Principled Programming

Introduction to Coding in Any Imperative Language

Tim Teitelbaum

Emeritus Professor
Department of Computer Science
Cornell University

Debugging

To err is human, and despite best efforts, problems inevitably arise. Errors in code are called *bugs*, and finding them is *debugging*.

☞ **Avoid debugging like the plague.**

Bugs can be *overt*, i.e., their presence is manifest, or bugs can be *latent*, i.e., not manifest, and lying in wait to bite.

The purpose of *testing* is to make as many bugs apparent as possible.

Bugs are revealed when code is run on particular *test data*. A program that runs to correctly on some particular test data is not necessarily bug free.

☞ **Validate program output thoroughly.**

Bugs may manifest as:

- Wrong output
- Infinite loops
- Execution crashes
- Abysmal performance

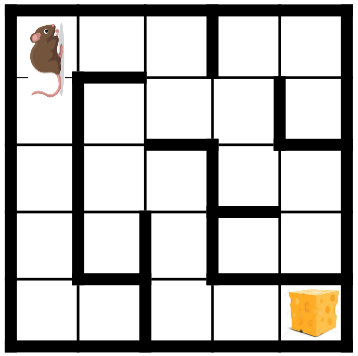
We present six bugs in real code, and describe how one can track them down.

We then demonstrate a *debugger,* a tool that assists in debugging.

Finally, we describe *defensive programming*, a prophylactic technique for revealing bugs in a helpful manner.

☞ **Validate program output thoroughly.**

Example Bugs:



In Bugs A through F, we deliberately introduce bugs into the code for Running a Maze from Chapter 16, or Appendix V.

For each bug, we run the program on the input maze shown.

Each bug is presented in four sections:

- Mistake
- Observed effect
- Forward trace
- Debugging

A *mistake* made in the code results in an *observed effect*, which is explained with the aid of a *forward trace* of execution. We then present how *debugging* could start from the observed effect, and discover the offending mistake.

The essence of the approach is to selectively instrument code so that it emits increasingly useful, partial forward traces that eventually allow you to pinpoint the bug.

Bug A: Code instrumentation.

- *Mistake:* We coded `isFacingWall` incorrectly, writing ">=" instead of "==":

```
45  public static boolean isFacingWall()
46     { return M[r+deltaR[d]][c+deltaC[d]] >= Wall; }
```

- *Observed effect:* Execution completes normally after emitting the incorrect output "Unreachable".
- *Forward trace:* Recall that `Wall` is -1 and `NoWall` is 0. Because the bug in `isFacingWall` causes it to always return **true**, the rat fails to find a way out of the upper-left cell. After three consecutive clockwise turns, it faces left (d=3), at which point `AboutToRepeat` returns **true**, and `Solve` completes

  Routine `RunMaze.Output` then calls `isAtCheese`, which returns **false**, so it prints "Unreachable".

Bug A: Code instrumentation.

- *Debugging.* The output is wrong. Perhaps `MRP.Input` failed to establish a correct maze in M. To check, we insert a call to `PrintMaze` immediately after the maze is read in:

```
3  /* Input maze, or reject input as malformed. */
4  private static void Input() {
5     MRP.Input();
      MRP.PrintMaze();
6     } /* Input */
```

We run the program again, and see that input worked fine.

`Solve` emits no output, so we need to instrument it to get a forward trace of its actions. We write `MRP.PrintState`, which emits the parameter string `s` followed by the `r`, `c`, `d`, and `move` components of MRP state:

```
   public static void PrintState(String s) {
      System.out.println(s+": "+r+" "+c+" "+d+" "+move);
      }
```

Bug A: Code instrumentation.

A convenient place to invoke `PrintState` is at the beginning of the loop in `Solve`, which will provide a top-level trace of the algorithm:

```
 8  /* Compute a direct path through the maze, if one exists. */
 9  private static void Solve() {
10     while ( !MRP.isAtCheese() && !MRP.isAboutToRepeat() ) {
          MRP.PrintState("Solve");
11        if (MRP.isFacingWall()) MRP.TurnClockwise();
12        else if (!MRP.isUnvisited()) Retract();
13        else {
14           MRP.StepForward();
15           MRP.TurnCounterClockwise();
16           }
          }
17    } /* Solve */
```

Bug A: Code instrumentation.

We run the program again, and luck out because the output is very short.

```
Solve: 1 1 0 1
Solve: 1 1 1 1
Solve: 1 1 2 1
Unreachable
```

It is clear from this trace that line 11 of `Solve` is repeatedly invoking `TurnClockwise`, and that the rat never moves from the upper-left cell. This can only happen if `isFacingWall` is **true** in every direction.

We have confirmed from the output of `PrintMaze` that there is no wall to the right of the upper-left cell, so the problem must be in `isFacingWall`.

Inspection of its code reveals the bug.

This is about as easy as debugging gets: From the observed effect, i.e., the incorrect output, and from our first attempt at instrumentation, we converged on the bug in short order.

Bug B: Instrumentation can produce vast amounts of output.

- *Mistake:* We coded `isAtCheese` incorrectly, writing “hi+1” rather than “hi”.

```
51  public static boolean isAtCheese()
52     { return (r==hi+1)&&(c==hi+1); }
```

- *Observed effect:* Execution completes normally after emitting the incorrect output “Unreachable”, the same as Bug A.
- *Forward trace:* The rat exhaustively explores the maze, not stopping at the cheese in the lower-right cell because the bug in `isAtCheese` causes it to always return **false**.

  When the rat returns to the upper left, and faces left (d=3), the exploration completes, and the output routine prints “Unreachable”.

Bug B: Instrumentation can produce vast amounts of output.

- *Debugging:* The observed effect is exactly the same as in Bug A, so we proceed in the same manner. However, this time the diagnostic output reveals an exhaustive maze exploration that blows right by the cheese at ⟨r,c⟩ = ⟨9,9⟩.

  This is enough information to focus our attention on method `isAtCheese`. Inspection reveals why it returned **false** when the rat entered the lower-right cell.

  The example illustrates that instrumentation can easily produce vast amounts of diagnostic output. However, we need not study it in detail because the salient information is apparent from the sole fact that the rat reached the cheese at ⟨r,c⟩ = ⟨9,9⟩, and didn't stop.

  Bug B was not much more difficult to diagnose than Bug A.

```
Solve: 1 1 0 1
Solve: 1 1 1 1
Solve: 1 3 0 2
Solve: 1 3 1 2
Solve: 1 5 0 3
...
Solve: 7 5 2 8
Solve: 9 5 1 9
Solve: 9 7 0 10
Solve: 9 7 1 10
Solve: 9 9 0 11
Solve: 9 9 1 11
Solve: 9 9 2 11
Solve: 9 9 3 11
Solve: 9 7 2 10
...
Solve: 3 1 3 2
Solve: 3 1 0 2
Unreachable
```

Bug C: Deductive reasoning.

- *Mistake:* We coded `TurnCounterClockwise` incorrectly, thinking that if increment-modulo-4 is `(d+1)%4`, then by analogy decrement-modulo-4 must be `(d-1)%4`:

```
37  public static void TurnCounterClockwise()
38     { d = (d-1)%4; }
```

Bug C: Deductive reasoning.

- *Observed effect:* Execution stops with a "subscript out-of-bounds" exception. The following diagnostic message is printed:

```
java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds for length 4
   at MRP.isFacingWall(MRP.java:46)
   at RunMaze.Solve(RunMaze.java:11)
   at RunMaze.main(RunMaze.java:41)
```

  The message says that an array of length 4 is indexed with a subscript of -1, and that the exception was triggered in method `isFacingWall`, at line 46 of class MRP:

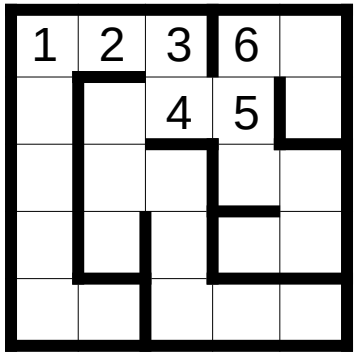
```
45  public static boolean isFacingWall()
46     { return M[r+deltaR[d]][c+deltaC[d]]==Wall; }
```

  The remaining lines are the *call stack*, and list (in reverse call order) method invocations that have not yet returned, i.e., line 41 in `main` invoked `Solve`, which on line 11 invoked `isFacingWall`.

Bug C: Deductive reasoning.

- *Forward trace:* Because the (incorrect) expression (d-1)%4 in `TurnCounterClockwise` correctly turns counterclockwise when d is greater than 0, the bug has no effect until the method is first invoked with direction d equal to 0, i.e., facing up. Accordingly, execution proceeds without incident until stepping forward into cell 6, whereupon it turns counterclockwise, which (incorrectly) sets d to -1. What happens next?

