhaps the most important part of modeling: very few benefits of modeling can be achieved without it. For ABMs, it is especially important to use standard concepts and formats to describe and design models: these models are complex, so we need

Entities, State Variables, and Scales

Basic entities for ABMs are built into NetLogo: the World of square patches, turtles as mobile agents, and the observer. The state variables of the turtles and patches (and perhaps other types of agents) are defined via the `turtles-own [ ]` and `patches-own [ ]` statements, and the variables characterizing the global environment are defined in the `globals []` statement. In NetLogo, as in ODD, these variables are defined right at the start.

Process Overview and Scheduling

This, exactly, is represented in the `go` procedure. Because a well-designed `go` procedure simply calls other procedures that implement all the submodels, it provides an overview (but not the detailed implementation) of all processes, and specifies their schedule, that is, the sequence in which they are executed each tick.

Design Concepts

These concepts describe the decisions made in designing a model and so do not appear directly in the NetLogo code. However, NetLogo provides many primitives and interface tools to support these concepts; part II of this book explores how design concepts are implemented in NetLogo.

Initialization

This corresponds to an element of every NetLogo program, the `setup` procedure. Pushing the `setup` button should do everything described in the Initialization element of ODD.

Input Data

If the model uses a time series of data to describe the environment, the program can use NetLogo's input primitives to read the data from a file.

Submodels

The submodels of ODD correspond closely but not exactly to procedures in NetLogo. Each of a model's submodels should be coded in a separate NetLogo procedure that is then called from the `go` procedure. (Sometimes, though, it is convenient to break a complex submodel into several smaller procedures.)

These correspondences between ODD and NetLogo make writing a program from a model's ODD formulation easy and straightforward. The correspondence between ODD and NetLogo's design is of course not accidental: both ODD and NetLogo were designed to capture and organize the important characteristics of ABMs.

4.3 Butterfly Hilltopping: From ODD to NetLogo

Now, let us for the first time write a NetLogo program by translating an ODD formulation, for the Butterfly model. The way we are going to do this is hierarchical and step-by-step, with many interim tests. We strongly recommend you always develop programs in this way:

- Program the overall structure of a model first, before starting any of the details. This keeps you from getting lost in the details early. Once the overall structure is in place, add the details one at a time.
- Before adding each new element (a procedure, a variable, an algorithm requiring complex code), conduct some basic tests of the existing code and save the file. This way, you always

proceed from "firm ground": if a problem suddenly arises, it very likely (although not always) was caused by the last little change you made.

First, let us create a new NetLogo program, save it, and include the ODD description of the model on the Information tab.

- Start NetLogo and use File/New to create a new NetLogo program. Use File/Save to save the program under the name "Butterfly-1.nlogo" in an appropriate folder.
- Get the file containing the ODD description of the Butterfly model from the chapter 4 section of the web site for this book.
- Go the Information tab in NetLogo, click the Edit button, and paste in the model description at the top.
- Click the Edit button again.

You now see the ODD description of the Butterfly model at the beginning of the documentation.

Now, let us start programming this model with the second part of its ODD description, the state variables and scales. First, define the model's state variables—the variables that patches, turtles, and the observer own.

- Go to the Procedure tab and insert

```
globals [ ]
patches-own [ ]
turtles-own [ ]
```

- Click the Check button.

There should be no error message, so the code syntax is correct so far. Now, from the ODD description we see that turtles have no state variables other than their location; because NetLogo already has built-in turtle variables for location (`xcor`, `ycor`), we need to define no new turtle variables. But patches have a variable for elevation, which is not a built-in variable, so we must define it.

- In the program, insert elevation as a state variables of patches:

```
patches-own [ elevation ]
```

If you are familiar with other programming languages, you might wonder where we tell NetLogo what type the variable "elevation" is. The answer is: NetLogo figures out the type from the first value assigned to the variable via the `set` primitive.

