6

AN ALGORITHM FOR THE REST OF US

MOVE A DISK BY MOVING AN OBJECT

> ***What is one and one and one and one and one and one and one and one and one and one?***
>
> THE WHITE QUEEN TO ALICE in *Through the Looking Glass,* Chapter 9

Lewis Carroll's joke would not be funny to a computer. A computer would simply add up the "ones" and get the right answer, while a person loses track after the fourth or fifth one. The recursive solution to the Tower of Hanoi is just as bad as the White Queen's math problem. Our programs in the previous chapters have determined what move to make by permuting pole numbers that were stored "on the stack." Each recursive call on moveTower:from:to:using: uses the values of the input parameters from the previous call to choose a move. This is fine for computers, but you have probably noticed that we humans have a "stack" that forgets when too many things are pushed onto it. No human would ever solve the tower puzzle the same way the recursive algorithm does. Let's switch the program over to an algorithm that

any person could use to solve the Tower of Hanoi. Besides using a more intuitive algorithm, we hope in this section to demonstrate what objects are really for, and to show you a program that is simpler in Smalltalk than in Pascal, C, or LISP.

Here are some simple rules that any person can use to solve the Tower of Hanoi:

(1) Don't make illegal moves (always put a small disk on top of a bigger disk).
(2) Don't move the disk you just moved (the last move should not be undone).
(3) If there are two legal moves, choose the one that does not put a disk back on the pole it came from the last time it moved (make forward progress).

This is much more like it! We won't offer a proof that these rules represent a mathematically airtight solution, but they do work. The three rules are heuristics based on practical experience. We are going to write a program that implements these three rules. Please don't confuse this with "rule-based programming." Smalltalk is not a system that takes rules as source code, as a "production system language" does.

The first two rules are easy. The third rule requires you to remember which pole each disk came from the last time it moved. That's a little bit of a strain, but we do have a computer here to help us. Clearly, each disk should have a variable to store the pole it last moved from. In addition, the new algorithm needs to remember what disk moved last, what disk we are considering moving next, and what disk we are thinking of as a possible destination.

In spite of the fact that we are completely changing the algorithm, the data structures stay almost exactly the same as before. We will divide the problem up into objects exactly as before, but we will use two new classes. Class TowerByRules represents the whole game and holds all the game-wide information. Each disk is an instance of class HanoiDiskRules. We would like our new program to be animated, and since the knowledge of animation in class HanoiDisk has nothing to do with the algorithm for deciding what disk to move, we will use most of HanoiDisk unchanged. In the last chapter we made class Animated-TowerOfHanoi be a subclass of TowerOf Hanoi. It inherited instance variables and messages. In the same way, TowerByRules will be a subclass of AnimatedTowerOfHanoi and HanoiDiskRules will be a subclass of HanoiDisk.

We are building up quite a large inheritance chain, so let's look at it explicitly.

```
Object ()
  TowerOfHanoi ('stacks')
    AnimatedTowerOfHanoi ('howMany' 'mockDisks')
      TowerByRules ('oldDisk' 'currentDisk' 'destinationDisk')
```

Here we see TowerByRules with its newly added instance variables. It inherits behavior and instance variables from AnimatedTowerOfHanoi, which in turn inherits from TowerOfHanoi, which inherits from Object.

```
Object ()
  HanoiDisk ('width' 'pole' 'rectangle' 'name')
    HanoiDiskRules ('previousPole')
```

HanoiDiskRules is a subclass of HanoiDisk and adds a single new instance variable. These two subclass tables can be seen in tlie browser. Alter we enter the two new classes in the next section, yon can select a class in area B, and then choose **heirarchy** Iroin tlie middle-button iiieiiii. Here is tlie definition of'class TowerByRules:

```
AnimatedTowerOfHanoi subclass: #TowerByRules
  instanceVariableNames: 'oldDisk currentDisk destinationDisk'
  classVariableNames:''
  poolDictionaries:''
  category: 'Kernel-Objects''
```