  The algorithm of `Solve` continues, and checks for the presence or absence of a wall by invoking `isFacingWall`. This is the moment when d is used to index array `deltaR`, and is out of bounds.



Bug C: Deductive reasoning.

- *Debugging:* How might a backward analysis proceed from the observed effect?

  The diagnostic output is sufficient to conclude that the proximal cause of the problem is likely that the value of `d` is -1. How so?

  Because the only arrays mentioned on the offending line of code that has been identified by the exception (MRP.isFacingWall:46) are `M[][]`, `deltaR[]`, and `deltaC[]`.

  It is conceivable that `M` has somehow been misallocated, and now involves an array of length 4, but it is far more likely that the array in question is either `deltaR[]` or `deltaC[]`, both of which have a declared length of 4. In both cases, the subscript expression is `d`.

Bug C

Bug C: Deductive reasoning.

How might d have gotten the value -1? Inspection of the code reveals five places where an assignment to d occurs:

- Initialization to 0 (at MRP.Input:109),
- Increment-modulo-4 (at MRP.TurnClockwise:35),
- Decrement-modulo-4 (at MRP.TurnCounterClockwise:38),
- Locate the previous cell on the path (at MRP.FacePrevious:78-79)
- Restore d to face the previous cell on the path (at MPR.RestoreDirection:69

We can rule out the first two: The first sets d to 0, and the second adds to d. The third is plausible as a place where d might have been assigned -1. The code made sense when we wrote it, but we are now forced to reconsider it. Consulting online documentation for the modulus operator (%), we discover our misunderstanding. Cases four and five are moot.

The diagnostic information provided by the runtime exception, together with systematic analysis, was sufficient to pinpoint the bug, even without instrumentation of the code.

Bug D: Sometimes we need iterative debugging steps.

- *Mistake:* We coded `isUnvisited` incorrectly, failing to scale the row offset by 2:

```
48  public static boolean isUnvisited()
49     { return M[r+deltaR[d]][c+2*deltaC[d]]==Unvisited; }
```

Bug D: Sometimes we need iterative debugging steps.

- *Observed effect:* Execution stops with a "subscript out-of-bounds" exception. The following diagnostic message is printed:

```
java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds for length 4
   at MRP.isFacingWall(MRP.java:46)
   at MRP.FacePrevious(MRP.java:79)
   at RunMaze.Retract(RunMaze.java:23)
   at RunMaze.Solve(RunMaze.java:12)
   at RunMaze.main(RunMaze.java:41)
```

The message states that an array of length 4 is being indexed with a subscript of 4. The offending line of code is the same code as for Bug C:

```
45  public static boolean isFacingWall()
46     { return M[r+deltaR[d]][c+deltaC[d]]==Wall; }
```

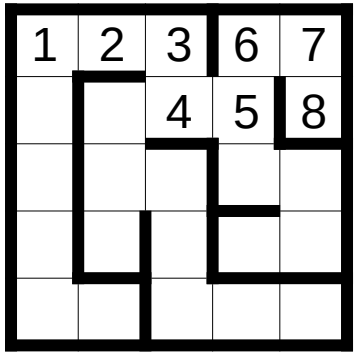
However, the call stack is different this time, and indicates that the error occurred in the course of retracting the path from a cul-de-sac.

Bug D: Sometimes we need iterative debugging steps.

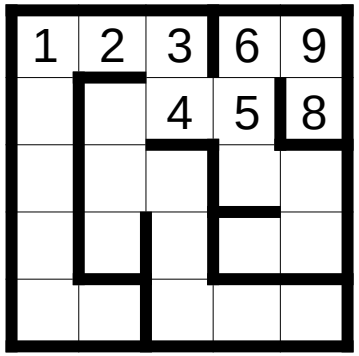
- *Forward trace:* As the rat proceeds in the forward direction, the algorithm in `Solve` steps forward into any cell that is not blocked by a wall, provided that that cell is not on the current path, which it determines by invoking the (flawed) method `isUnvisited`. Such checks will work correctly when d=1 or d=3 because, in these cases, the (correct) row increment is zero, and therefore the missing scaling factor is irrelevant.

  However, when d=0 or d=2, `isUnvisited` will always return **true**. Why? Because it will (erroneously) inspect the very same element of `M` that `isFacingWall` just inspected. Since there was no wall, `isUnvisited` will compare `NoWall` (which is 0) with `Unvisited` (which is 0), and return **true**.

  Thus, the rat makes it all the way to the end of the cul-de-sac at cell 8, at which point it (correctly) discovers walls in the right, down, and left directions, but no wall in the up direction. It is ready to step forward, but this is the precise moment when it is relying on correct execution of `isUnvisited` to detect the cul-de-sac.



Bug D: Sometimes we need iterative debugging steps.



Because d=0, the element of `M` that `isUnvisited` inspects is the one that encodes that there is no wall between cells 8 and 7, not the element that contains the 7. Accordingly, the rat blithely steps forward into the upper-right cell, overwriting 7 with 9, and then turns counterclockwise, facing left (d=3).

The program has begun to go haywire.

You may think that the rat will proceed forward, overwriting the existing path, but this is not what happens.

Recall that `isUnvisited` works correctly when d=1 or d=3. Accordingly, the rat now (correctly) detects cell 6 as already visited, which stops its forward momentum.

Retract is then invoked to back out of a (supposed) cul-de-sac at 9.

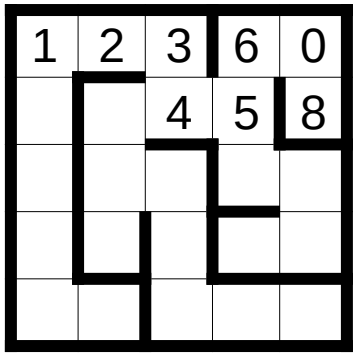
Bug D: Sometimes we need iterative debugging steps.

In preparation for backing out, `Retract` invokes `FacePrevious`,

```
77  public static void FacePrevious() {
78     d = 0;
79     while ( isFacingWall() ||
                M[r+2*deltaR[d]][c+2*deltaC[d]]!=M[r][c]-1 ) d++;
80     }
```

which (correctly) identifies cell 8 as the predecessor of cell 9, and orients the rat facing down.

`Retract` then invokes `StepBackward`, which sets the upper-right cell to `Unvisited`, i.e., 0, and moves the rat back into cell 8.



Bug D: Sometimes we need iterative debugging steps.

We are in the unwinding loop of `Retract`, so once again, it invokes `FacePrevious`, this time to search for the predecessor of the cell now numbered 8, but none of its neighbors is numbered 7.

This is a situation that is supposed to never arise. The search runs through all four legal values of d, and then invokes `isFacingWall` with (an illegal value of) d=4. This triggers the “subscript out-of-bounds” exception, with the call stack, as shown.

An important general-purpose takeaway from this forward trace is that once a bug upsets a carefully-crafted program design, it is possible for “all hell to break loose”, at which point anything may happen.

Bug D: Sometimes we need iterative debugging steps.

- *Debugging:* We now have to find the bug by reasoning backwards without the benefit of having seen the trace in advance.

  As with Bug C, inspection of the code of `isFacingWall` leads us to conclude that the value of d is 4, which is improper for an array of length 4, i.e., `deltaR` or `deltaC`.

  We identify the same five places in the code where d can (in principle) be assigned a value, but in this case, we know the culprit for sure. How so?

  The call stack shows that at the time of the exception, `isFacingWall` had been invoked by `FacePrevious`. Since that routine necessarily assigns a new `value` to d, the problem must be there. Its code searches for the direction to the cell from which the rat entered its present cell. The loop in `FacePrevious` invokes `isFacingWall` on each iteration to avoid looking outside the maze. We conclude that the search must have failed to find the cell the rat came from, which resulted in d becoming 4.

Bug D: Sometimes we need iterative debugging steps.

We ask: How can this be? Surely, the rat entered cell ⟨r,c⟩ from some cell that must have been numbered `M[r][c]-1`. There was no wall separating it from ⟨r,c⟩ when the rat came from there, and so the search in `FacePrevious` should have found it. We are befuddled.

Note that we are missing critical information: We don't know where the rat was at the moment the exception was raised, i.e., we don't know ⟨r,c⟩, and we don't know how the rat got there. The idea that the rat actually backed into 8 from 9 is beyond our wildest imagination.

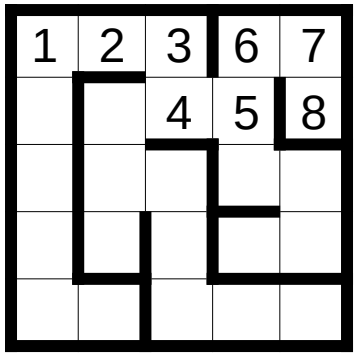
In summary, we need to reason backward along the forward trace, but the program provided no such trace as it careened toward failure. It ran silently.