Now we need to specify the model's spatial extent: how many patches it has.

- Go to the Interface tab, click the Settings button, and change "Location of origin" to Corner and Bottom Left; change the number of columns (`max-pxcor`) and rows (`max-pycor`) to 149. Now we have a world, or landscape, of 150 × 150 patches, with patch 0,0 at the lower left corner. Turn off the two world wrap tick boxes, so that our model world has closed boundaries. Click OK to save your changes.

You probably will see that the World display (the "View") is now extremely large, too big to see all at once. You can fix this:

- Click the Settings button again and change "Patch size" to 3 or so, until the View is a nice size. (You can also change patch size by right-clicking on the View to select it, then dragging one of its corners.)

The next natural thing to do is to program the `setup` procedure, where all entities and state variables are created and initialized. Our guide of course is the Initialization part of the ODD description. Back in the Procedures tab, let us again start by writing a "skeleton."

- At the end of the existing program, insert this:

```
to setup
  ca
  ask patches
  [

  ]
  reset-ticks
end
```

Click the Check button again to make sure the syntax of this code is correct. There is already some code in this `setup` procedure: `ca` to delete everything, which is almost always first in the `setup` procedure, and `reset-ticks`, which is almost always last. The `ask patches` statement will be needed to initialize the patches by giving them all a value for their elevation variable. The code to do so will go within the brackets of this statement.

Assigning elevations to the patches will create a topographical landscape for the butterflies to move in. What should the landscape look like? The ODD description is incomplete: it simply says we start with a simple artificial topography. We obviously need some hills because the model is about how butterflies find hilltops. We could represent a real landscape (and we will, in the next chapter), but to start it is a good idea to create scenarios so simple that we can easily predict what should happen. Creating just one hill would probably not be interesting enough, but two hills will do.

- Add the following code to the `ask patches` command:

```
ask patches
[
  let elev1 100 - distancexy 30 30
  let elev2 50 - distancexy 120 100

  ifelse elev1 > elev2
    [set elevation elev1]
    [set elevation elev2]

  set pcolor scale-color green elevation 0 100
]
```

The idea is that there are two conical hills, with peaks at locations (x,y-coordinates) 30,30 and 120,100. The first hill has an elevation of 100 units (let's assume these elevations are in meters) and the second an elevation of 50 meters. First, we calculate two elevations (*elev1* and *elev2*) by assuming the patch's elevation decreases by one meter for every unit of horizontal distance between the patch and the hill's peak (remind yourself what the primitive `distancexy` does by putting your cursor over it and hitting the F1 key). Then, for each patch we assume the elevation is the greater of two potential elevations, *elev1* and *elev2*: using the `ifelse` primitive, each patch's elevation is set to the higher of the two elevations.

Finally, patch elevation is displayed on the View by setting patch color `pcolor` to a shade of the color `green` that is shaded by patch elevation over the range 0 and 100 (look up the extremely useful primitive `scale-color`).

Remember that the `ask patches` command establishes a patch context: all the code inside its brackets must be executable by patches. Hence, this code uses the state variable `elevation` that we defined for patches, and the built-in patch variable `pcolor`. (We could *not* have used the built-in variable `color` because `color` is a turtle, not a patch, variable.)

Now, following our philosophy of testing everything as early as possible, we want to see this model landscape to make sure it looks right. To do so, we need a way to execute our new `setup` procedure.

- On the Interface, press the Button button, select "Button," and place the cursor on the Interface left of the View window. Click the mouse button.

- A new button appears on the Interface and a dialog for its settings opens. In the Commands field, type "setup" and click OK.

Now we have linked this button to the `setup` procedure that we just programmed. If you press the button, the `setup` procedure is executed and the Interface should look like figure 4.1.

There is a landscape with two hills! Congratulations! (If your View does not look like this, figure out why by carefully going back through *all* the instructions. Remember that any mistakes you fix will not go away until you hit the `setup` button again.)