Here is tlie comment tor TowerByRules:

```
An object of this class represents the game. It holds an array of stacks that
hold disks. It also keeps track of which disk just moved and which disk
should move next. The new instance variables are
  oldDisk-the disk that was moved last time
  currentDisk-we are considering moving this disk
  destinationDisk-and putting it on top of this disk
```

Tlie instance variables stacks, howMany, and mockDisks are tlie same as before. At the beginning ot a ino\ e, we know oldDisk, the disk lliat moved last time. If we can pick a currentDisk and destinationDisk tliat satisly (lie tliree rules, the rest is easv. This suggests a "main loop ' to find tlie next move, and we will put it in tlie hanoi method.

```
hanoi
   "Ask the user how many disks, set up the game, and move disks until
      we are done."
   howMany <- (FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>') asNumber.
   self setUpDisks.       "create the disks and stacks"

   "Iterate until all disks are on one tower again."
   ["decide which to move and also set destinationDisk"
   currentDisk <- self decide.
   "remove the disk and put it on the new pole"
   (stacks at: currentDisk pole) removeFirst.
   (stacks at: destinationDisk pole) addFirst: currentDisk.
   "tell the disk where it is now"
   currentDisk moveUpon: destinationDisk.
   oldDisk <- currentDisk.       "get ready for the next move"
   self allOnOneTower] whileFalse.       "test if done"

    "(TowerByRules new) hanoi"
```

The entire second half of the method is a "while loop." The statement

[*statements, a boolean expression*] whileFalse.

executes the statements repeatedly until the boolean expression is true. (In Smalltalk's terminology, the code inside the square brackets is a block of unevaluated code. The block is an object and it is sent the message whileFalse. It executes itself repeatedly until the last expression in the block has a value of true.)

Let's consider what needs to be done to make a move in the middle of the game. Starting with the disk that just finished moving, decide which other disk to move next and where to put it. Move the disk by transferring it from one stack to another and altering the disk's internal state (its location). When all disks are on one pole, we are done; otherwise start again on a new move.

Now let's look at the code in more detail. Inside the whileFalse loop in hanoi, the first statement

```
currentDisk <- self decide.
```

does all the work of determining which disk to move, and, as a side effect, sets destinationDisk to be the disk onto which currentDisk will move. The method decide uses the three rules to choose the next disk to move. To do this, decide ignores the disk that just finished moving and considers the disks on top of the other two poles. It tests to see if

they have any legal moves. If one does, decide chooses its best move (in case it can move to two places) by invoking Rule 3. It sets destinationDisk to be the disk that the moving disk will land on top of, and returns the object that represents the moving disk.

```
decide
  "use the last disk moved (oldDisk) to find a new disk to move
    (currentDisk) and a disk to put it on top of (destinationDisk)"
  self topsOtherThan: oldDisk do: [:movingDisk |
    movingDisk hasLegalMove ifTrue:
      ["remember the disk upon which to move"
      destinationDisk <- movingDisk bestMove.
       f  movingDisk       "return the disk that moves"]].
```

The entire method is one long statement: a rather odd control structure named topsOtherThan:do:. Most computer languages give you a few standard control structures such as if-then-else, tor-loops, and repeat-until; and that's all you get. In Smalltalk, you can write your own control methods and use them to direct the flow of control in programs. The method topsOtherThan:do: accepts two arguments. The first is oldDisk, the disk that will not be moving. The second is a block of code enclosed in square brackets. The block is of the form

[:movingDisk j *statements].*

The block of code is not evaluated when the message topsOtherThan:do: is sent, but is an object which contains unevaluated code and is handed as an argument to the method topsOtherThan:do:. The block contains a local variable, movingDisk. The colon before movingDisk means that movingDisk's value will be assigned at the time the block runs. As you might expect, the block is run once for each of the disks that is a candidate to move, and the object that represents that disk is assigned to movingDisk.* The message topsOtherThan:do: is sent to self, so when we write the code for topsOtherThan:do:, we will put it in class TowerByRules, where the decide method is located. The statement inside the block is

```
movingDisk hasLegalMove ifTrue:
  ["remember the disk upon which to move"
  destinationDisk «- movingDisk bestMove.
   f  movingDisk   "return the disk that moves"]
```

* *LISP* programmers note that a block is like *(LAMBDA(mocingDisk) statements).* Pascal programmers note that it is like a local procedure of one argument.