What is needed now is a critical portion of the forward trace, i.e., a portion that is informative, and that points to the bug.

Bug D: Sometimes we need iterative debugging steps.

In general, a complete forward trace from the beginning of execution is far too long and detailed to be helpful. We need a goal-directed process that selectively reveals relevant portions of the trace. The process is iterative.

The call stack shows that `Retract` has been invoked, and has not yet returned, but the maze has many cul-de-sacs, so we don't know which invocation of `Retract` it is. We instrument `Retract` to find out:




```
 19  /* Unwind abortive exploration. */
 20  private static void Retract() {
         System.out.println("Enter Retract");
         ...
...      }
```

Running the program again reveals that the exception occurs within the very first invocation of `Retract`.

Bug D: Sometimes we need iterative debugging steps.

We (incorrectly) suspect that the retraction started from 8, and then backed out of 8, 7, and 6, but we don't know where along this sequence the exception occurs. To determine this, we insrument `FacePrevious`:

```
 77   public static void FacePrevious() {
 78      PrintState("FacePrevious");
         ...
...    }
```

and `StepBackward`:

```
 71   public static void StepBackward() {
 72      PrintState("StepBackward");
         ...
...    }
```

where `PrintState` is the same diagnostic method introduced in Bug A.

Bug D: Sometimes we need iterative debugging steps.

```
Enter Retract
FacePrevious: 1 9 3 9
StepBackward: 1 9 2 9
FacePrevious: 3 9 2 8
```

Running the program again produces useful diagnostic output before the exception is thrown. This output can be paraphrased as:

- FacePrevious was invoked with the rat in the upper-right cell (⟨r,c⟩=⟨1,9⟩).
- StepBackward was invoked from there with the rat facing down (d=2).
- FacePrevious was invoked again with the rat in the cell below the upper-right, i.e., ⟨3,9⟩.

This anomalous output reveals that the order of retraction is not as we had suspected. Rather than backing out of 8, the rat is backing into cell 8. At this point, there are two mysteries:

- Notwithstanding that we never should have been in the situation of backing into 8, we may ask why the program crashes?
- How did we get into this situation?

To answer the first question, we expand `PrintState` to include an invocation of `PrintMaze`:

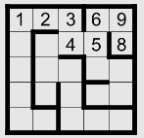
Bug D: Sometimes we need iterative debugging steps.

```
public static void PrintState(String s) {
   System.out.println(s+": "+r+" "+c+" "+d+" "+move);
   PrintMaze();
   }
```

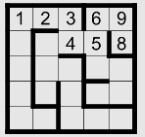
We then rerun the program, and obtain the output showm (the maze is rendered graphically rather than a textually. How do we interpret it?

- We can see that the rat reached the upper-right cell, and numbered it 9. We have no idea how it got there.
- We can see that the retraction started there, zeroed cell 9, and backed into cell 8. This makes sense, given where the retraction started.
- We can see that `FacePrevious` was invoked from cell 8, and must have been searching for a cell numbered 7. We can see that there is no such cell, which explains why d became 4, which is why `isFacingWall` triggered the "subscript out-of-bounds" exception.

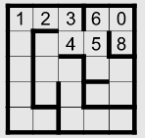
Enter Retract
FacePrevious: 1 9 3 9



StepBackward: 1 9 2 9



FacePrevious: 3 9 2 8



Bug D: Sometimes we need iterative debugging steps.

Now we understand the tail end of the trace, and why the program crashes. But we still have no idea how we got to cell 9, and why.

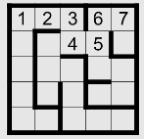
To see what led up to the retraction, we instrument `StepForward`. To limit the output, we condition the diagnostic on `move>6`:

```
 40   public static void StepForward() {
 41      if (move>6) PrintState("StepForward");
         ...
...      }
```

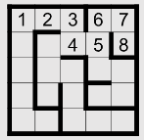
Rerunning the program, we obtain the more detailed output. How do we interpret it?

- We can see in the path display that the rat's trajectory proceeded normally from 6 to 7, and then from 7 to 8. This is as it should be.
- We can see that the rat didn't stop at 8, and went incorrectly to 9.

StepForward: 1 9 2 7

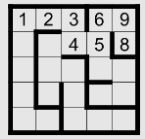


StepForward: 3 9 0 8

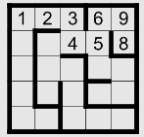


Enter Retract

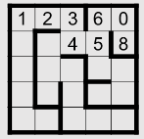
FacePrevious: 1 9 3 9



StepBackward: 1 9 2 9



FacePrevious: 3 9 2 8



Bug D: Sometimes we need iterative debugging steps.

What code should have prevented the rat from doing so? The program would have been in Solve, proceeding along a forward trajectory:

```
 8  /* Compute a direct path through the maze, if one exists. */
 9  private static void Solve() {
10     while ( !MRP.isAtCheese() && !MRP.isAboutToRepeat() ) {
11        if (MRP.isFacingWall()) MRP.TurnClockwise();
12        else if (!MRP.isUnvisited()) Retract();
13        else {
14           MRP.StepForward();
15           MRP.TurnCounterClockwise();
16           }
          }
17     } /* Solve */
```

The rat would have been in cell 8 facing up (d=0). On seeing no wall (line 11), its next step was to detect whether it was about to enter a cell already on the path (line 12). Why did it fail to invoke Retract?

Bug D: Sometimes we need iterative debugging steps.

Specifically, why was `isUnvisited` **true** when the upper-right cell so clearly contains 7? To answer this question, we turn to the code, and study it:

```
48  public static boolean isUnvisited()
49     { return M[r+deltaR[d]][c+2*deltaC[d]]==Unvisited; }
```

Ideally, we would spot the problem as soon as we look at this code. But, if we don't, we can print out the values of the subexpressions, one-by-one:

```
48  public static boolean isUnvisited() {
       if (move==8) {
         int rr= r+deltaR[d];
         int cc= c+2*deltaC[d];
         int mm = M[r+deltaR[d]][c+2*deltaC[d]];
         System.out.println("M["+rr+"]["+cc+"] is "+mm);
         }
49     return M[r+deltaR[d]][c+2*deltaC[d]]==Unvisited;
       }
```

Bug D: Sometimes we need iterative debugging steps.

Rerunning the program, this outputs the line:

```
M[2][9] is 0
```

which explains the expression returned by `isUnvisited` when `move=8`.

Staring at this line, the row index “2” should jump out at us as wrong. Cells of the maze are indexed at odd row and column subscripts. What is that “2” doing there? Looking again at the code, we cannot fail to notice that the row subscript expression is “r+deltaR[d]” when it should have been “r+2*deltaR[d]”.

Correcting this, we run the program one more time, and it works.

All we have to do now is remove the diagnostic instrumentation, and we are done.

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

- *Mistake:* We coded `deltaR` incorrectly, putting as 0 instead of a 1 for the down row offset in

```
48  // Unit vectors in direction d =        0,     1,   2,     3
49  //                                    up, right, down, left
50  private static final int deltaR[] = { -1,     0,    0,    0 };
51  private static final int deltaC[] = {  0,     1,    0,   -1 };
```

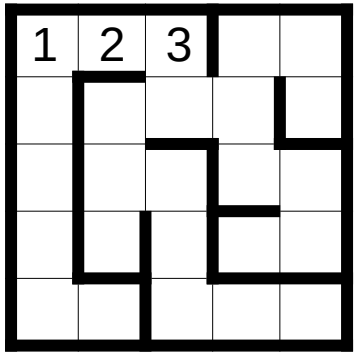
- *Observed effect:* The program runs without producing any output, and without stopping.

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

- *Forward trace:* When the rat faces down (d=2), both `deltaR[d]` and `deltaC[d]` will (incorrectly) be 0. Thus, access to `M[r+deltaR[d]][c+deltaC[d]]`, e.g., in `isFacingWall`, will really access `M[r][c]`.

  Likewise, access to `M[r+2*deltaR[d]][c+2*deltaC[d]]`, e.g., in `isUnvisited`, will also just access `M[r][c]`.

  The first time the rat faces down will be in cell 3. The algorithm in `Solve` asks (on line 11) whether the rat is facing a wall by invoking `isFacingWall`:




```
45  public static boolean isFacingWall()
46     { return M[r+deltaR[d]][c+deltaC[d]]==Wall; }
```

  However, rather than inspecting the element of `M` that contains `NoWall`, method `isFacingWall`, in effect, inspects `M[r][c]`, which contains 3. Since `Wall` is encoded by -1, which is not equal to 3, `isFacingWall` returns **false**, which (serendipitously) is correct.

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

Accordingly, the rat prepares to step forward into the cell below. But before doing so, `Solve` invokes `isUnvisited` to make sure the rat is not at a cul-de-sac, and about to step into a cell already on the path:

```
48  public static boolean isUnvisited()
49   { return M[r+2*deltaR[d]][c+2*deltaC[d]]==Unvisited; }
```

However, rather than inspecting the value of the cell below, `isUnvisited`, in effect, inspects `M[r][c]`, which contains 3, not `Unvisited`. Accordingly, the rat (incorrectly) believes it would be entering a cell already on the path, and invokes `Retract` to back out of the apparent cul-de-sac:

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

```
19  /* Unwind abortive exploration. */
20  private static void Retract() {
21     MRP.RecordNeighborAndDirection();
22     while ( !MRP.isAtNeighbor() ) {
23        MRP.FacePrevious();
24        MRP.StepBackward();
25        }
26     MRP.RestoreDirection();
27     MRP.TurnCounterClockwise();
28     } /* Retract */
```

`Retract` first invokes `RecordNeighborAndDirection` to obtain and save the `neighborNumber` of the cell in direction d. But d=2, so "the cell in direction d" is (incorrectly computed to be) the very cell the rat is in, and `neighborNumber` is set to 3.

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

Next, `Retract` invokes `isAtNeighbor` to see whether the unwinding is finished. But we are at cell 3, so the loop terminates immediately.

Next, Retract invokes `RestoreDirection`, which sets `d` to 2, which it already was.

Next, `Retract` invokes `TurnCounterClockwise`, which sets `d` to 1, i.e., once again facing a wall to the right.

This completes execution of `Retract`, and control returns to Solve.

We have been in this state before: Method `Solve` calls `TurnClockwise`, which again turns the rat to face down, and the process repeats.

We are caught in an unending loop.

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

- *Debugging:* All we know at the beginning is that we are stuck in an infinite loop.

  The first thing we must do is to interrupt execution using whatever command our programming environment offers for this. The good news is that we can stop execution; the bad news is that we typically have no idea where in the program we stopped it.

  As with Bugs C and D, we instrument the code to provide diagnostic information. This time, we choose to instrument (with calls to `MRP.PrintState`) the beginning of each time around the `Solve` loop, and entry to `Retract`.

  We quickly terminate execution (before too much output accumulates), and inspect the trace.

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

The pattern in the output is clear: We are forever repeating the three lines shown, which we interpret as follows:

- We can see that the rat is in the cell that would be numbered 3, facing right (d=1).
- We can see that the rat turns clockwise so that it faces down (d=2).
- The rat must have seen no wall because it was prepared to step forward, but apparently it believed that would renter a cell already on the path, so it called `Retract`.
- The net effect of invoking `Retract` is to return the rat to facing right (d=1).

This is mysterious, but at least we now know the extent of the infinite loop.

```
Solve: 1 1 0 1
Solve: 1 1 1 1
Solve: 1 3 0 2
Solve: 1 3 1 2
Solve: 1 5 0 3
Solve: 1 5 1 3
Solve: 1 5 2 3
Retract: 1 5 2 3
Solve: 1 5 1 3
Solve: 1 5 2 3
Retract: 1 5 2 3
Solve: 1 5 1 3
Solve: 1 5 2 3
Retract: 1 5 2 3
Etc.
```

```
Solve: 1 5 1 3
Solve: 1 5 2 3
Retract: 1 5 2 3
```

Bug E: Recurring pattern in diagnostics reveals cause of infinite loop.

We instrument `isUnvisited` as we did in Bug D, but with the output conditioned on `move==3` rather than `move==8`.

Rerunning the program produces the output shown.

The diagnostic output from `isUnvisited` is clearly problematic because it should be checking element `M[3][5]`, not element `M[1][5]`.

Inspection of the code of `isUnvisited` shows nothing wrong. The only place to inspect is in the initialization of `deltaR`:

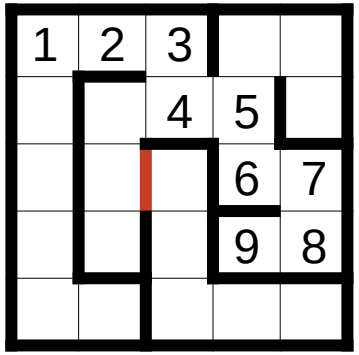
```
Solve: 1 1 0 1
Solve: 1 1 1 1
M[1][3] is 0
Solve: 1 3 0 2
Solve: 1 3 1 2
M[1][5] is 0
Solve: 1 5 0 3
Solve: 1 5 1 3
Solve: 1 5 2 3
M[1][5] is 3
Retract: 1 5 2 3
Solve: 1 5 1 3
Solve: 1 5 2 3
M[1][5] is 3
Retract: 1 5 2 3
Solve: 1 5 1 3
Solve: 1 5 2 3
M[1][5] is 3 3
Retract: 1 5 2 3
Etc.
```

```
48 // Unit vectors in direction d =       0,     1,   2,     3
49 //                                    up, right, down, left
50 private static final int deltaR[] = { -1,     0,    0,    0 };
```

There we spot the 0 instead of 1.

Fixing the error, we rerun the program, and obtain the correct output.

Bug F: Use of binary search to find a bug.



- *Mistake:* The mistake is contrived, but models a common occurrence: A rare event in obscure code causes damage that is often benign, but on occasion has disastrous effect. We concoct the example by inserting a nonsensical statement into `FacePrevious:`, which has the effect of inserting the red wall shown on move 9:

```
77  public static void FacePrevious() {
78     d = 0;
79     while ( isFacingWall() ||
                M[r+2*deltaR[d]][c+2*deltaC[d]]!=M[r][c]  ) d++;
       if ( move==9 ) M[r-2][c-3] = Wall;
80     }
```

- *Observed effect:* The incorrect output “Unreachable” is printed.
- *Forward trace:* The sample maze happens to have a cul-de-sac at move 9, so the spurious red wall is introduced, eliminating the only solution.

Bug F: Use of binary search to find a bug.

- *Debugging:* The observed effect is exactly the same as in Bug A and Bug B, so we proceed in the same manner.

  In Bug A, the diagnostic trace immediately revealed that the rat was struck in the upper-left cell. In Bug B, it revealed that the rat reached the lower-right cell, but didn't stop.

  In this bug, the output shows that rat gets nowhere near the cheese. Unfortunately, the step where the rat is blocked by the offending wall is buried deep in the trace, and we are not likely to spot it.

  Furthermore, the offense of inserting a fictitious wall was committed at an obscure earlier moment.

  Making matters still worse, the encounter with the fictitious wall was perfectly ordinary, e.g., it didn't cause the program to crash, and execution continued for a long time thereafter.

  These are the bugs that try men's souls.

Bug F: Use of binary search to find a bug.

Devising an effective strategy is left as an exercise for the reader. We give one hint.

Suppose that by hard work, and some luck, you have spotted the fictitious wall. How might you discover how it got there?

Answer: Use binary search along the timeline from the start of execution to moment when the wall's presence mattered. Repeatedly divide that interval (roughly) in half, checking on each probe for the presence or absence of the (spurious) wall, and choosing which half-interval of time to focus on next, accordingly.

You will eventually converge on the moment when the wall was introduced. Lo and behold, it is a nonsensical line of code in `FacePrevious`.

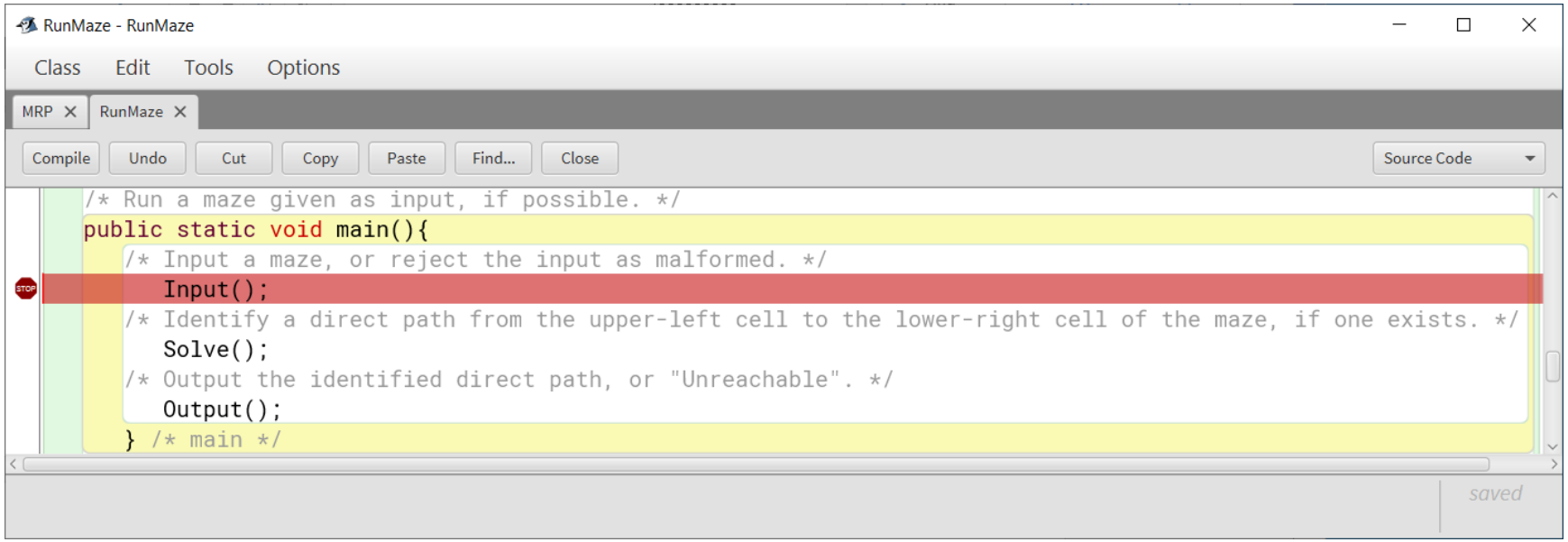
Who could have guessed?

Using a Debugger:

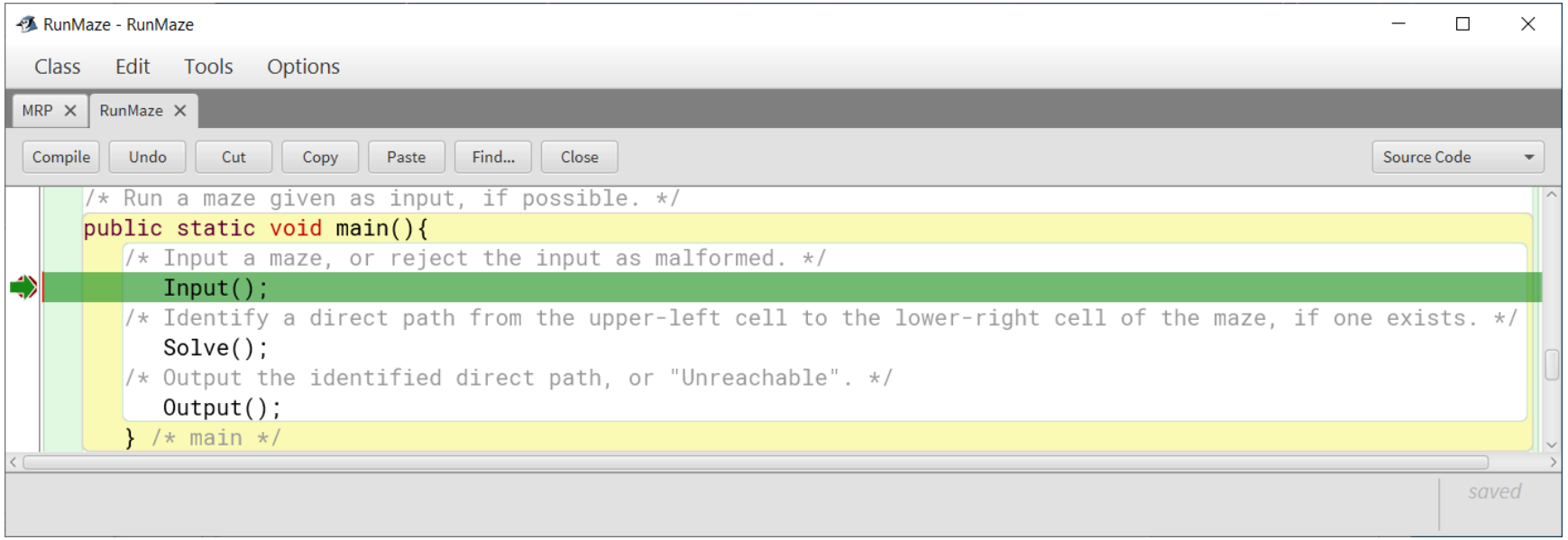
The techniques are basically the same, but some instrumentation is obviated by the debugger’s facilities.

In a debugger, you can set a *breakpoint* in the code, which causes execution to stop (and return to the debugger) if control ever reaches there.

Here, we set a breakpoint on the first line of method `main`:

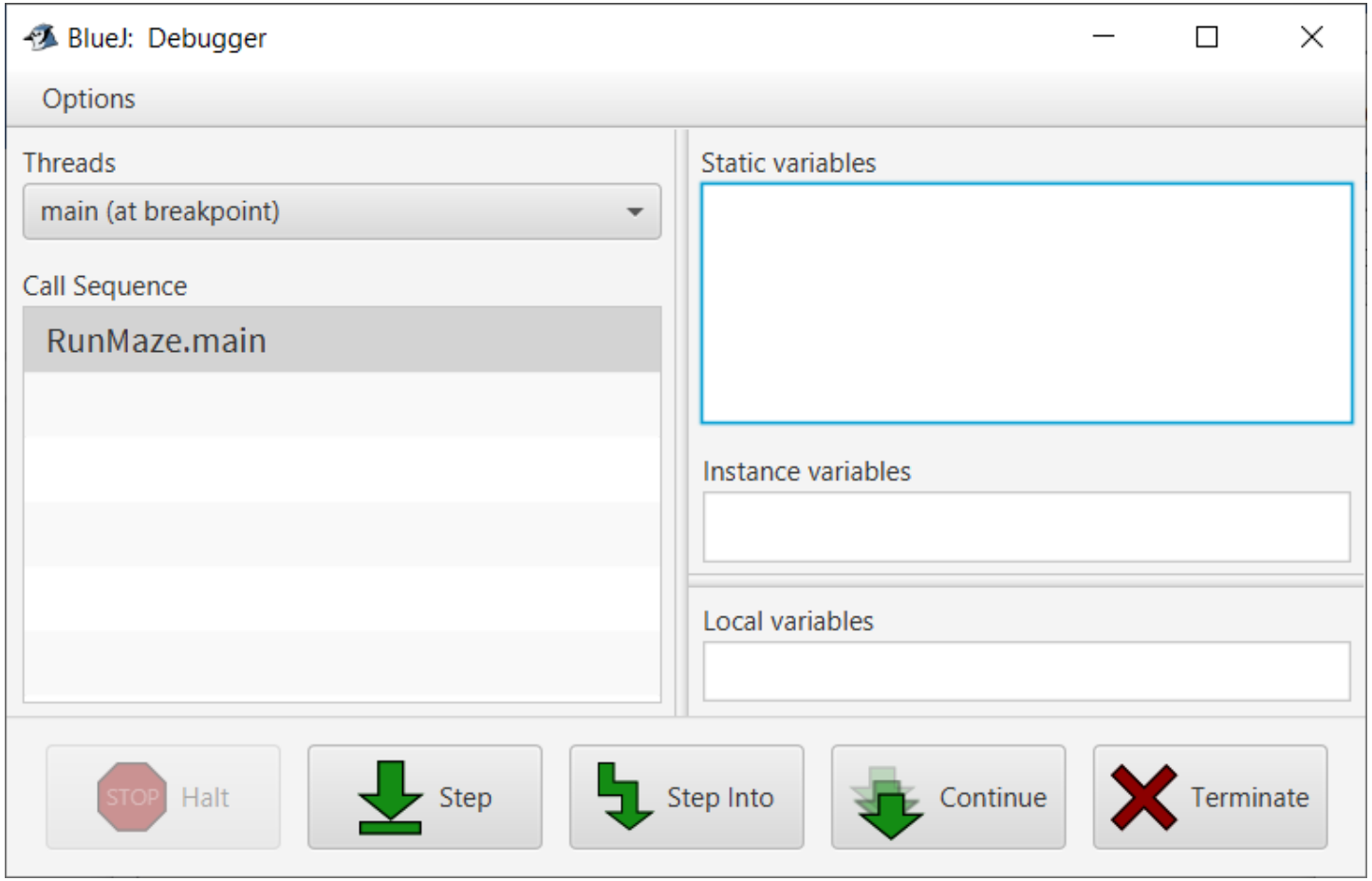


When we launch execution, we immediately hit the breakpoint:

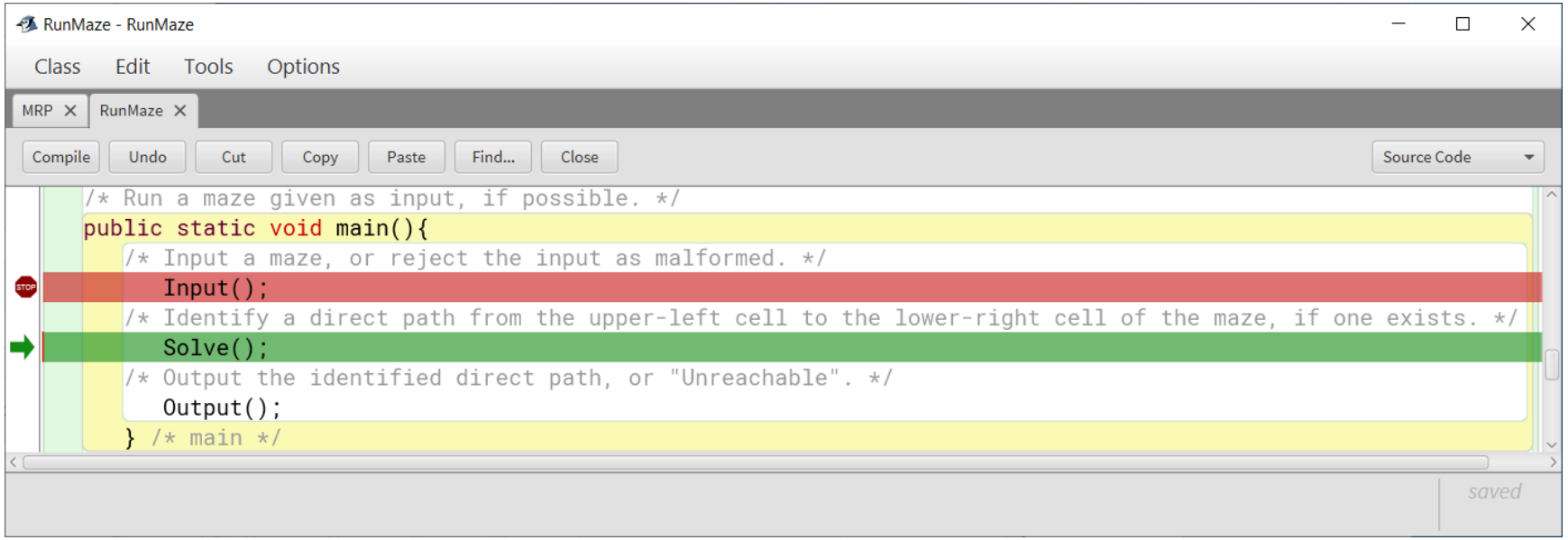


On return to the debugger, we are presented with its control panel.

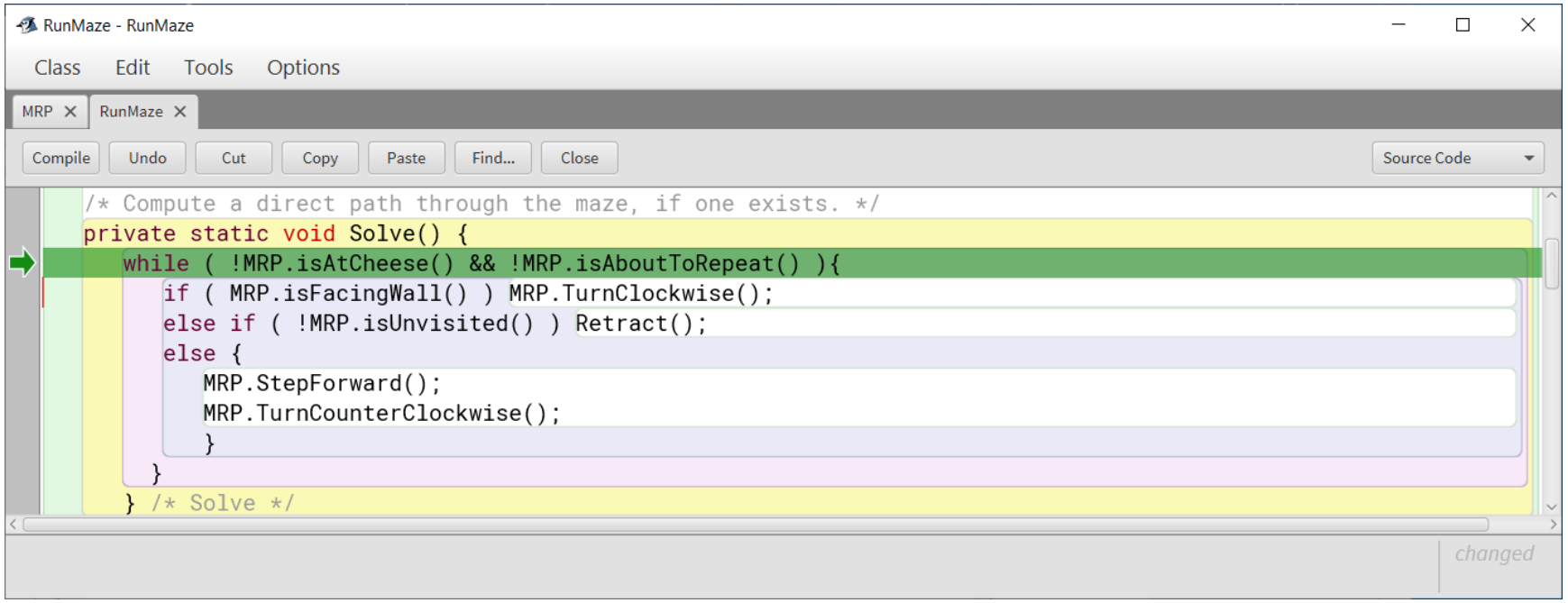
Step directs the debugger to execute the current line all in one step. This is called *single-step execution*. The call to `Input` is performed in its entirety.



This brings us to the second line of code in `main`, the call to `Solve`:



This time, we strike **Step Into**, which directs the debugger to execute the current line, but to stop at the first line in method `Solve`:

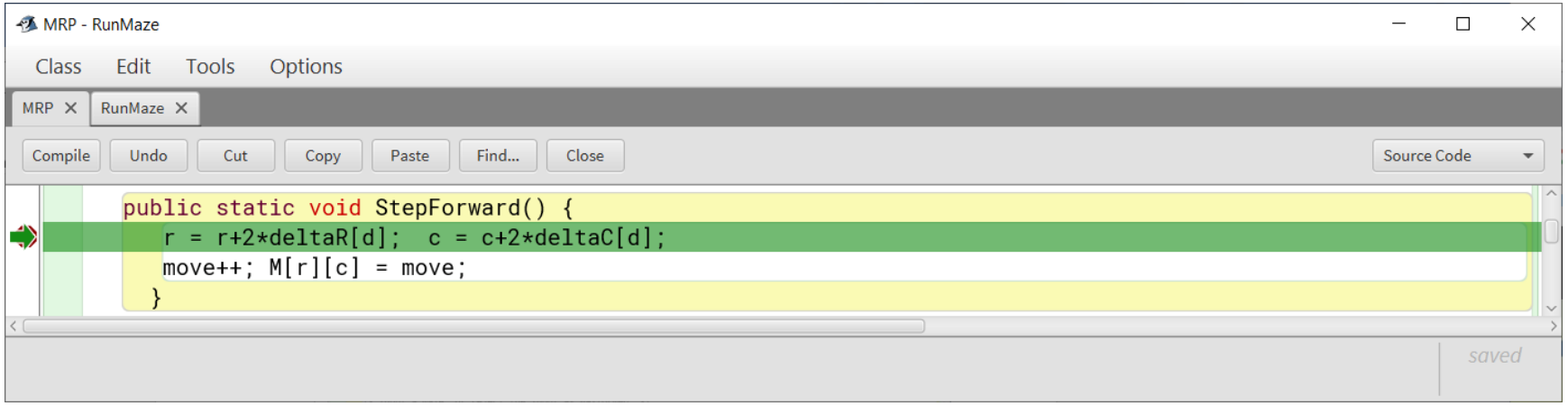


Suppose we were debugging Bug A. Recall that the bug caused the rat to never leave the upper-left cell. We can repeatedly strike **Step** and see that the loop iterates three times, calling `MRP.TurnClockwise` each time. From this, we can conclude that the bug must be in `MRP.isFacingWall`.

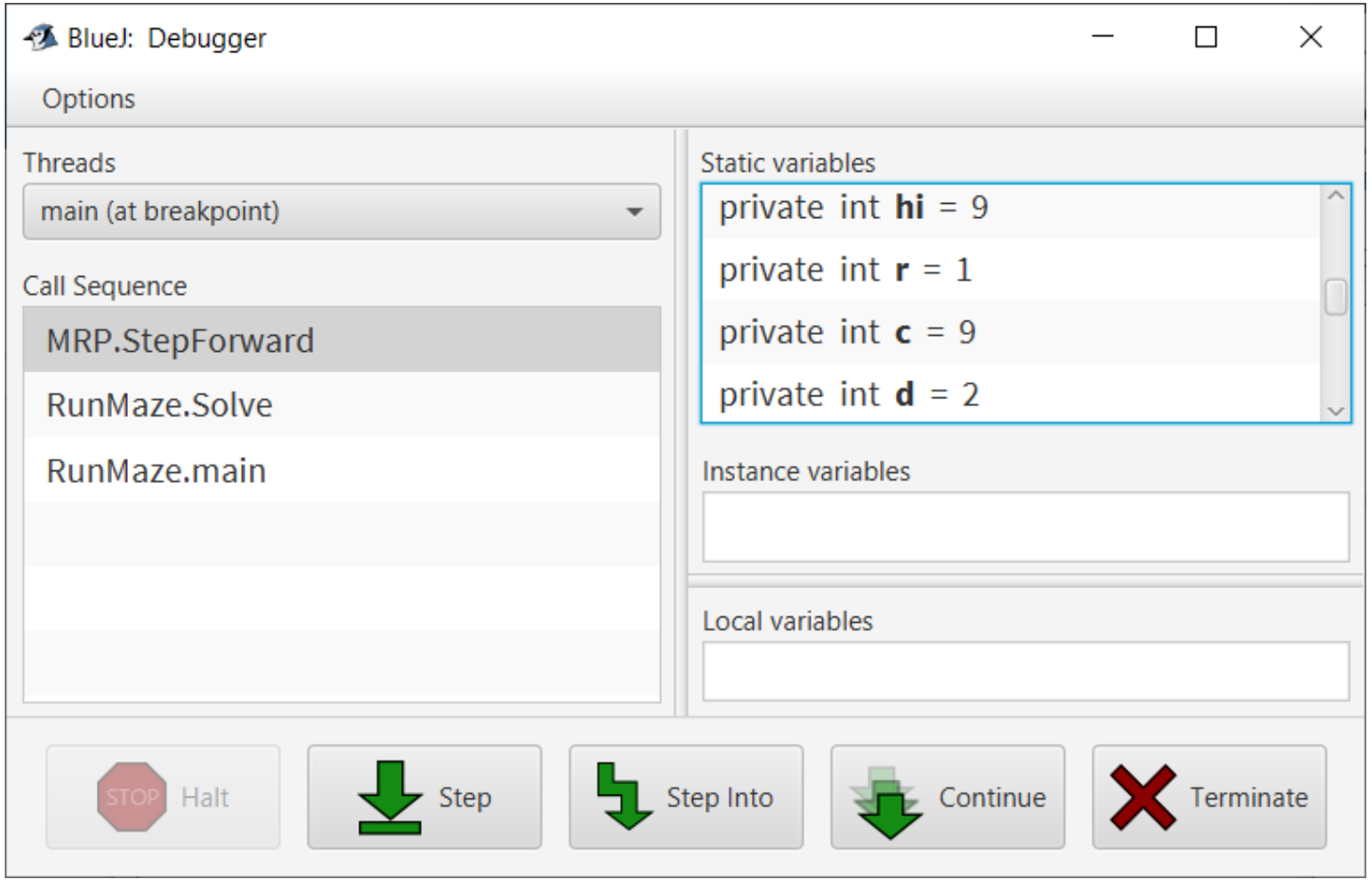
Now suppose we were debugging Bug B. Recall that the mistake in Bug B causes the rat to not stop when it reaches the lower-right cell. In this case, single-step execution gets tedious because the rat visits so many cells.

We can set a breakpoint on the invocation of `MRP.StepForward`, and then strike **Continue**, which directs execution to go full speed (until reaching a breakpoint).

From the invocation of `StepForward`, we can strike **Step Into**, and review the MRP state variables in the control panel:



There we see the call stack (`main` called `Solve`, which called `MRP.StepForward`), and the values of MRP's static variables:

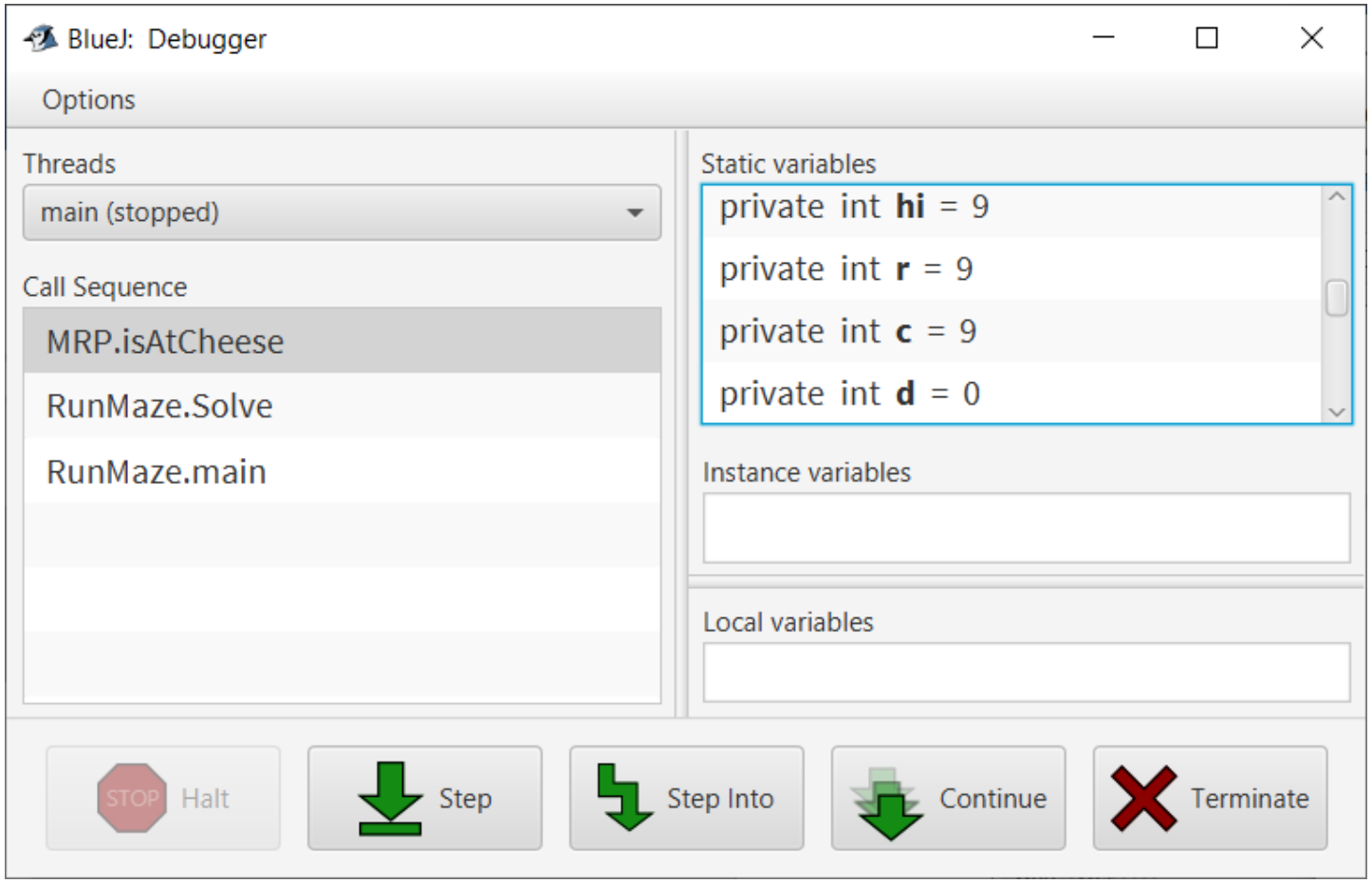


Assuming we want to confirm that we get to the lower-right cell, we can strike **Continue** a few more times, but then switch to repeatedly using **Step Into**:

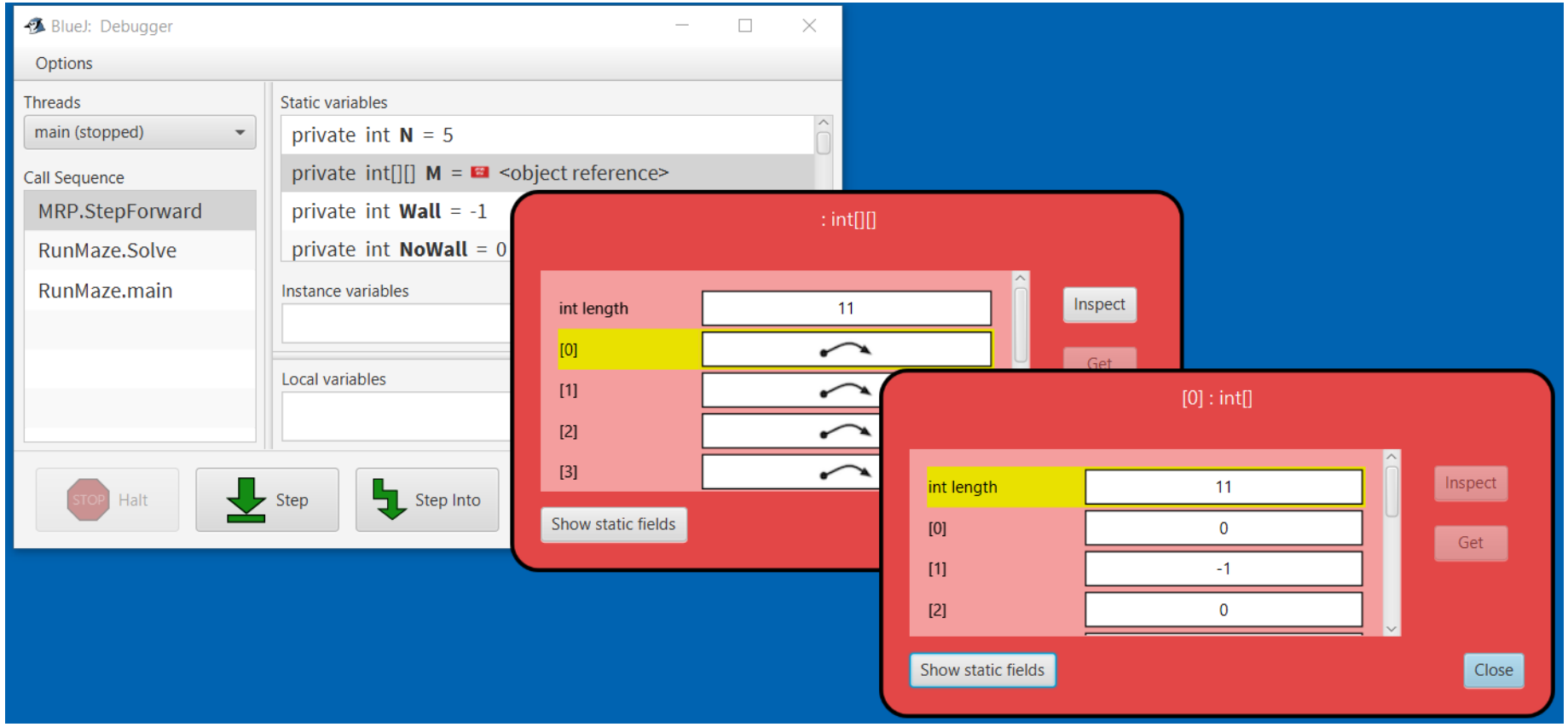
Eventually, we arrive in `MRP.isAtCheese`, with ⟨r,c⟩ = ⟨9,9⟩, confirming that we got to the lower-right cell.

One more **Step Into**, and we see that `MRP.isAtCheese` fails to return **true**.

Inspection of the code then reveals the bug.



Recall that in debugging Bug A, our initial suspicion was that the maze might not have been correctly input. We could have inspected the array M directly in the debugger.



Establish a framework:

A program's code makes assumptions at various places without explicitly checking that they hold.

The earliest manifestation of a bug is internal: An assumption is violated. However, such a violation is not immediately observable externally.

In some cases, the violation of an assumption is benign, e.g., a representation invariant gets broken, but program execution from that point on does not rely on the truth of the full invariant. In other cases, the program eventually throws a runtime exception, or gets caught in an infinite loop, or produces bad output.

Defensive programming aims to make the violation of assumptions manifest as early as possible during program execution. It does so by using assertions.

Defensive Programming: Stay in Control.

A program's code makes assumptions at various places without explicitly checking that they hold.

The earliest manifestation of a bug is internal: An assumption is violated, but the violation is not immediately observable externally.

In some cases, the violation of an assumption is benign, e.g., a representation invariant gets broken, but program execution from that point on does not rely on the truth of the full invariant. In other cases, the program eventually throws a runtime exception, or gets caught in an infinite loop, or produces bad output.

Defensive programming aims to make the violation of critical assumptions manifest as early as possible during program execution. It does so by using **assert** statements.

Defensive Programming: Stay in Control.

The main places in code at which assumptions can be checked are these:

- For an input statement, the code assumes that the input data will comply with its specified format.
- For a statement-level specification of the form:

```
/* Given precondition, establish postcondition. */
   Implementation
```

the code assumes that the *precondition* is **true** before the first statement of the *implementation*, and the *postcondition* is **true** after the last statement of the *implementation*.

Defensive Programming: Stay in Control.

- For a declaration of the form:

```
Declaration-of-one-variable // Representation invariant
```

or a declaration of the form:

```
/* Representation invariant. */
   Declarations-of-related-variables
```

the *representation invariant* is assumed to hold throughout the scope of the *variables*, except prior to initialization, and until completion of the code that seeks to reestablish the invariant after an update.

Defensive Programming: Stay in Control.

- For a loop of the form:

```
/* Loop invariant. */
   while ( condition ) statement
```

or of the form:

```
/* Loop invariant. */
   for ( init; condition; update ) statement
```

the *loop invariant* is assumed to be **true** before and after each execution of the *statement*.

Defensive Programming: Stay in Control.

- For a method definition of the form:

```
/* Given precondition on input parameters, establish postcondition
   on output parameters, and return value, if any. */
Method definition
```

the definition assumes that the *preconditions* of input parameters are **true** on entry to the body of the method, and the *postconditions* of output parameters (as well as of its return value, if any) are **true** just before returning from the method.

Defensive Programming: Stay in Control.

- For a method invocation of the form:

```
name( argument-list )
```

the code assumes that each input argument value satisfies the *precondition* of the corresponding input parameter, and that each output argument (as well as the return value, if any) satisfies the *postcondition* of the corresponding output parameter (or result).

Defensive Programming: Stay in Control.

Assert statements were introduced in Chapter 3 when we had scant use for them. We now illustrate their use the program for Running a Maze.

We implement `isValid`, a **`boolean`** method that returns **false** if it discovers evidence that one of MRP's representation invariants is not being correctly maintained.

Defensive Programming: Stay in Control.

Consider the rat's representation invariant in class MRP:

```
16  /* Rat. The rat is located in cell M[r][c] facing direction d, where a
17     d of ⟨0,1,2,3⟩ represents the orientation ⟨up,right,down,left⟩,
18     respectively. */
19     private static int r, c, d;
```

This utility method confirms that this representation invariant holds:

```
    /* Return false iff rat's representation invariant is violated. */
    public static boolean isValidRat() {
       if ( r<0 || r>hi || c<0 || c>hi ) return false;
       else if ( d < 0 || d>3 ) return false;
       else if ( M[r][c]!=move ) return false;
       else return true;
       } /* isValidRat */
```

Defensive Programming: Stay in Control.

Consider the path's representation invariant in class MRP:

```
21  /* Path. When the rat has traveled to cell ⟨r,c⟩ via a given path through
22     cells of the maze, the elements of M that correspond to those cells
23     will be 1, 2, 3, etc., and all other elements of M that correspond to
24     cells of the maze will be Unvisited. The number of the last step in
25     the path is move. */
26   private static final int Unvisited = 0;
27   private static int move;
```

Routine `isValidPath` was introduced to make our Running a Maze program self-checking, but it can be used now to validate maintenance of the path invariant to cell ⟨r,c⟩ whenever we want.

Defensive Programming: Stay in Control.

Putting them together:

```
/* Return false on evidence that a representation invariant is violated. */
public static boolean isValid() {
   return isValidRat() && isValidPath(r,c);
   } /* isValid */
```

Defensive Programming: Stay in Control.

Check MRP representation once per iteration of Solve:

```
/* Compute a direct path through the maze, if one exists. */
private static void Solve() {
   while ( !MRP.isAtCheese() && !MRP.isAboutToRepeat() ) {
      assert MRP.isValid(): "Invalid MRP representation.";
      if (MRP.isFacingWall()) MRP.TurnClockwise();
      else if (!MRP.isUnvisited()) Retract();
      else {
         MRP.StepForward();
         MRP.TurnCounterClockwise();
         }
      }
   } /* Solve */
```

Defensive Programming: Stay in Control.

or even scattered throughout the (error-prone) implementation:

```
public static void TurnCounterClockwise() {
    d = (d-1)%4;
    assert isValid(): "Invalid MRP representation.";
    }
```

which would have led to the immediate detection of Bug C:

```
java.lang.AssertionError: Invalid MRP representation.
 at MRP.TurnCounterClockwise(MRP.java:39)
 at RunMaze.Solve(RunMaze.java:15)
 at RunMaze.main(RunMaze.java:41)
```