> **NetLogo brainteaser**
>
> When you click the new `setup` button, why does NetLogo seem to color the patches in spots all over the View, instead of simply starting at the top and working down? (You may have to turn the speed slider down to see this clearly.) Is this a fancy graphics rendering technique used by NetLogo? (Hint: start reading the Programming Guide section on Agentsets very carefully.)

Now we need to create some agents or, to use NetLogo terminology, turtles. We do this by using the `create-turtles` primitive (abbreviated `crt`). To create one turtle, we use `crt 1 [ ]`. To better see the turtle, we might wish to increase the size of its symbol, and we also want to specify its initial position.

- Enter this code in the `setup` procedure, after the `ask patches` statement:

```
crt 1
[
```

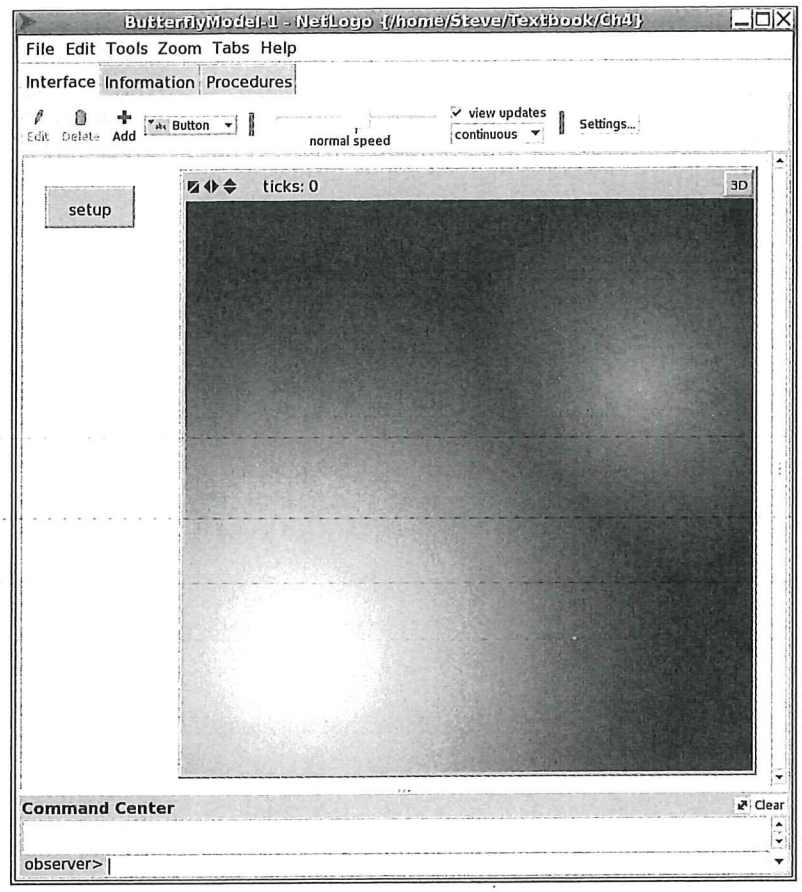

Figure 4.1
The Butterfly model's artificial landscape.

```
  set size 2
  setxy 85 95
]
```

- Click the Check button to check syntax, then press the `setup` button on the Interface.

Now we have initialized the World (patches) and agents (turtles) sufficiently for a first minimal version of the model.

The next thing we need to program is the model's schedule, the `go` procedure. The ODD description tells us that this model's schedule includes only one action, performed by the turtles: movement. This is easy to implement.

- Add the following procedure:

```
to go
  ask turtles [move]
end
```

- At the end of the program, add the skeleton of the procedure `move`:

```
to move

end
```

Now you should be able to check syntax successfully.