The method hasLegalMove returns true ifmovingDisk fits on top of any other pole. If so, destination Disk is assigned the result sending the message bestMove to movingDisk. It answers with one of the two disks it could move upon. destinationDisk is an instance variable of TowerByRules, so it is available inside the hanoi method when this method returns (we'd like to return multiple values here, both destinationDisk and movingDisk, but Smalltalk does not provide for easy returning of multiple values). The expression f movingDisk forces control to leave the loop, terminates this message, and returns the value that worked for movingDisk.

The messages decide and hanoi were both in class TowerByRules. Since hasLegalMove is a message that is sent to an individual disk, the method is in class HanoiDiskRules. Lets look at class HanoiDiskRules. It is a subclass of HanoiDisk.

```
HanoiDisk subclass: #HanoiDiskRules
  instanceVariableNames:' previousPole'
  classVariableNames:''
  poolDictionaries: ''
  category: 'Kernel-Objects'
```

Only the new instance variable, previousPole, is mentioned. Each instance of HanoiDiskRules also has the variables name, width, pole, and rectangle that it inherits from HanoiDisk. Here is the comment:

```
previousPole-number of the pole this disk was on previously
```

And the code for hasLegalMove is

```
hasLegalMove
  "Do either of the other two poles have a top disk large enough for
    this disk to rest on?"
  TheTowers polesOtherThan: self do: [:targetDisk |
    "when a pole has no disk, targetDisk is a mock disk with infinite width"
    width < targetDisk width ifTrue: [ f true]].
  f false
```

Once again TheTowers is the object that represents the whole game, in this case an instance of TowerByRules. It is being sent the message polesOtherThan:do:, which looks suspiciously like the message topsOtherThan:do: in decide. (Notice that topsOtherThan:do: was sent to self. Inside decide, self was a TowerByRules, but here self is a disk, so we must send it to TheTowers.)

Looking again at the code for hasLegalMove, we see a new control structure. polesOtherThan:do: is just like topsOtherThan:do: except for the way it treats poles that have no disks on them. In decide we looked for disks that would move, so an empty pole was of no interest. The local variable targetDisk can be either a real disk or something that stands for an empty pole. As before we'll use a mock disk to stand for the top disk on an empty pole. The message polesOtherThan:do: executes the block that it gets as its second argument once for each of the other two poles. If the pole has a stack of disks, the top disk is assigned to the block's local variable. If the pole is empty, the mock disk for that pole is assigned to targetDisk.

```
width < targetDisk width ifTrue: [ f true]
```

Inside the loop, targetDisk is the top disk on a pole other than self's (movingDisk's) pole, and we test to see if self can legally move there. The variable width is self's own width. The second mention of width is a message to targetDisk, asking it to return its width. If self's width is less, then return true. If both other poles have been tested by polesOtherThan:do: and no move was legal, the next statement, f false, informs the caller that self cannot make a legal move.

Since we mentioned the message width, let's define it.

```
width
  "Return the size of this disk"
  f width
```

Each instance of HanoiDiskRules responds to the message width by returning the value of its variable called width. It is common practice to define a message with a name that is the same as an instance variable name. By informal convention, the message width returns the value of the variable width.

Now we come to the question of computing the best move if a disk has two possible moves. Rule 3 says that a disk should not move back to the pole from which it last moved. Each disk has a variable, previousPole, that holds the number of its previous pole. In the method decide, we wrote

```
destinationDisk <— movingDisk bestMove.
```

Like hasLegalMove, bestMove is a message sent to movingDisk, and like hasLegalMove it must look at each of the possible places that movingDisk can go.

```
bestMove    | secondBest |
  "If self can move two places, which is best? Return the top disk of
    the pole that this disk has not been on recently."
  TheTowers polesOtherThan: self do: [:targetDisk |
    width < targetDisk width ifTrue:
      [secondBest <- targetDisk.
      targetDisk pole = previousPole ifFaise: [ f targetDisk]]].
  t secondBest "as a last resort, return a pole it was on recently"
```

For each other pole, see if self will fit on the top disk. If so, remember this disk in the local variable secondBest (we may need it). If the new pole is not this disk's previous pole, then return targetDisk. If, instead, the only legal move puts the disk back to its previous pole, return that move (the value of secondBest). The code for the message pole is inherited from HanoiDisk.

Now that we've covered everything in the decide method, let's see how the two custom control structures are defined in TowerBy-Rules.

```
topsOtherThan: thisDisk do: aBlock
  "Evaluate the block of code using the top disk on each of the other
    two poles. If a pole is empty, ignore it. This is for actual disks."
  1 to: 3 do: [:aPole |
    "If the pole does not have thisDisk and is not empty, then
      execute aBlock"
    (aPole ~= thisDisk pole) & ((stacks at: aPole) isEmpty not) ifTrue:
      {aBlock value: (stacks at: aPole) first "execute the block"]]
```

For each of the three poles, if it is not thisDisk's pole and if the stack on the pole is not empty, we want to run the block that the caller supplied. We want to send in the top disk on this pole as the argument to the block.

```
aBlock value: (stacks at: aPole) first.
```

The message first asks this OrderedCollection for its top element. The message value: tells a block of unevaluated code to execute and to accept the object that follows (i.e., the top disk) as the value of the block's local variable.*

Some unfamiliar messages are being sent. A tilde followed by an equal sign ~= is the selector for "not equal." It returns true or false,

* LZSP programmers note that evaluating a block is like *(APPLY aBlock (LIST arg))*. Pascal programmers note that it is similar to *aBbick(arg)* where *aBlock is* a parametric procedure.

and these booleans understand the message & to be logical "and." Every OrderedCollection (stack) understands the message isEmpty. It also returns a boolean that understands the message not.

So that you can see how topsOtherThan:do: is used, here again is the code for decide:

```
decide
  "use the last disk moved (oldDisk) to find a new disk to move
    (currentDisk) and a disk to put it on top of (destinationDisk)"
  self topsOtherThan: oldDisk do: [:movingDisk |
    movingDisk hasLegalMove ifTrue:
      ["remember the disk upon which to move"
      destinationDisk <- movingDisk bestMove.
       f  movingDisk   "return the disk that moves"]].
```

When topsOtherThan:do: is running, aBlock is the entire piece of code in square brackets, and the result of the expression (stacks at: aPole) first is stored into movingDisk just before the block executes. The "return" statement inside the block, f movingDisk, causes the code in the block to terminate. And it causes the methods topsOtherThan:do: and decide to terminate. In decide the f is inside a block, and the block is not evaluated in the original method, but two methods below. When the return is executed, it exits all procedures until it encounters the one in which the f symbol actually appears (Chapter 3 of the Blue Book).

The code for polesOtherThan:do: is very similar to topsOther-Than:do:.

```
polesOtherThan: thisDisk do: aBlock
  "Evaluate the block of code using the top disk on each of the other
    two poles. If a pole is empty, use the mockDisk for that pole."
  1 to: 3 do: [:aPo!e |
    "Want a pole other than the pole of thisDisk"
    (aPole ~= thisDisk pole) ifTrue:
      [(stacks at: aPole) isEmpty ifTrue:
          ["If the pole is empty, use a mock disk"
          aBlock value: (mockDisks at: aPole) "execute the block"]
        if False:
          ["else use the top disk"
          aBlock value: (stacks at: aPole) first "execute the block"]]]
```

If a pole has no disks, the value we supply to the block is different. We index the proper mock disk from the array, mockDisks, and supply it as the argument to the block.

We are almost done with this example. Please remember the Japanese proverb: "Patience is bitter, but its fruit is sweet." In the hanoi method, after we decide which disk to move and transfer it from one pole to another, we must give the disk that moved a chance to update its internal state. We send the currentDisk the message moveUpon:. Besides updating the disk, moveUpon: controls the animation. The only thing we want to do differently than HanoiDisk did is to set previousPole. We will write a method called moveUpon: in HanoiDiskRules, but we don't really want to override the old version. We want to call the old version, and then execute another statement. In Smalltalk, the reserved word super allows one to call a message in a higher class despite its being redefined in the subclass.

```
moveUpon: destination
   "This disk just moved. Record the new pole and tell the user."
   previousPole <- pole.
   "Run the version of moveUpon: defined in class HanoiDisk."
   super moveUpon: destination.
```

At the end of the main loop in hanoi, we test if the game is finished.

```
allOnOneTower
   "Return true if all of the disks are on one tower."
   stacks do: [:each | each size = howMany ifTrue: [ f true]].
    f false
```

All kinds of collections in Smalltalk understand the message do:. For each element in the array stacks, supply that stack as an argument to the block. The code in the block tests whether the stack has all of the disks on it. When one stack has all the disks, we are done.

The rest of the code is concerned with initializing the data structures. In class TowerByRules we modify setUpDisks sightly:

```
setUpDisks      | disk displayBox |
   "Create the disks and set up the poles."
   "Make self global for debugging. Later, the user can examine the data
      structures by selecting Hanoi inspect and choosing do it."
   Smalltalk at: #Hanoi put: self.
   "Tell all disks what game they are in and set disk thickness and gap"
   HanoiDiskRules new whichTowers: self.
   displayBox <- 20@100 corner: 380@320.
   Display white: displayBox.
   Display border: displayBox width: 2.
```

```
"The poles are an array of three stacks. Each stack is an
   OrderedCollection."
stacks <- (Array new: 3) coilect: [:each | OrderedCollection new].
howManyto: 1 by: -1 do: [:size |
   disk <- HanoiDiskRules new width: size pole: 1. "Create a disk"
   (stacks at: 1) addFirst: disk.       "Push it onto a stack"
   disk invert       "show on the screen"].

"When a poie has no disk on it, one of these mock disks acts as a bottom
   disk. A moving disk will ask a mock disk its width and pole number"
mockDisks <- Array new: 3.
1 to: 3 do: [:index |
   mockDisks at: index put: (HanoiDiskRules new width: 1000 pole: index)].
"On the first move, look for another disk (a real one) to move."
oldDisk <- mockDisks at: 3.
```

This method is identical to the one in class AnimatedTowerOf-Hanoi, except for the name of the class of the disks (HanoiDiskRules) and an extra statement at the beginning and one at the end. The statement

```
Smalltalk at: #Hanoi put: self.
```

creates a global variable Hanoi in Smalltalk, the dictionary of global variables. This does not help our program, but it allows us to get our hands on the object that represents the game. After the program has finished, or if it stops in the middle with an error, we can see the state of the game by executing Hanoi inspect. An inspector window allows you to see inside an object and change the values of its instance variables. Inspectors are covered in Chapter 8 of the User's Guide.

At the start of the Tower of Hanoi program, oldDisk must be a disk that is not on stack number one. This is because the program looks for a disk to move that is on top of a pole other than oldDisk's pole. However, all real disks *are* on stack number one! Assigning oldDisk a mock disk that thinks it is on pole number three does the trick.

```
oldDisk«- mockDisks at: 3.
```

In class HanoiDiskRules, we need fro define the message that initializes a disk (a disk in the game that is, not a disk in a disk drive).

```
width: size pole: whichPole
   "Invoke widtirpole: in the superclass"
   super width: size pole: whichPole.
   previousPole «-1.
```

We use super to do exactly what HanoiDisk would have done, and then set previousPole to 1.

DEFINING THE CLASSES HanoiDiskRules AND TowerByRules

That's not writing, that's typing!
TRUMAN CAPOTE COMMENTING ON JACK KEBOUAC'S WORK

We are now ready to enter the code we discussed in the previous section. Installing these two classes is exactly like installing the two classes in the last chapter. Here is the definition of HanoiDiskRules:

```
HanoiDisk subclass: #HanoiDiskRules
   instanceVariableNames:' previousPole'
   classVariableNames: ''
   poolDictionaries:''
   category: 'Kernel-Objects'
```

The class comment is simple:

```
   previousPole-number of the pole this disk was on previously
```

The protocols for HanoiDiskRules are

```
('access')
('moving')
```

In the **access** protocol:

```
width
   "Return the size of this disk"
    T width


width: size pole: whichPole
   "Invoke width:pole: in the superclass"
   super width: size pole: whichPole.
   previousPole <-1.
```

In the protocol **moving:**

```
bestMove    | secondBest |
  "If self can move two places, which is best? Return the top disk of
    the pole that this disk has not been on recently."
  TheTowers polesOtherThan: self do: [:targetDisk |
    width < targetDisk width ifTrue:
      [secondBest <- targetDisk.
      targetDisk pole = previousPole ifFalse: [ f  targetDisk]]].
  f secondBest "as a last resort, return a pole it was on recently"
```

Confirm that polesOtherThan:do: is a new message.

```
hasLegalMove
  "Do either of the other two poies have a top disk large enough for
    this disk to rest on?"
  TheTowers polesOtherThan: self do: [:targetDisk |
    "when a pole has no disk, targetDisk is a mock disk with infinite width"
    width < targetDisk width ifTrue: [ f  true]].
  f  false


moveUpon: destination
  "This disk just moved. Record the new pole and tell the user."
  previousPole«- pole.
  "Run the version of moveUpon: defined in class HanoiDisk."
  super moveUpon: destination.
```

Now define the other new class and install its comment and protocols.

```
AnimatedTowerOfHanoi subclass: #TowerByRules
  instanceVariableNames: "oldDisk currentDisk destinationDisk'
  classVariableNames: "
  poolDictionaries:"
  category: 'Kernel-Objects'
```

And its comment:

An object of this class represents the game. It holds an array of stacks that hold disks. It also keeps track of which disk just moved and which disk should move next. The new instance variables are
  oldDisk-the disk that was moved last time
  currentDisk-we are considering moving this disk
  destinationDisk-and putting it on top of this disk

Under **protocols,** let's divide the messages into two groups:

```
('initialize')
('moves')
```

There are two messages in the **initialize** protocol. Be sure to select **initialize** in area C before typing in a method.

```
hanoi
  "Ask the user how many disks, set up the game, and move disks until
    we are done."
  howMany <— (FillInTheBlank request: 'Please type the number of
disks in the tower, and <cr>') asNumber.
  self setUpDisks.      "create the disks and stacks"
  "Iterate until all disks are on one tower again."
  ["decide which to move and also set destinationDisk"
  currentDisk <- self decide.
  "remove the disk and put it on the new pole"
  (stacks at: currentDisk pole) removeFirst.
  (stacks at: destinationDisk pole) addFirst: currentDisk.
  "tell the disk where it is now"
  currentDisk moveUpon: destinationDisk.
  oldDisk <- currentDisk.      "get ready for the next move"
  self allOnOneTower] whileFalse.      "test if done"

  " (TowerByRules new) hanoi"
```

Copy this method from AnimatedTowerOfHanoi and make the underlined changes:

```
setUpDisks      | disk displayBox |
  "Create the disks and set up the poles."
  "Make self global for debugging. Later, the user can examine the data
    structures by selecting Hanoi inspect and choosing do it."
  Smalltalk at: #Hanoi put: self.
  "Tell all disks what game they are in and set disk thickness and gap"
  HanoiDiskRules new whichTowers: self.
  displayBox <- 20@100 comer: 380@320.
  Display white: displayBox.
  Display border: displayBox width: 2.
  "The poles are an array of three stacks. Each stack is an
    OrderedCollection."
  stacks *- (Array new: 3) collect: [:each | OrderedCollection new].
  howMany to: 1 by: -1 do: [:size |
    disk <- HanoiDiskRules new width: size pole: 1. "Create a disk"
    (stacks at: 1) addFirst: disk.      "Push it onto a stack"
    disk invert      "show on the screen"].
```

```
"When a pole has no disk on it, one of these mock disks acts as a bottom
  disk. A moving disk will ask a mock disk its width and pole number"
mockDisks <— Array new: 3.
1 to: 3 do: [:index |
  mockDisks at: index put: (HanoiDiskRules new width: 1000 pole: index)].
"On the first move, look for another disk (a real one) to move."
oldDisk <— mockDisks at: 3.
```

Select protocol **moves** and enter four methods.

```
allOnOneTower
  "Return true if all of the disks are on one tower."
  stacks do: [:each [ each size = howMany ifTrue: [ f  true]].
   f  false
```

```
decide
  "use the last disk moved (oldDisk) to find a new disk to move
    (currentDisk) and a disk to put it on top of (destinationDisk)"
  self topsOtherThan: oldDisk do: [:movingDisk |
    movingDisk hasLegalMove ifTrue:
      ["remember the disk upon which to move"
      destinationDisk <- movingDisk bestMove.
       f  movingDisk   "return the disk that moves"]].
```

Confirm that topsOtherThan:do: is a new message.

```
polesOtherThan: thisDisk do: aBlock
  "Evaluate the block of code using the top disk on each of the other
    two poles. If a pole is empty, use the mockDisk for that pole."
  1 to: 3 do: [:aPole [
    "Want a pole other than the pole of thisDisk"
    (aPole ~= thisDisk pole) ifTrue:
      [(stacks at: aPole) isEmpty ifTrue:
          ["If the pole is empty, use a mock disk"
          aBlock value: (mockDisks at: aPole) "execute the block"]
        ifFalse:
          ["else use the top disk"
          aBlock value: (stacks at: aPole) first "execute the block"]]]
```

```
topsOtherThan: thisDisk do: aBlock
  "Evaluate the block of code using the top disk on each of the other
    two poles. If a pole is empty, ignore it. This is for actual disks."
  1 to: 3 do: [:aPole |
    "If the pole does not have thisDisk and is not empty, then
      execute aBlock"
    (aPole ~= thisDisk pole) & ((stacks at: aPole) isEmpty not) ifTrue:
      [aBlock value: (stacks at: aPole) first "execute the block"]]
```

Now lets see the program run with the new algorithm. Look at the hanoi method in the **initialize** category in TowerByRules. Select and execute:

```
(TowerByRules new) hanoi
```

We programmers tend to think of the algorithm as everything, but there are lots of things to write in a program, besides the actual algorithm. If you've ever tried to program an algorithm that you already knew on a system you didn't know, you quickly discover the "other stuff." We did a somewhat unusual thing in this chapter: we kept everything from the AnimatedTowerOfHanoi example except the algorithm. Several different aspects of Smalltalk helped us replace the algorithm easily. Because we divided the problem cleanly into objects, many of the old data structures were right for the new algorithm (stacks, howMany, and a disk's name, width, pole, and rectangle). Because Smalltalk sends messages between objects, and because the methods that answer those messages are short and single-purposed, many of our example's methods for input, output, and initialization did not need to change when the algorithm changed. Because Smalltalk allows subclasses to inherit structure and behavior from their parent classes, we did not have to make a copy of the entire class HanoiDisk in order to change it. The definition of our new class, HanoiDiskRules, contains only the changes from the old class. As programmers, we spend much of our time changing existing programs. Many of the features in the Smalltalk-80 system are there to support the following principle (often expressed by Peter Deutsch): When you make a conceptual change, you should only have to modify the parts of the program that embody that concept.