There is one important piece of `go` still missing. This is where, in NetLogo, we control a model's "temporal extent": the number of time steps it executes before stopping. In the ODD description of model state variables and scales, we see that Butterfly model simulations last for 1000 time steps. In chapter 2 we learned how to use the built-in tick counter via the primitives `reset-ticks`, `tick`, and `ticks`. Read the documentation for the primitive `stop` and modify the `go` procedure to stop the model after 1000 ticks:

```
to go
  ask turtles [move]
  tick
  if ticks >= 1000 [stop]
end
```

- Add a `go` button to the Interface. Do this just as you added the `setup` button, except: enter "go" as the command that the button executes, and activate the "Forever" tick box so hitting the button makes the observer repeat the `go` procedure over and over.

> **Programming note:** Modifying Interface elements
>
> To move, resize, or edit elements on the Interface (buttons, sliders, plots, etc.), you must first select them. Right-click the element and chose "Select" or "Edit." You can also select an element by dragging the mouse from next to the element to over it. Deselect an element by simply clicking somewhere else on the Interface.

If you click `go`, nothing happens except that the tick counter goes up to 1000—NetLogo is executing our `go` procedure 1000 times but the `move` procedure that it calls is still just a skeleton. Nevertheless, we verified that the overall structure of our program is still correct. And we added a technique very often used in the `go` procedure: using the `tick` primitive and `ticks` variable to keep track of simulation time and make a model stop when we want it to.

(Have you been saving your new NetLogo file frequently?)

Now we can step ahead again by implementing the `move` procedure. Look up the ODD element "Submodels" on the Information tab, where you find the details of how butterflies move. You will see that a butterfly should move, with a probability q, to the neighbor cell with the highest elevation. Otherwise (i.e., with probability $1 - q$) it moves to a randomly chosen neighbor cell. The butterflies keep following these rules even when they have arrived at a local hilltop.

- To implement this movement process, add this code to the `move` procedure:

```
to move
  ifelse random-float 1 < q
```

```
    [ uphill elevation ]
    [ move-to one-of neighbors ]
end
```

The `ifelse` statement should look familiar: we used a similar statement in the `search` procedure of the Mushroom Hunt model of chapter 2. If q has a value of 0.2, then in about 20% of cases the random number created by `random-float` will be smaller than q and the `ifelse` condition true; in the other 80% of cases the condition will be false. Consequently, the statement in the first pair of brackets (`uphill elevation`) is carried out with probability q and the alternative statement (`move-to one-of neighbors`) is done with probability $1-q$.

The primitive `uphill` is typical of NetLogo: following a gradient of a certain variable (here, *elevation*) is a task required in many ABMs, so `uphill` has been provided as a convenient shortcut. Having such primitives makes the code more compact and easier to understand, but it also makes the list of NetLogo primitives look quite scary at first.

The remaining two primitives in the `move` procedure (`move-to` and `one-of`) are self-explanatory, but it would be smart to look them up so you understand clearly what they do.

■ Try to go to the Interface and click `go`.

NetLogo stops us from leaving the Procedures tab with an error message telling us that the variable `q` is unknown to the computer. (If you click on the Interface tab again, NetLogo relents and lets you go there, but turns the Procedures tab label an angry red to remind you that things are not all OK there.)

Because q is a parameter that all turtles will need to know, we define it as a global variable by modifying the `globals` statement at the top of the code:

```
globals [ q ]
```

But defining the variable q is not enough: we also must give it a value, so we initialize it in the `setup` procedure. We can add this line at the end of `setup`, just before `end`:

```
set q 0.4
```

The turtles will now move deterministically to the highest neighbor patch with a probability of 0.4, and to a randomly chosen neighbor patch with a probability of 0.6.

Now, the great moment comes: you can run and check your first complete ABM!

■ Go to the Interface and click `setup` and then `go`.

Yes, the turtle does what we wanted it to do: it finds a hilltop (most often, the higher hill, but sometimes the smaller one), and its way is not straight but somewhat erratic.

Just as with the Mushroom Hunt model, it would help to put the turtle's "pen" down so we can see its entire path.

■ In the `crt` block of the `setup` procedure (the code statements inside the brackets after `crt`), include a new line at the end with this command: `pen-down`.

You can now start playing with the model and its program, for example by modifying the model's only parameter, q. Just edit the `setup` procedure to change the value of q (or the initial location of the turtle, or the hill locations), then pop back to the Interface tab and hit

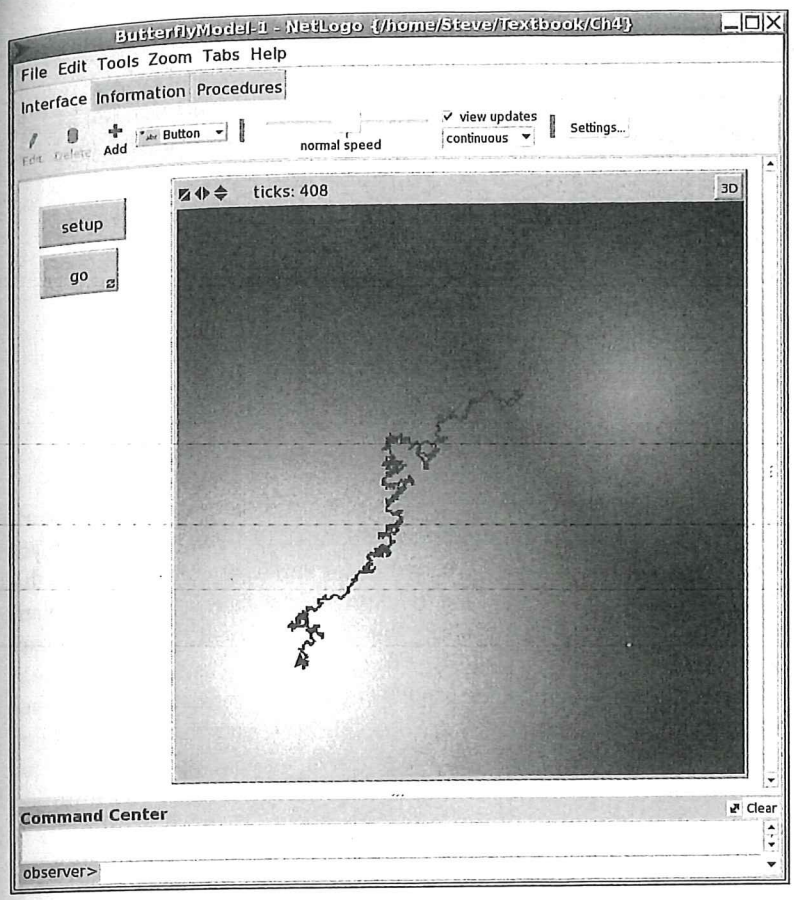

Figure 4.2
Interface of the Butterfly model's first version, with $q = 0.2$.

the `setup` and `go` buttons. A typical model run for $q = 0.2$, so butterflies make 80% of their movement decisions randomly, is shown in figure 4.2.

4.4 Comments and the Full Program

We have now implemented the core of the Butterfly model, but four important things are still missing: comments, observations, a realistic landscape, and analysis of the model. We will address comments here but postpone the other three things until chapter 5.

Comments are any text following a semicolon (`;`) on the same line in the code; such text is ignored by NetLogo and instead is for people. Comments are needed to make code easier for others to understand, but they are also very useful to ourselves: after a few days or weeks go by, you might not remember why you wrote some part of your program as you did instead of in some other way. (This surprisingly common problem usually happens where the code was hardest to write and easiest to mess up.) Putting a comment at the start of each procedure saying whether the procedure is in turtle, patch, or observer context helps you write the procedures by making you think about their context, and it makes revisions easier.

- As a good example comment, change the first line of your `move` procedure to this, so you always remember that the code in this procedure is in turtle context: