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 15, or Appendix V.

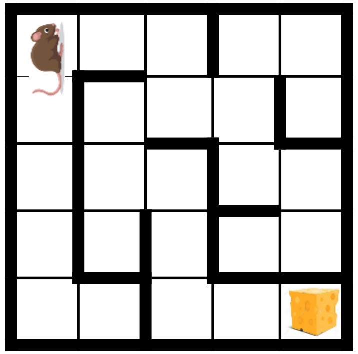
For each bug, we run the program on the input maze shown.

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.



Example Bugs

Bug A

Bug A: Code instrumentation.

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

```
public static boolean isFacingWall()
   { 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:

```
/* Input maze, or reject input as malformed. */
private static void Input() {
   MRP.Input();
   MRP.PrintMaze();
   } /* 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 each iteration of the loop in `Solve`, which will provide a top-level trace of the algorithm:

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

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

- *Observed effect:* Execution completes normally after emitting the same incorrect output as Bug A: "Unreachable".
- *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: Error diagnostics contain vital information.

- *Mistake:* We coded `TurnClockwise` incorrectly, forgetting to compute the result of incrementing d **mod** 4:

```
public static void TurnClockwise()
   { d = (d+1)      ; }
```

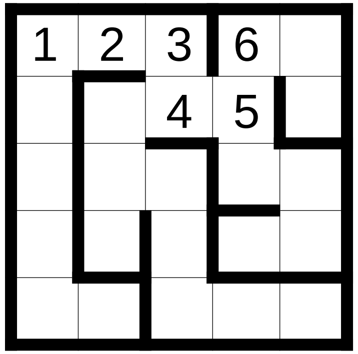
- *Observed effect:* Execution stops with a “subscript out-of-bounds” exception. The following diagnostic message is printed:

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

Bug C: Error diagnostics contain vital information.

The message states that there has been an attempt to index an array with a subscript that is 4, which is out of range, and that the exception was triggered in method `isFacingWall`, at line 46 of class `MRP`. The remaining lines are the *call stack* at the time of the error, and list method invocations that have not yet returned, i.e., line 41 in the `RunMaze` module invoked `main`, which on line 11 invoked `Solve`, which on line 46 invoked `isFacingWall`.

- *Forward trace:* Because the (incorrect) expression `(d+1)` in `TurnClockwise` correctly increments `d` when it is less than 3, the bug has no effect until we reach cell 6. There, initially facing left, we expect to turn clockwise (to face up), and turn clockwise again (to face right). After each turn, `Solve` would normally call `isFacingWall` to see if another turn is needed, but the first such invocation attempts to subscript arrays `deltaR` and `deltaC` with an out-of-bounds subscript of 4, and the program stops.



Bug C: Error diagnostics contain vital information.

- *Debugging:* The example illustrates three all-important facts:
    1. When a crash occurs, you may know little about how far execution progressed before stopping.
    2. The location where a bug triggers a crash (e.g., `isFacingWall`) may be arbitrarily distant from the location that contains the flaw (e.g., `TurnClockwise`).
    3. Error diagnostics can contain vital information.

  We know from the diagnostic text that something has gone wrong in:

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

  so the offending index is necessarily attempting to subscript into one of the three arrays `deltaR`, `deltaC`, or `M`. We know from the crash diagnostic that the implicated array has length 4, which would seem to rule out `M[][]` unless something is wildly wrong for this 5-by-5 example maze. Thus, the array must be either `deltaR` or `deltaC`, for which the subscript in either case is d.

Bug C: Error diagnostics contain vital information.

To learn how far execution progressed before the crash, an easy approach is to place a call to `PrintState` in method `Solve`, after line 30, i.e., the same as we did for Bugs A and B.

While a vast amount of text may fly by us on the screen, we are only interested in the last few lines before the crash, so the amount of output is of no great concern.

We can readily interpret the last two lines of output as:

- We are in cell 6, and `d` was 3.
- We remained in cell 6, and `d` became 4, which is an out-of-bounds subscript for either `deltaR` or `deltaC`.

The code that increments d, and that would have been called by `Solve` immediately before it called `isFacingWall`, is `TurnClockwise`.

Knowing that `TurnClockwise` incorrectly set `d` to 4, we cannot help but see the bug by staring at the code.

```
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
Solve 3 5 1 4
Solve 3 7 0 5
Solve 1 7 3 6
Solve 1 7 4 6
⟨crash⟩
```

Bug D: Interleave iterative debugging steps with deduction.

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

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

Bug D: Interleave iterative debugging steps with deduction.

- *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:

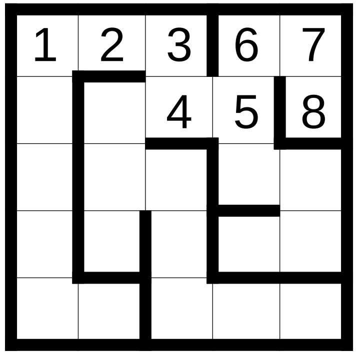
```
45  public static boolean isFacingWall()
46     { return M[r+deltaR[d]][c+deltaC[d]]==Wall; }
```

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: Interleave iterative debugging steps with deduction.

- *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 `isFacingUnvisited`. Despite the bug, these checks will work correctly when d=1 or d=3 because, in these cases, the (correct) row increment is 0, and therefore the missing scaling factor is irrelevant. However, when d=0 or d=2, `isFacingUnvisited` 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, `isFacingUnvisited` 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. This is the precise moment when correct execution of `isFacingUnvisited` to detect the cul-de-sac is critical.



Bug D: Interleave iterative debugging steps with deduction.

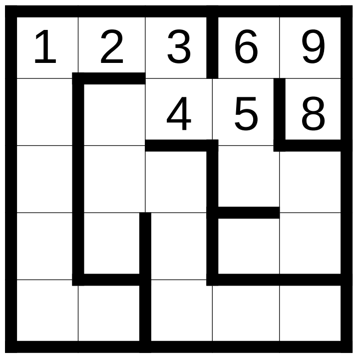
Because `d=0`, the element of `M` that `isFacingUnvisited` inspects is the one that encodes that there is no wall between cells 8 and 7, not the element that contains the 7. Accordingly, the rat blithely steps forward into the upper-right cell, overwriting 7 with 9, and then turns counterclockwise, facing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



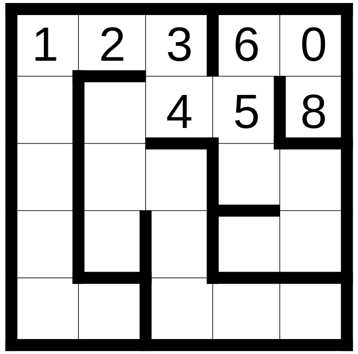
Bug D: Interleave iterative debugging steps with deduction.

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

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

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: Interleave iterative debugging steps with deduction.

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: Interleave iterative debugging steps with deduction.

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

  As with Bug C, the crash occurs in `isFacingWall`, but this time in a different context: `Retract` called `facePrevious`, which called `isFacingWall`.

  As with Bugs A, B, and C, we arrange to call `printState("Solve")` from the loop of method `Solve`, and we obtain the output shown at right. What can we infer from it?

  - We are in cell ⟨r,c⟩ = ⟨1,9⟩, and believe that we have made 9 moves.
  - We have been in this cell before, when move was 7. We have no business being there again, but have no idea how this happened.
  - We can see that our recent trajectory has been ⟨1,9⟩⇒⟨3,9⟩⇒⟨1,9⟩.
  - We can see in the Traceback that we are in the midst of a retraction, but don't know when it started.

  We place the call `System.out.println("Enter Retract")` in `Retract`, and rerun.

```
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
Solve 3 5 1 4
Solve 3 7 0 5
Solve 1 7 3 6
Solve 1 7 0 6
Solve 1 7 1 6
Solve 1 9 0 7
Solve 1 9 1 7
Solve 1 9 2 7
Solve 3 9 1 8
Solve 3 9 2 8
Solve 3 9 3 8
Solve 3 9 0 8
Solve 1 9 3 9
Traceback...
```

Bug D: Interleave iterative debugging steps with deduction.

But this output is anomalous, as the retraction should have started earlier, when we were in ⟨r,c⟩ = ⟨3,9⟩ facing up (d=0) at the 7, which we must not overwrite. Somehow, the test `isFacingUnvisited` must have failed then, i.e., concluded that we were facing an unvisited cell at ⟨1,9⟩ despite its containing 7. How could this be?

Inspection of `isFacingUnvisited`, and the obvious dissimilarity between the codes for the row and column coordinates, reveals the bug.

Interestingly, the bug was identified without our having to understand the horrors of the detailed forward trace.

```
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
Solve 3 5 1 4
Solve 3 7 0 5
Solve 1 7 3 6
Solve 1 7 0 6
Solve 1 7 1 6
Solve 1 9 0 7
Solve 1 9 1 7
Solve 1 9 2 7
Solve 3 9 1 8
Solve 3 9 2 8
Solve 3 9 3 8
Solve 3 9 0 8
Solve 1 9 3 9
Enter Retract
Traceback…
```

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

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

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

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

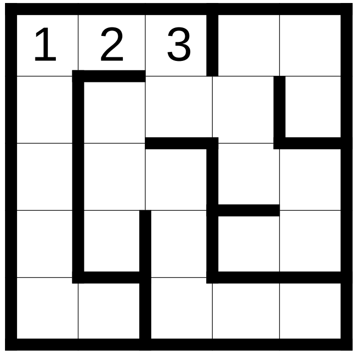
Example Bugs
Bug E

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]]` in `isFacingUnvisited`, will also just access `M[r][c]`.

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




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

  The bug causes `M[r][c]` (which contains 3) to be inspected rather than `M[r+1][c]` (which contains `NO_WALL`). Serendipitously, we return the correct value (**False**) indicating no wall despite the bug.

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

Accordingly, the rat is will step forward into the cell below, but only provided `isFacingUnvisited` indicates that the cell is not on the current path:

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

However, rather than inspecting the value of the cell below (`M[r+2][c]`), the bug causes `isFacingUnvisited` to inspect `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 at 3.

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

`Retract` first invokes `RecordNeighborAndDirection` to obtain and save the `neighborNumber` of the cell in direction `d`, and the direction to it:

```
/* Unwind abortive exploration. */
private static void Retract() {
   MRP.RecordNeighborAndDirection();
   while ( !MRP.isAtNeighbor() ) {
      MRP.FacePrevious();
      MRP.StepBackward();
      }
   MRP.RestoreDirection();
   MRP.TurnCounterClockwise();
   } /* Retract */
```

But `d`=2 (down), the very direction for which `deltaR[d]` is incorrectly initialized to 0. So “the cell in direction `d`” is (incorrectly computed to be) the very cell the rat is currently in. Accordingly, `neighborNumber` is set (incorrectly) 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`.

But we have been in this state before: In cell 3 facing right. So 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, as with the other bugs, we choose to instrument (with calls to `MRP.PrintState`) at the beginning of each iteration of the `Solve` loop, and also on 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.

The call to `isFacingUnvisited` failed, so the natural thing to do is to stare it its code and see if we can spot the problem:

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

Seeing nothing wrong, we decide to get additional diagnostic information about the value of `M` being inspected:

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

Example Bugs

Bug E

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

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

When d=2, the only way

```
rr = r + deltaR[d]
```

could be producing the wrong value is for either `r` or `deltaR[2]` to be wrong. But there appears to be nothing wrong with `r`, so the problem must be with `deltaR[2]`. Inspecting deltaR[2], we see the 0 where a 1 was needed:

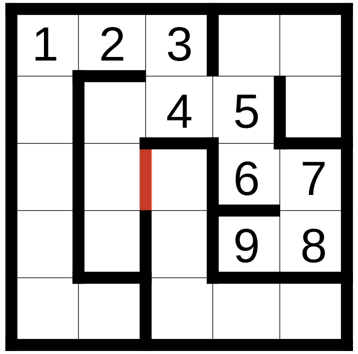
```
// Unit vectors in direction d =       0,      1,    2,      3
//                                    up, right, down, left
private static final int deltaR[] = { -1,      0,    0,      0 };
```

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

```
...
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
_Etc.
```

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

- *Mistake: 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:




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

- *Observed effect:* The incorrect output is printed: “Unreachable”.
- *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 the 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 execution 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`.

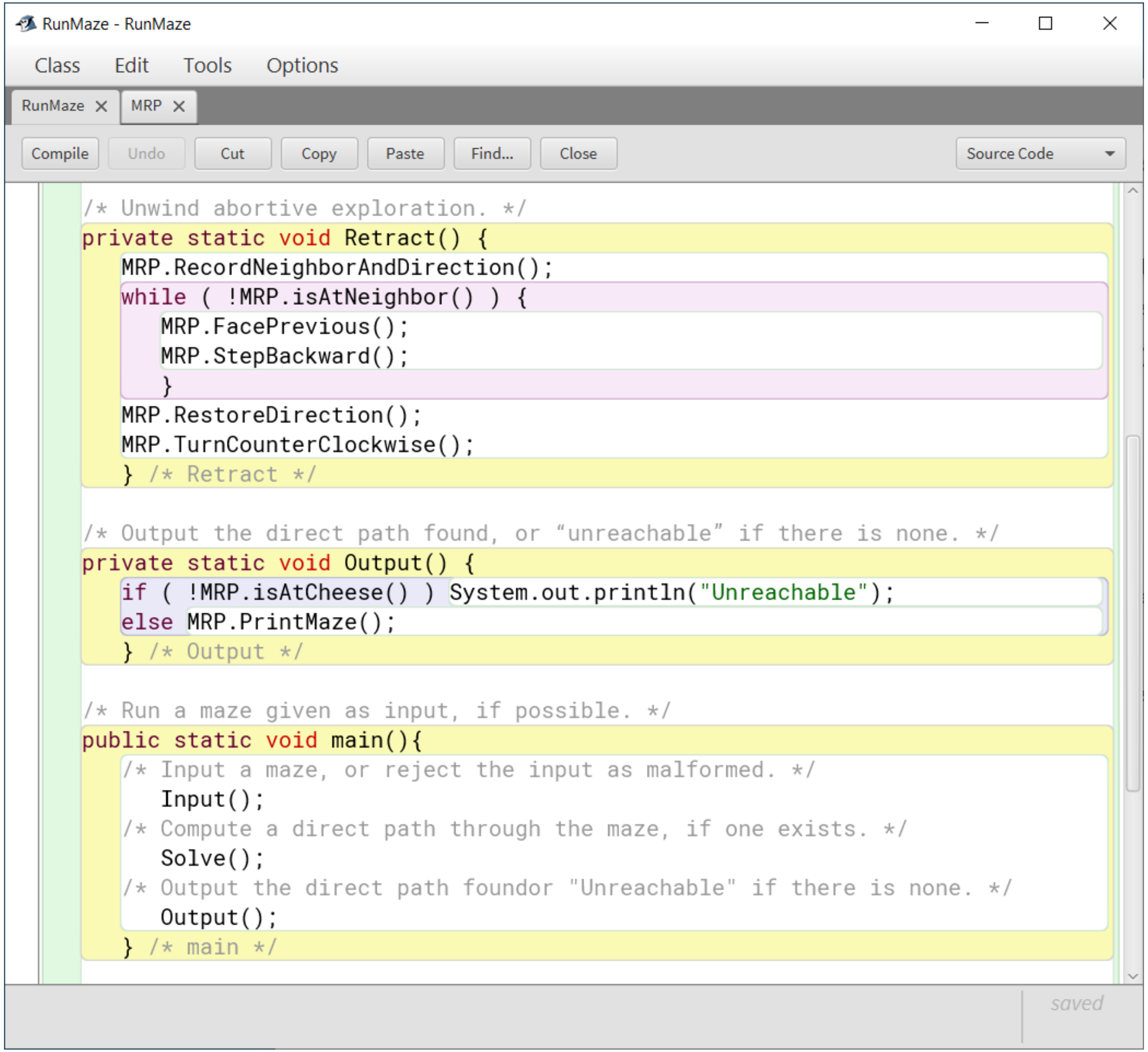
Who could have guessed?

Using a Debugger

Debuggers make debugging much easier, albeit the techniques are basically the same with or without one: Selective reconstruction of relevant portions of forward execution traces that identify the mistake.

The main benefit of a debugger is that its controls and observation mechanisms obviate much of the manual instrumentation we have been illustrating.

BlueJ is a free Integrated Development Environment (IDE) for Java. We illustrate a small sample of typical debugger features using a BlueJ project for our maze running program.

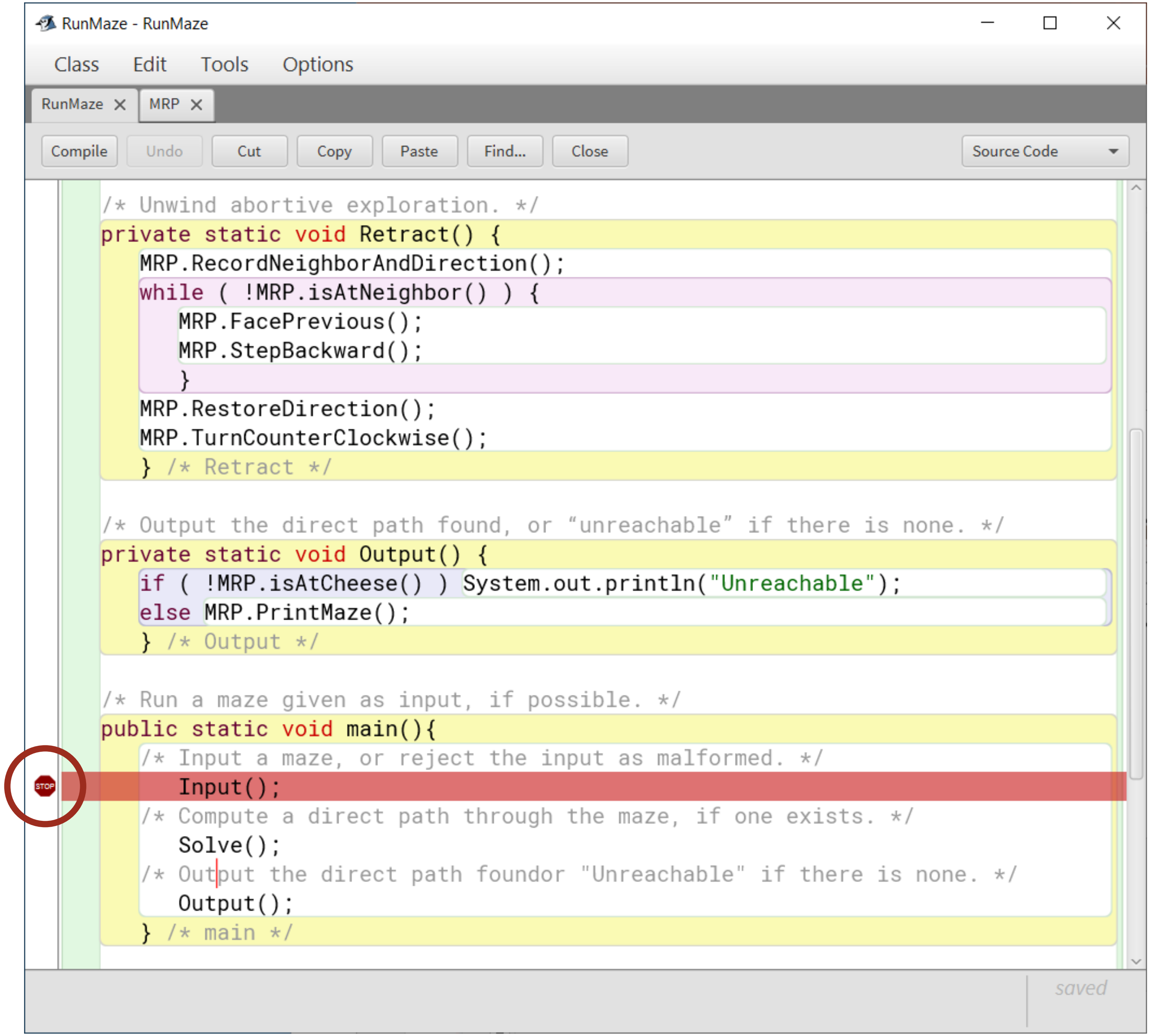


Breakpoints

A *breakpoint* is a location in code identified as a stopping point of interest.

Setting appropriate breakpoints allows execution to proceed full speed ahead, but guarantees that the user will regain control in the debugger whenever execution reaches one of the designated points of interest.

Here, we have set a breakpoint on the first line of method `main`, at a call to `Input`. We did this by clicking in the margin, where the stop sign appeared.



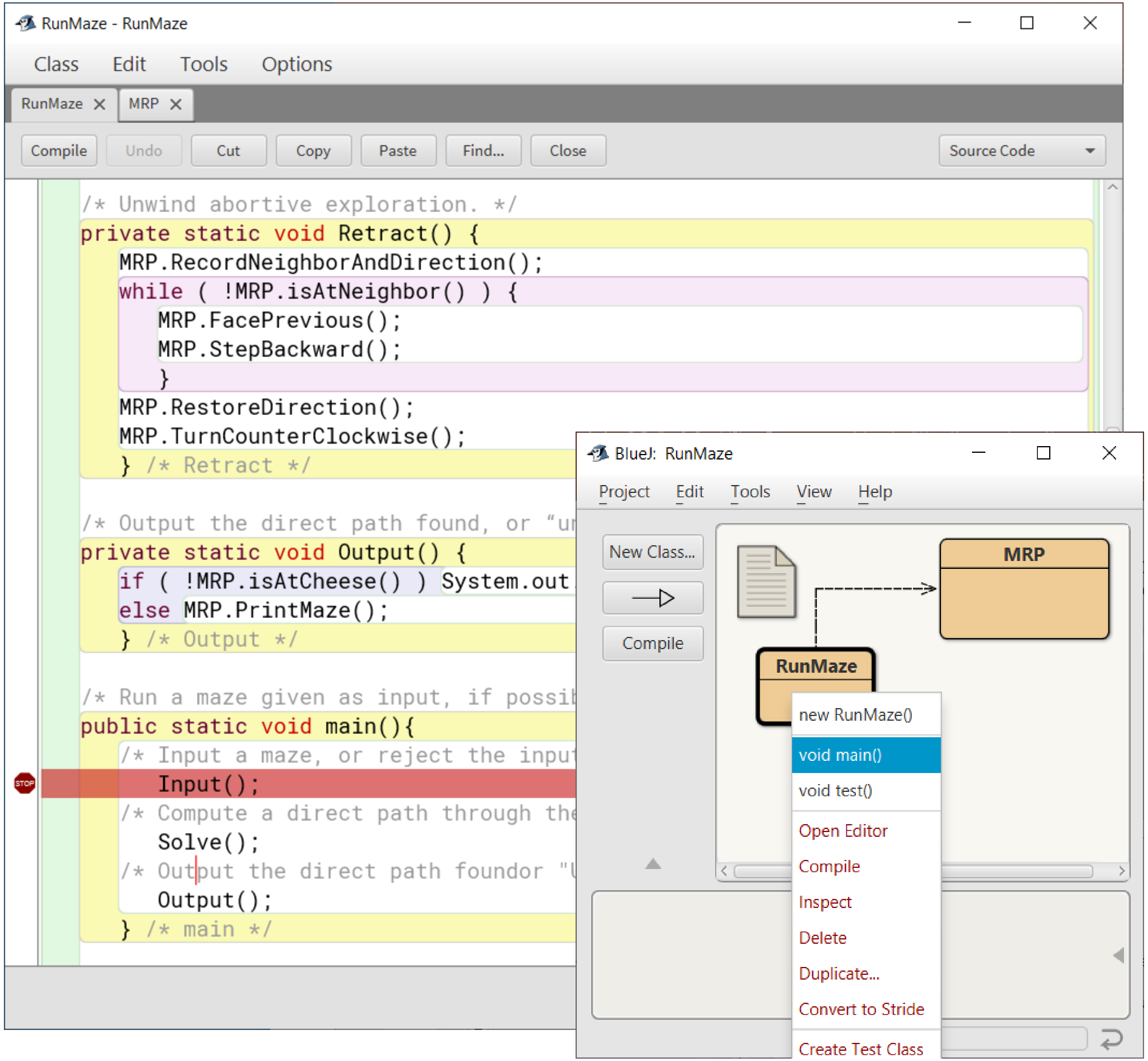
Breakpoints

A *breakpoint* is a region of code identified as a stopping point of interest.

Setting appropriate breakpoints allows execution to proceed full speed ahead, but guarantees that the user will regain control in the debugger whenever execution reaches one of the designated points of interest.

Here, we have set a breakpoint on the first line of method `main`, at a call to `Input`. We did this by clicking in the margin, where the stop sign appeared.

We fire up program execution by right-clicking on RunMaze in the project diagram, and selecting void main() from the popup menu that appears.



Breakpoints

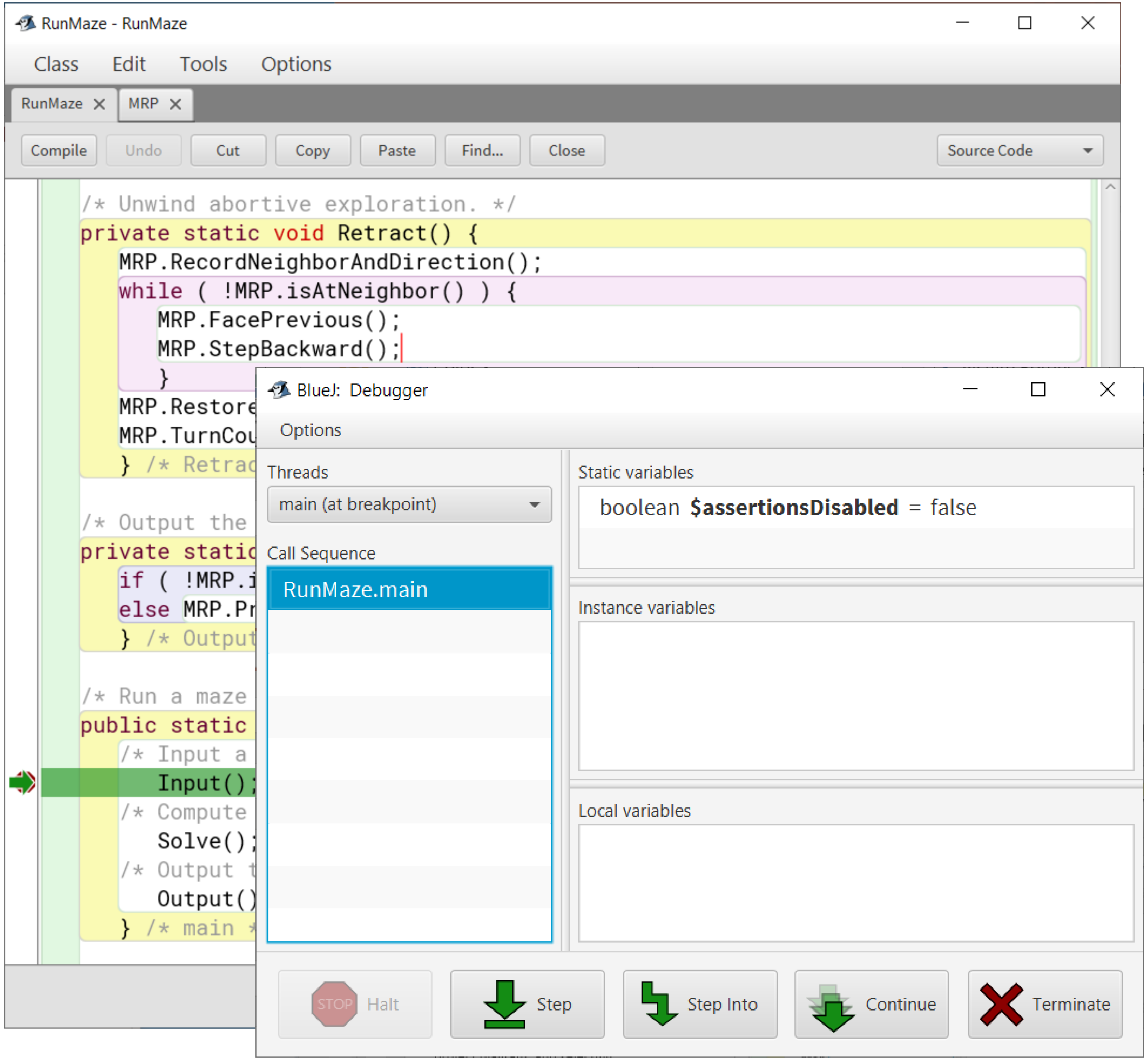
A *breakpoint* is a region of code identified as a stopping point of interest.

Setting appropriate breakpoints allows execution to proceed full speed ahead, but guarantees that the user will regain control in the debugger whenever execution reaches one of the designated points of interest.

Here, we have set a breakpoint on the first line of method `main`, at a call to `Input`. We did this by clicking in the margin, where the stop sign appeared.

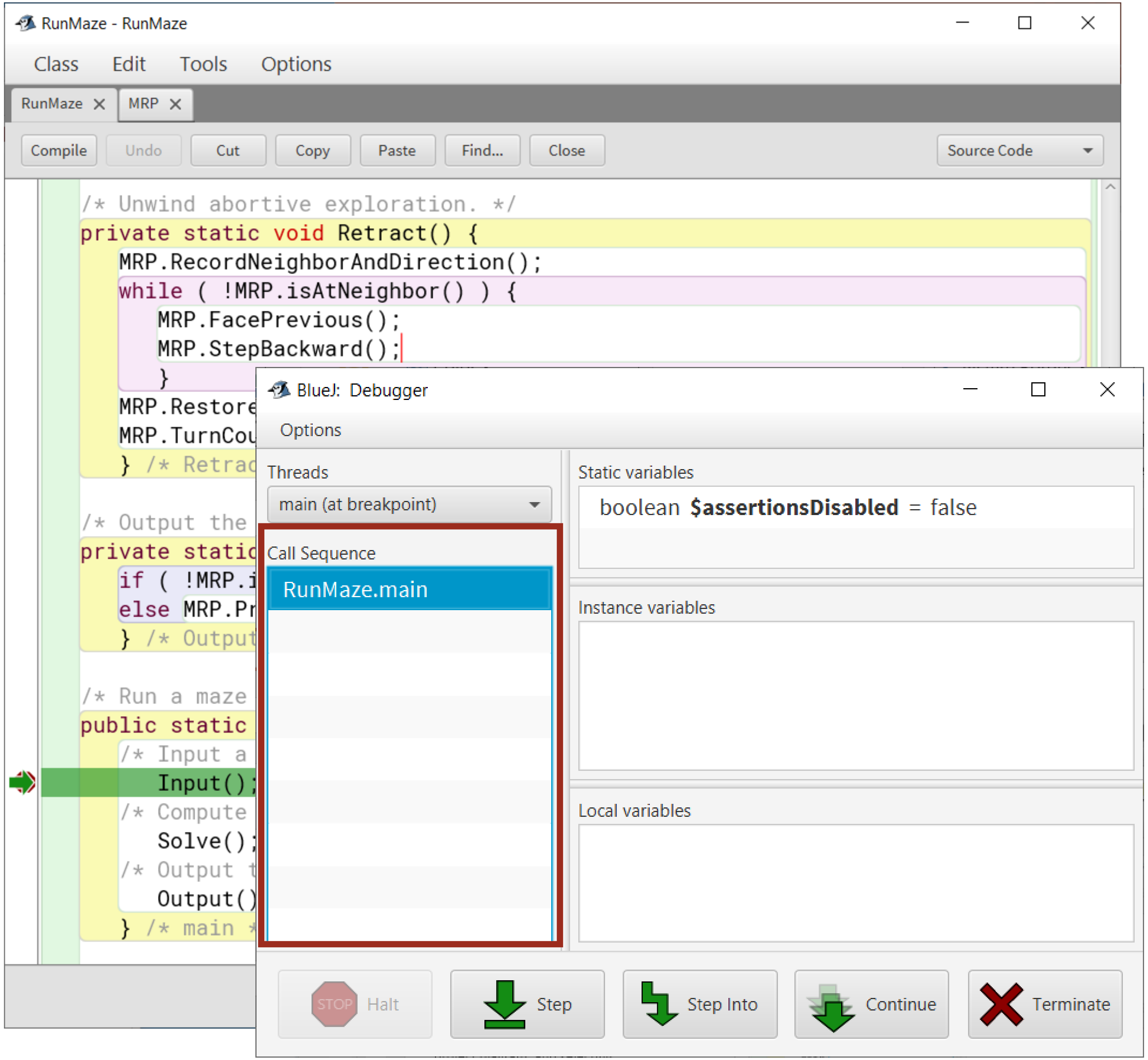
We fire up program execution by right-clicking on RunMaze in the project diagram, and selecting void main() from the popup menu that appears.

On reaching the breakpoint, we regain control in the debugger.



Control Panel

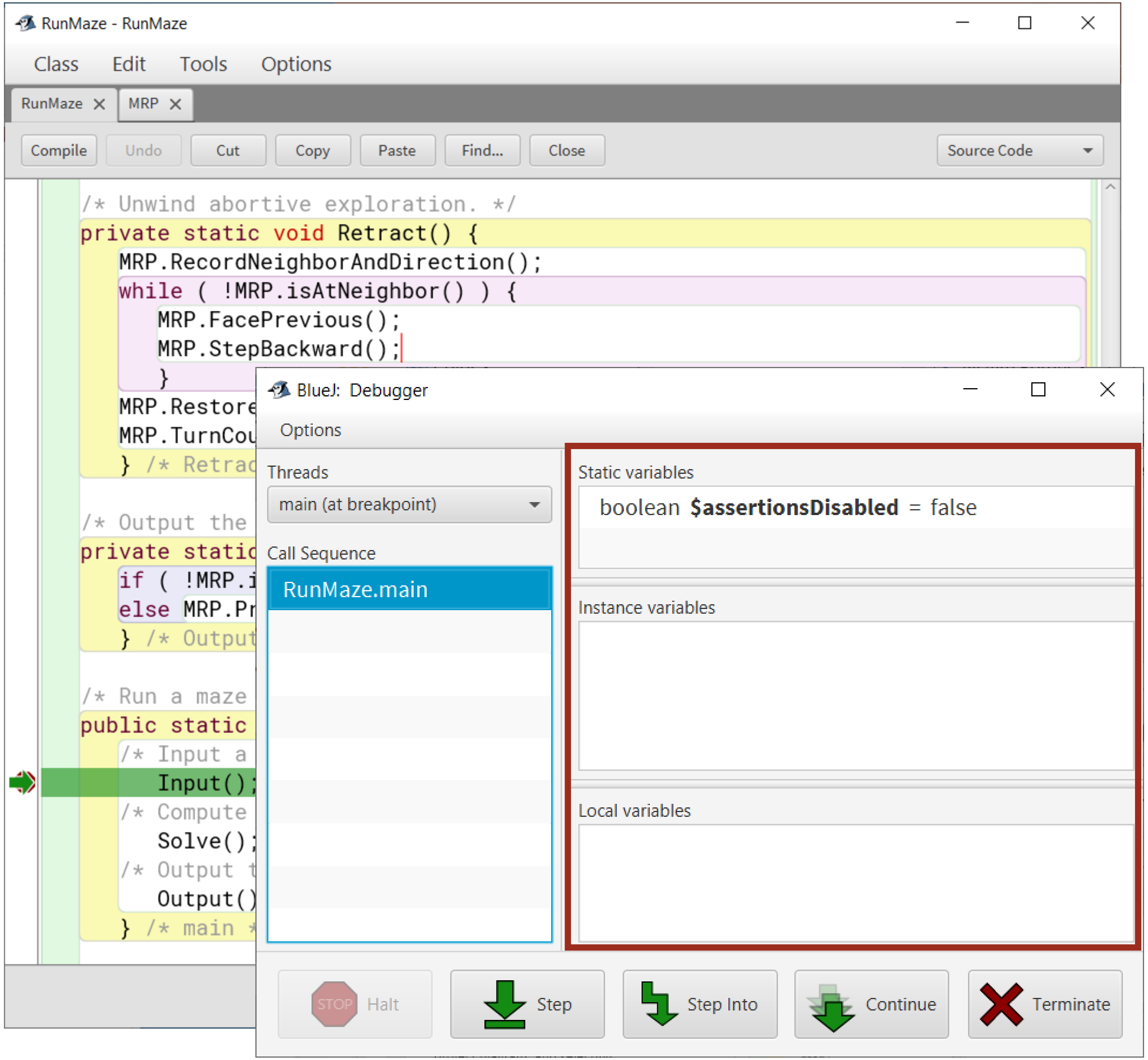
The debugger's *control panel* has a region for the display of the current call stack



Debuggers

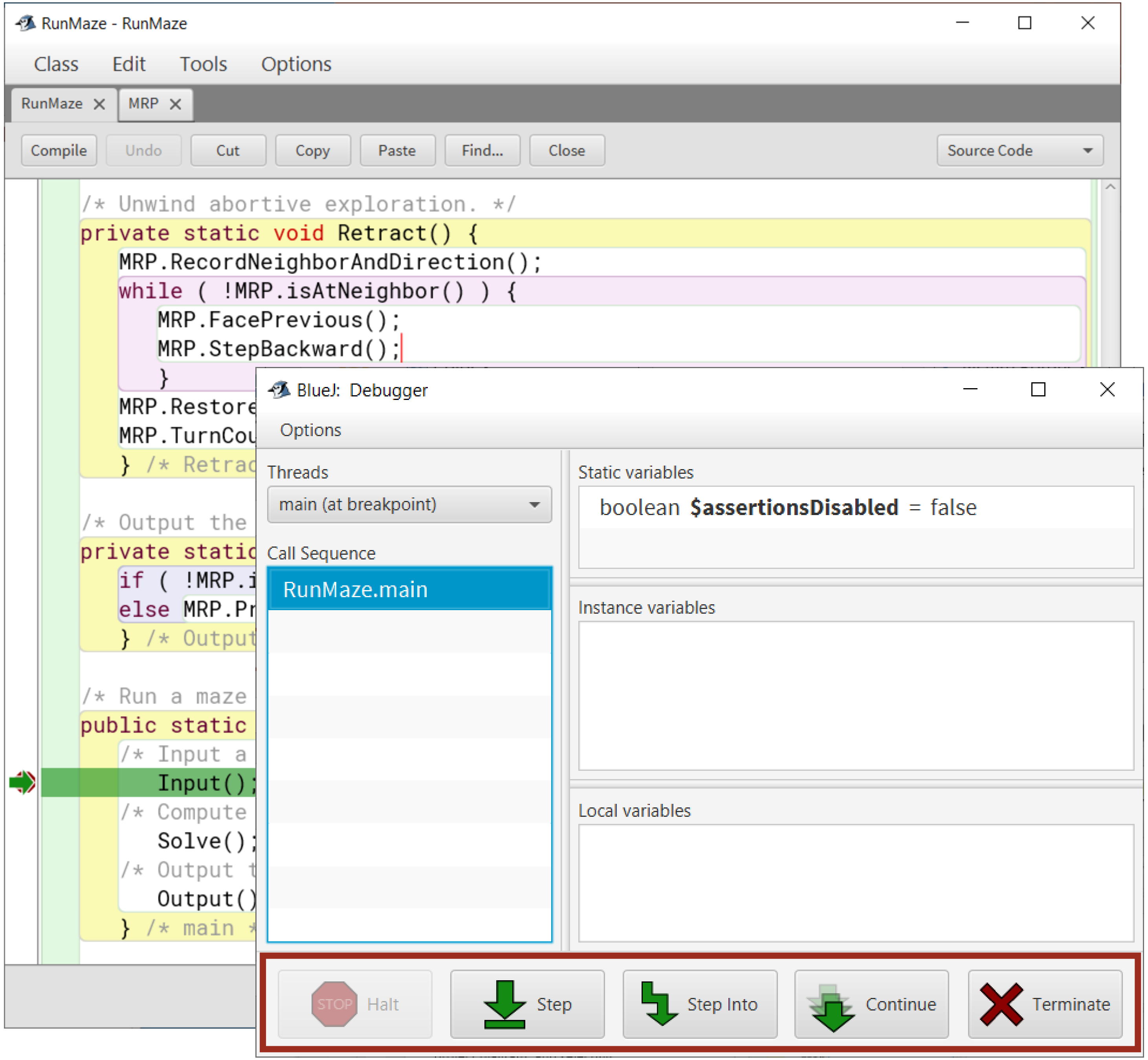
Control Panel

The debugger's *control panel* has a region for the display of the current call stack, program variables



Control Panel

The debugger's *control panel* has a region for the display of the current call stack, program variables, and buttons for manual control of the pace of subsequent execution steps.

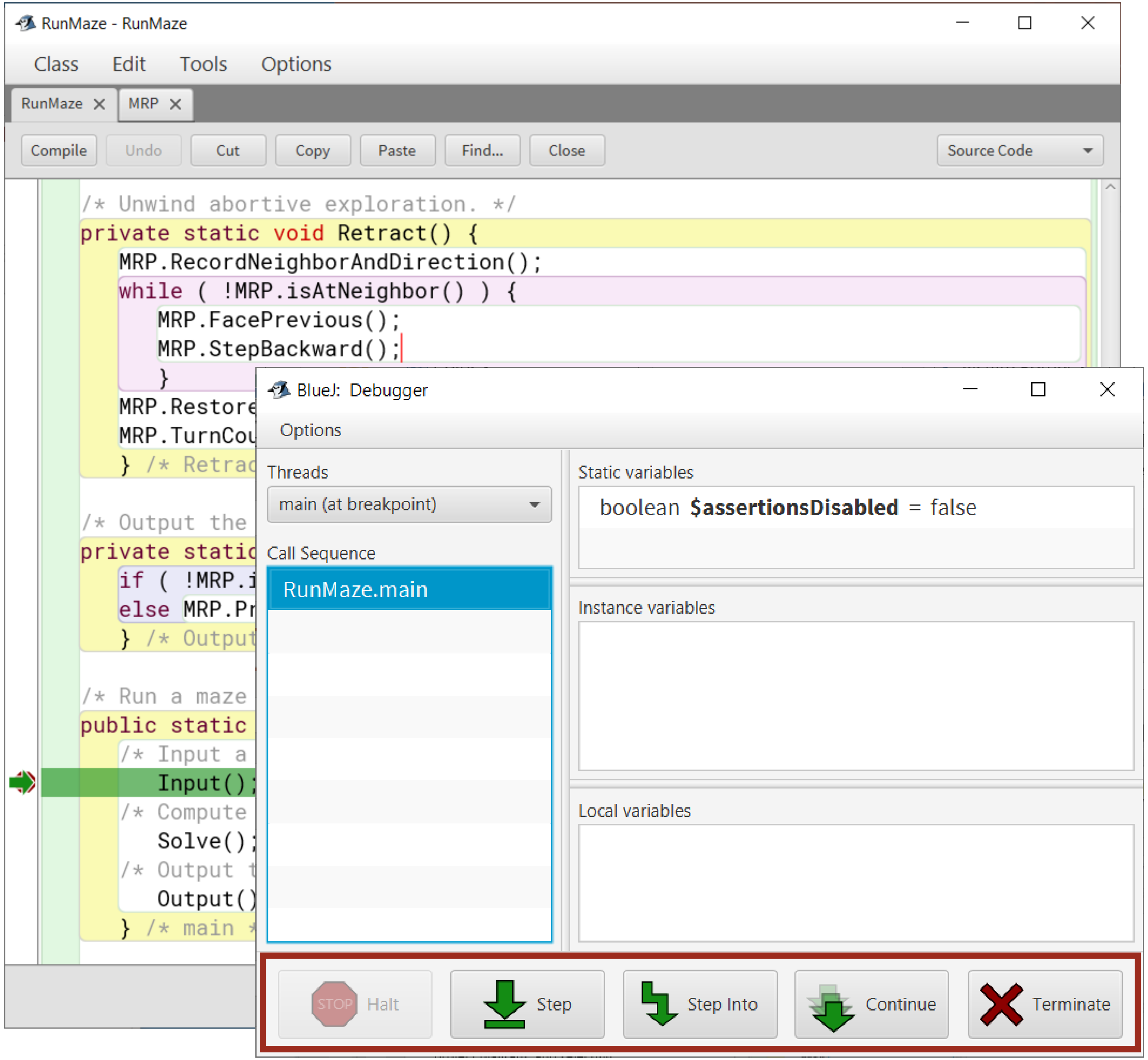


Control Panel

The debugger's *control panel* has a region for the display of the current call stack, program variables, and buttons for manual control of the pace of subsequent execution steps.

The controls of immediate interest are:

- **Step (Over}**
- **Step Into**
- **Continue**



Control Panel

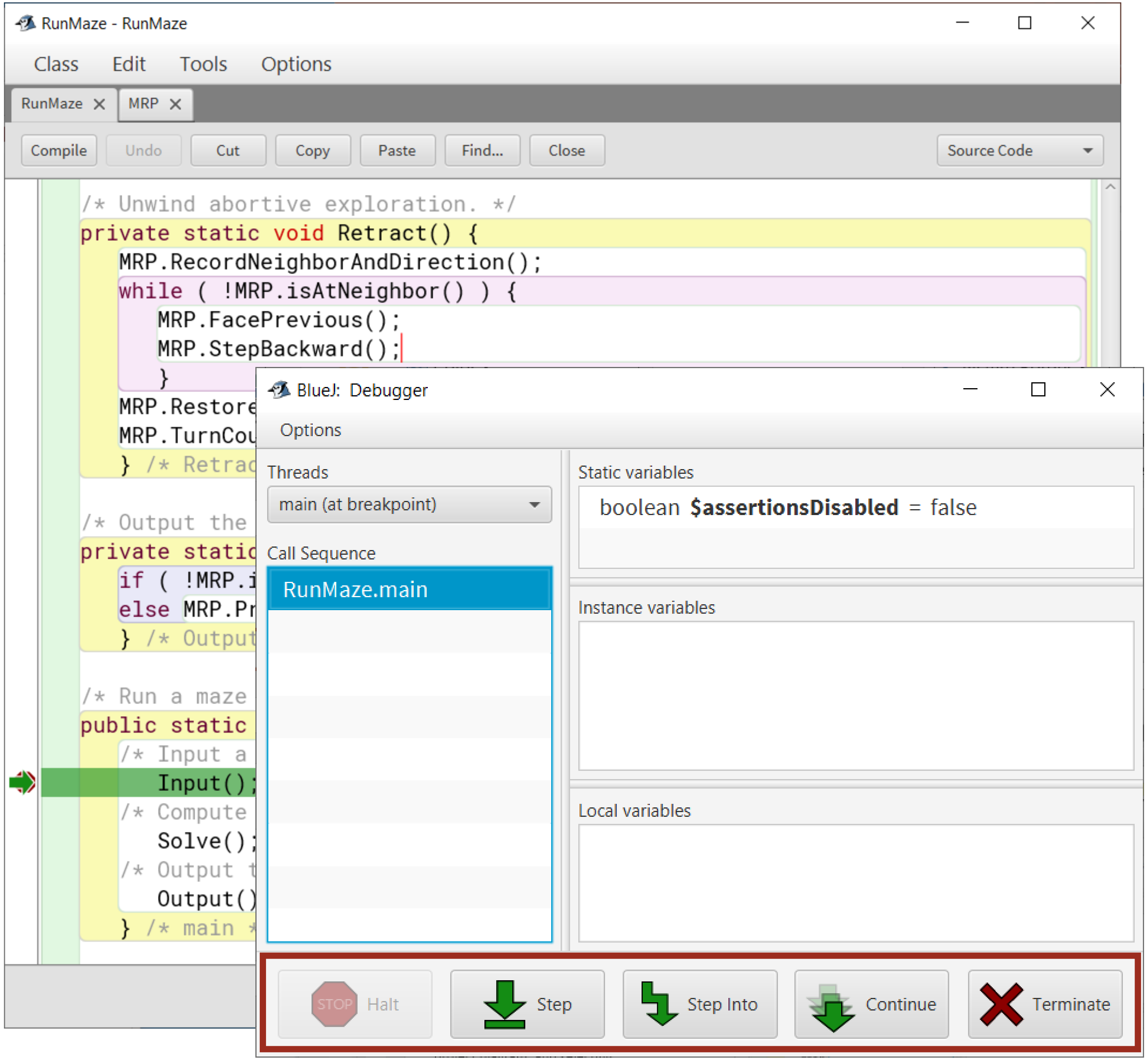
The debugger's *control panel* has a region for the display of the current call stack, program variables, and buttons for manual control of the pace of subsequent execution steps.

The controls of immediate interest are:

- **Step (Over)**
- **Step Into**
- **Continue**

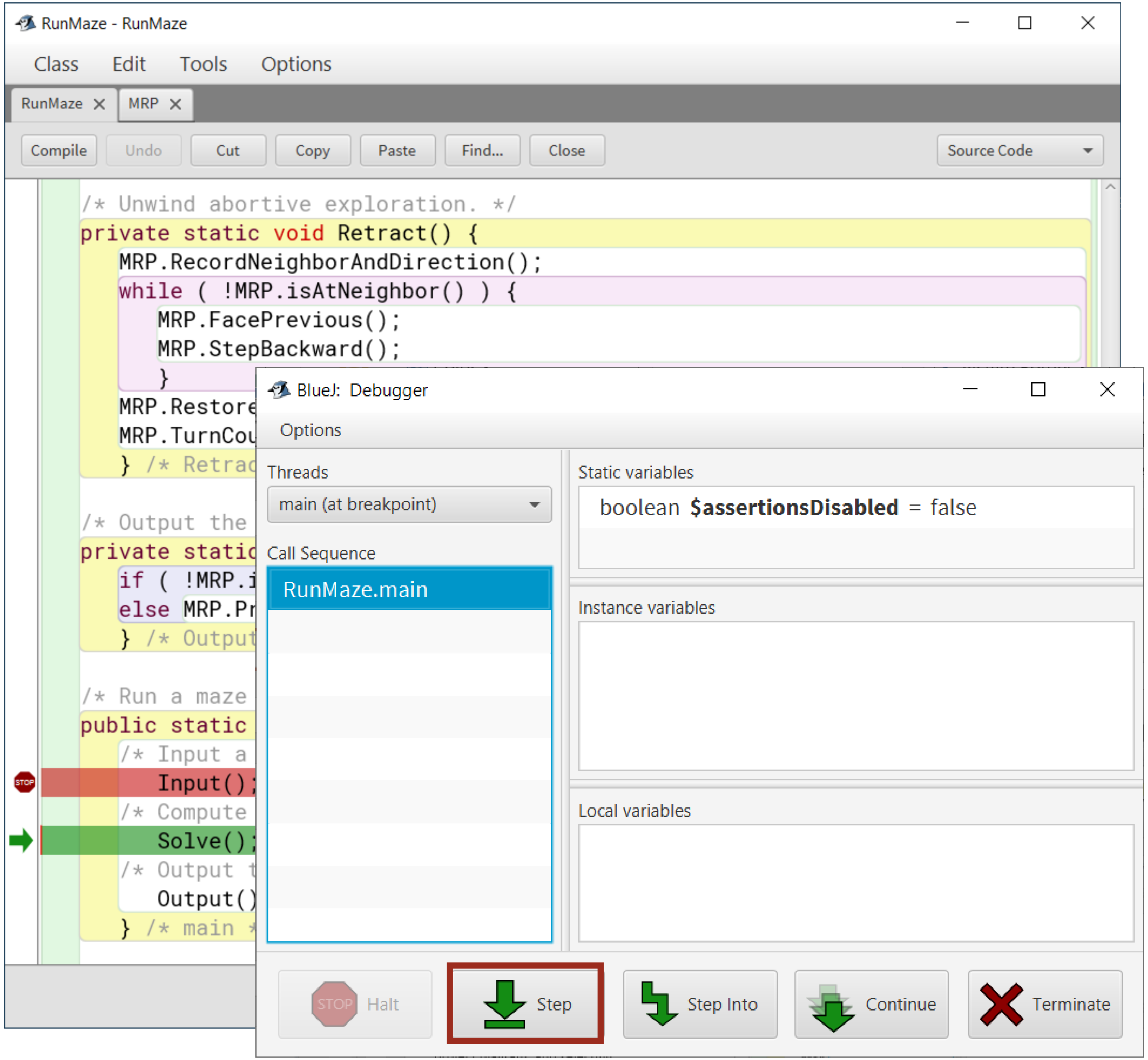
meaning:

- **Step (Over).** Execute the current line all in one step; then return to the debugger.
- **Step Into.** Advance execution to the first line of code *within* the designated statement.
- **Continue.** Proceed at top speed.



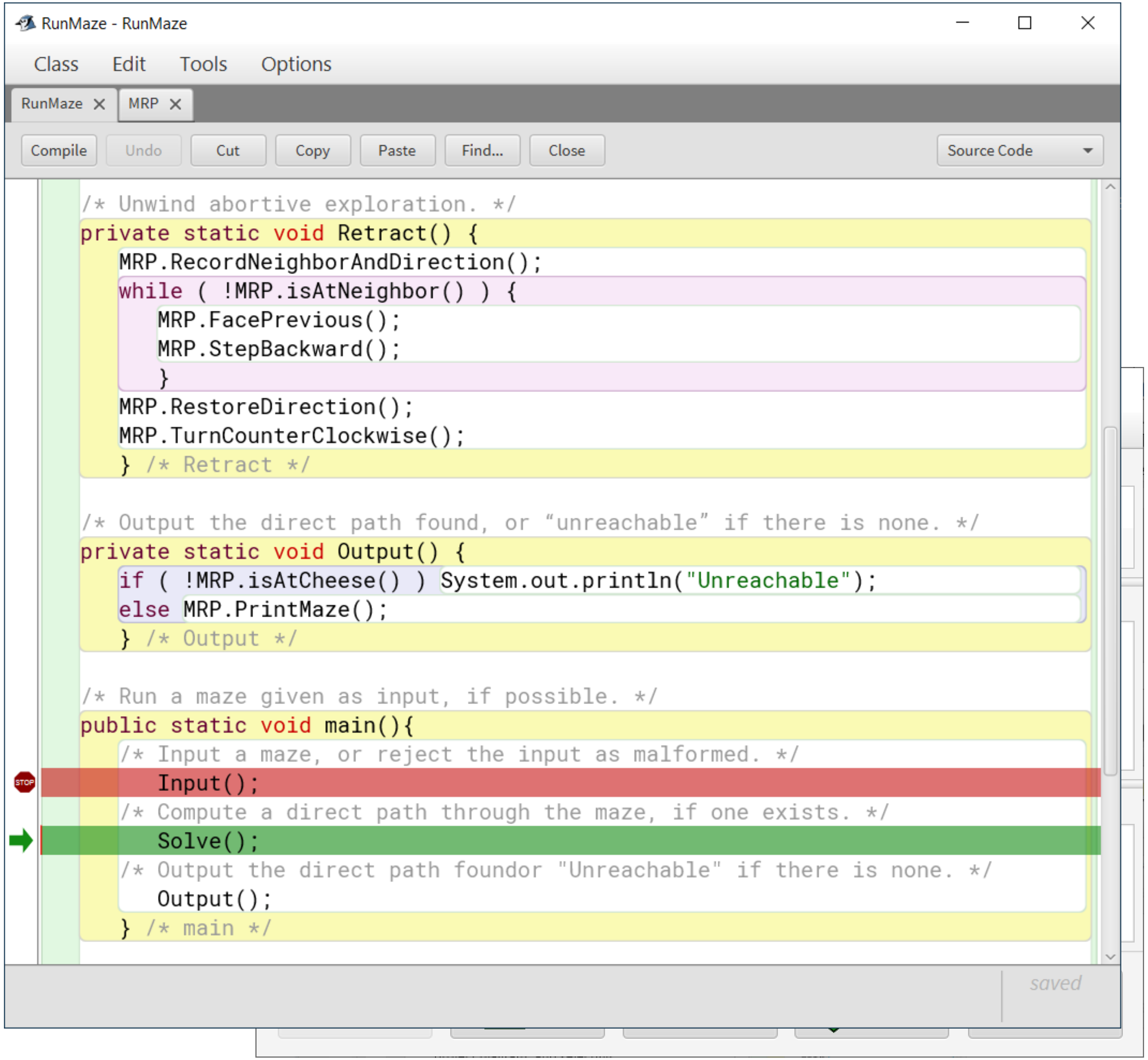
Single-step Execution

We have no current interest in the details of `Input`, so we click **Step (Over)**



Single-step Execution

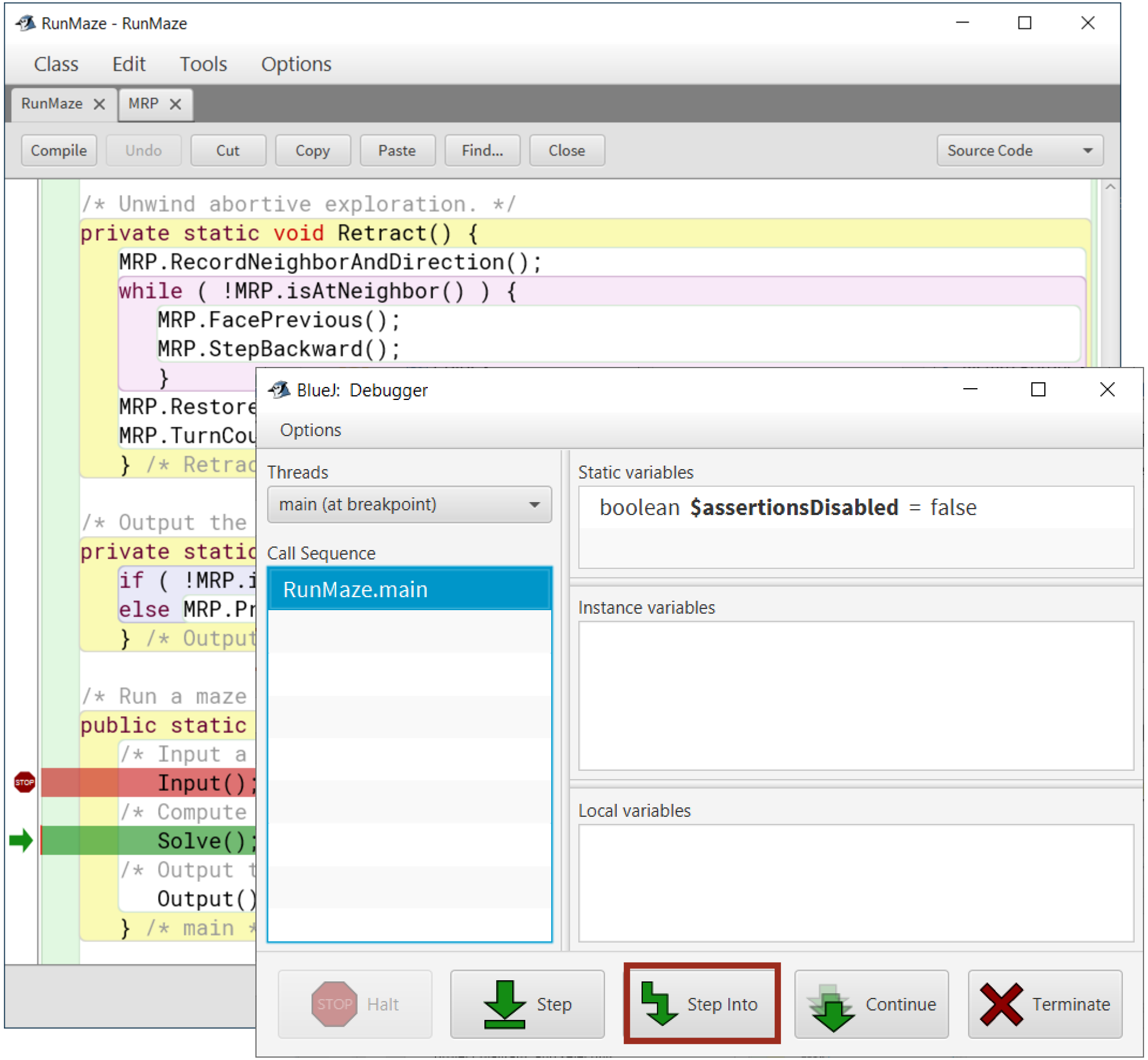
We have no current interest in the details of `Input`, so we click **Step (Over),** which brings us to the second statement in `main`, the call to `Solve`.



Single-step Execution

We have no current interest in the details of `Input`, so we click **Step (Over),** which brings us to the second statement in `main`, the call to `Solve`.

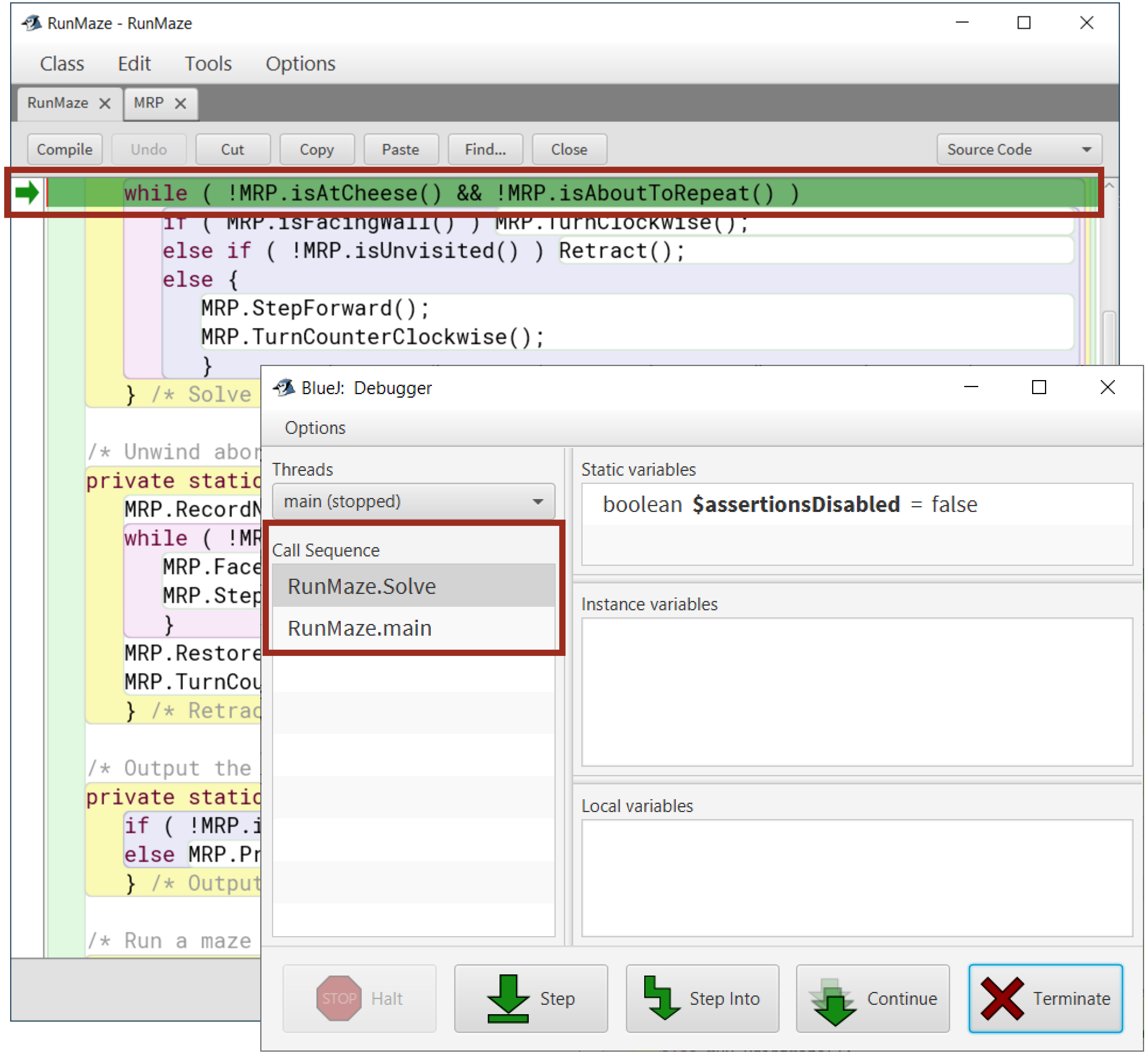
Next, we wish to inspect execution within `Solve` in fine-grained detail, so we click **Step Into**.



Single-step Execution

We have no current interest in the details of `Input`, so we click **Step (Over),** which brings us to the second statement in `main`, the call to `Solve`.

Next, we wish to inspect execution within `Solve` in fine-grained detail, so we click **Step Into**, which brings us to that method's first statement.



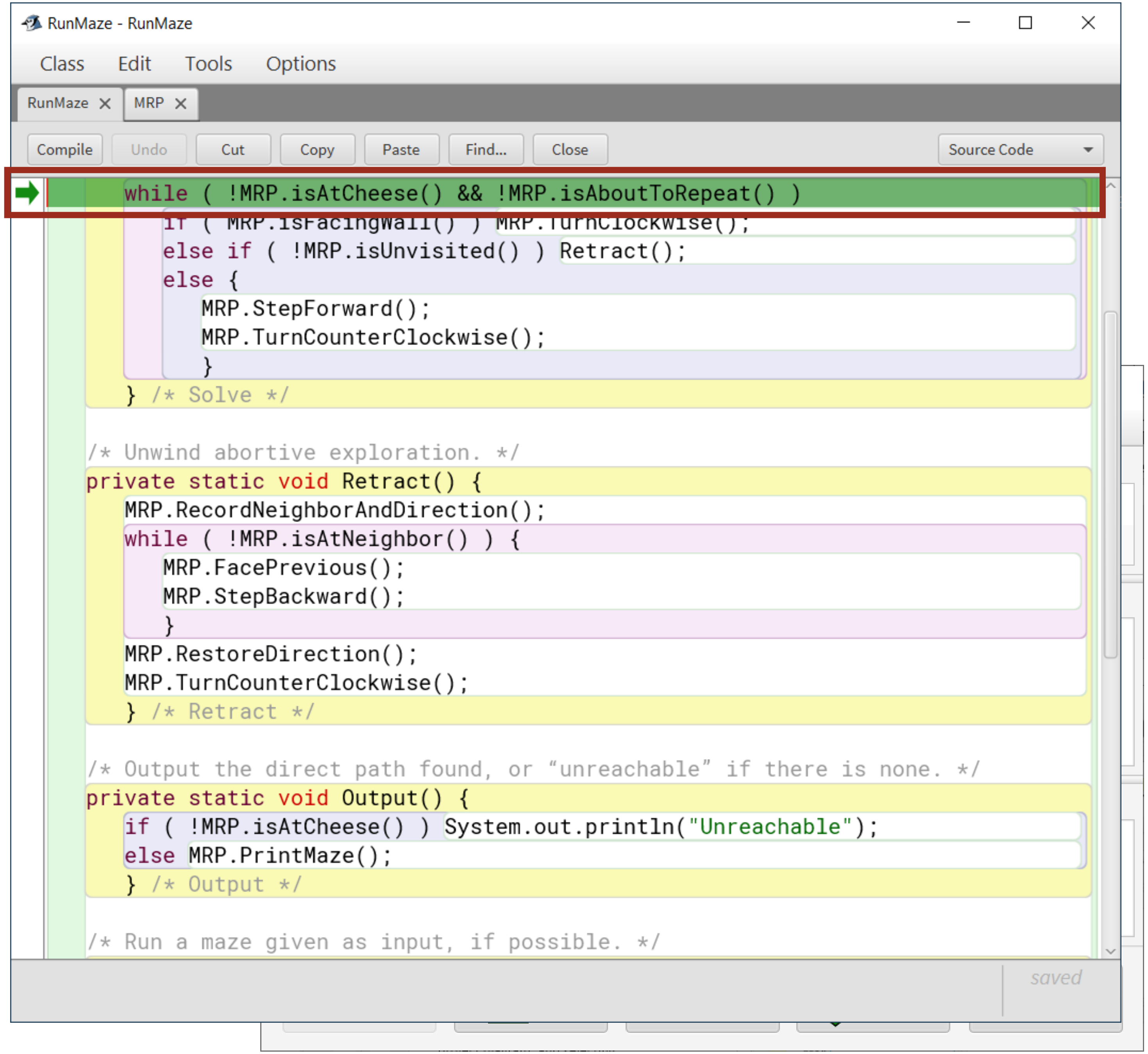
Single-step Execution

We have no current interest in the details of `Input`, so we click **Step (Over),** which brings us to the second statement in `main`, the call to `Solve`.

Next, we wish to inspect execution within `Solve` in fine-grained detail, so we click **Step Into**, which brings us to that method's first statement.

Suppose, now, that we are working on Bug A, and are trying to understand why the rat fails to find a path to the cheese.

Recall that the mistake in Bug A was an error in method `isFacingWall`.



Single-step Execution

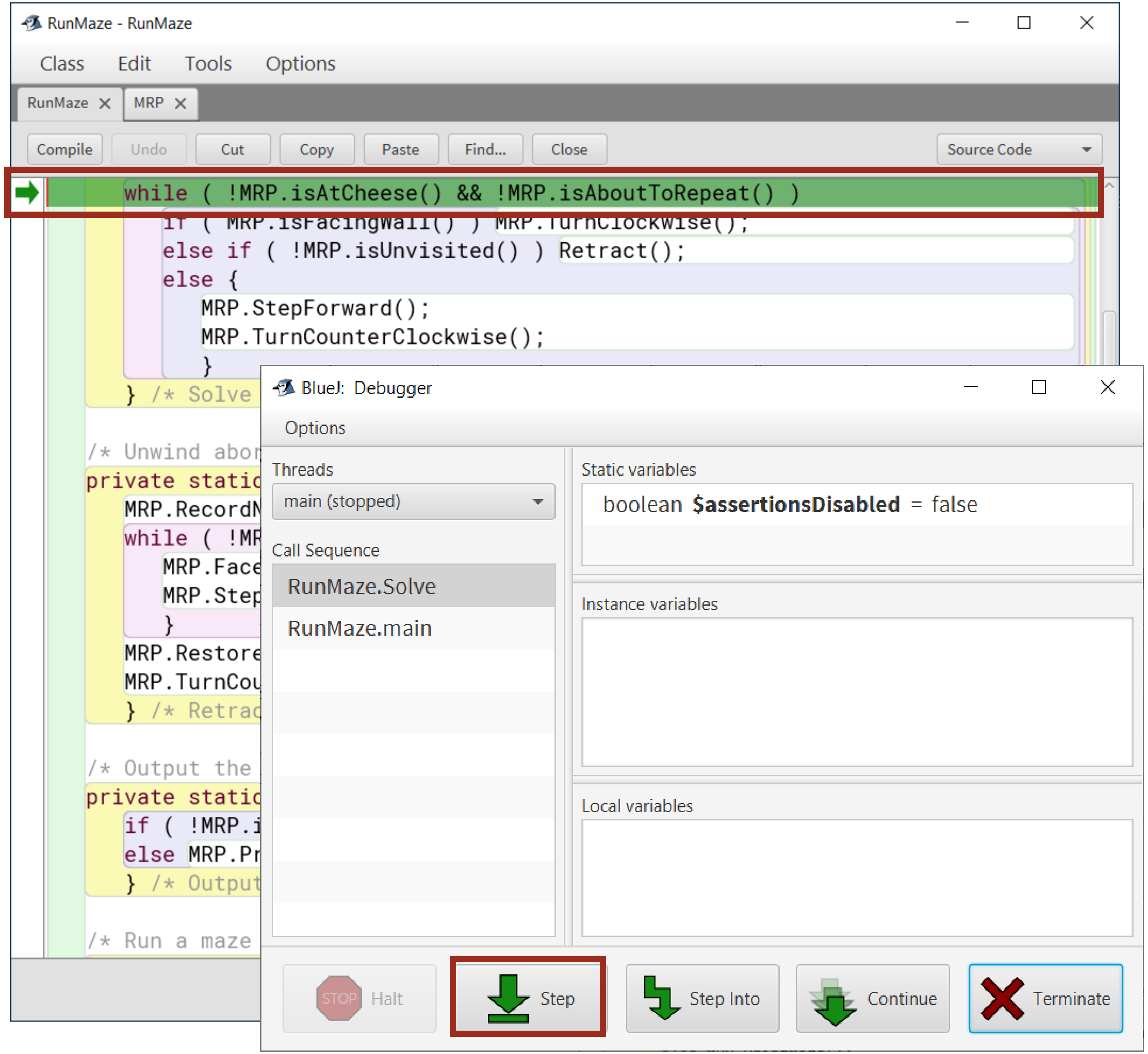
We have no current interest in the details of `Input`, so we click **Step (Over),** which brings us to the second statement in `main`, the call to `Solve`.

Next, we wish to inspect execution within `Solve` in fine-grained detail, so we click **Step Into**, which brings us to that method's first statement.

Suppose, now, that we are working on Bug A, and are trying to understand why the rat fails to find a path to the cheese.

Recall that the mistake in Bug A was an error in method `isFacingWall`.

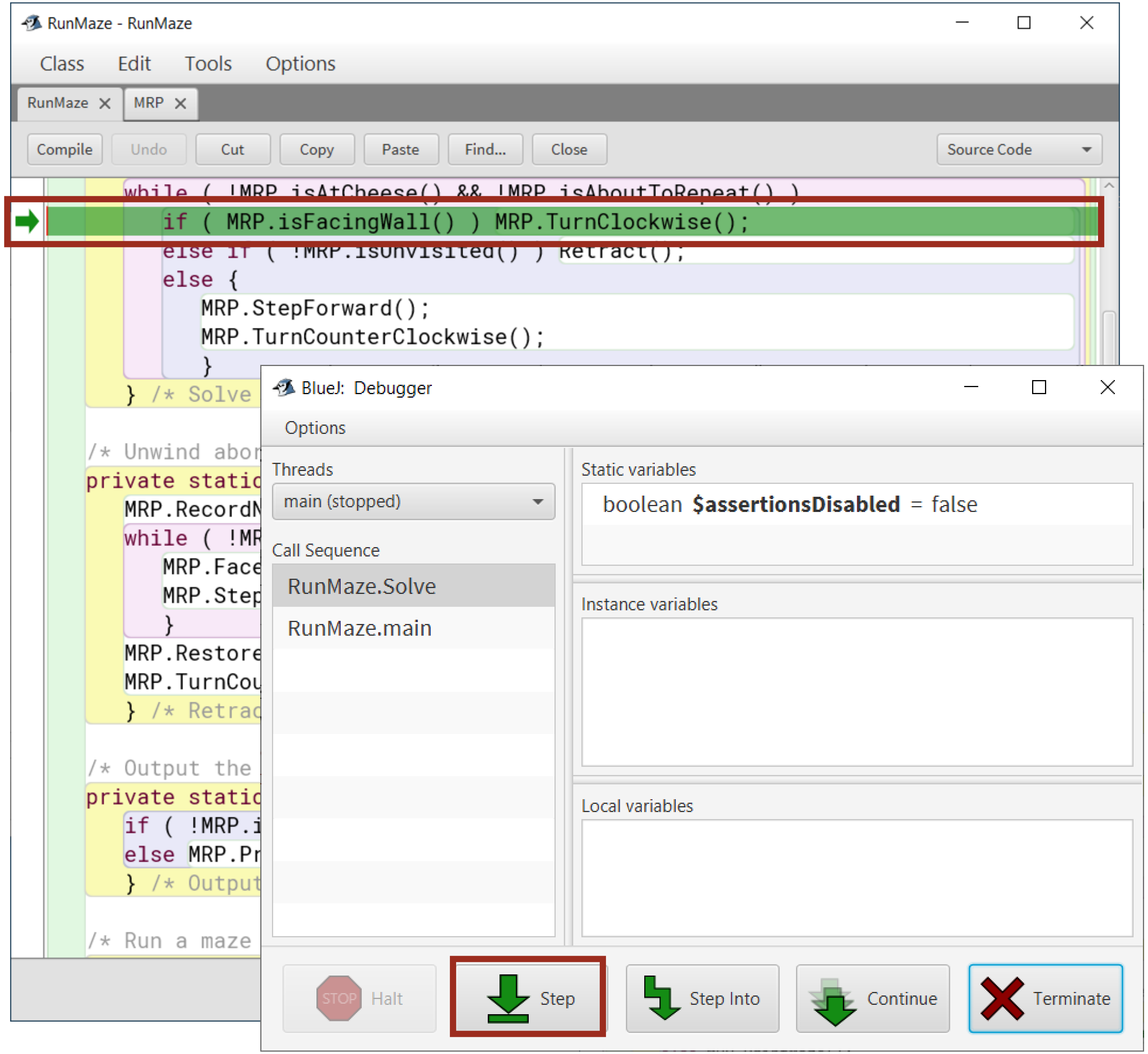
We repeatedly click **Step (Over)**, and watch the loop iterate, eventually three times.



Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

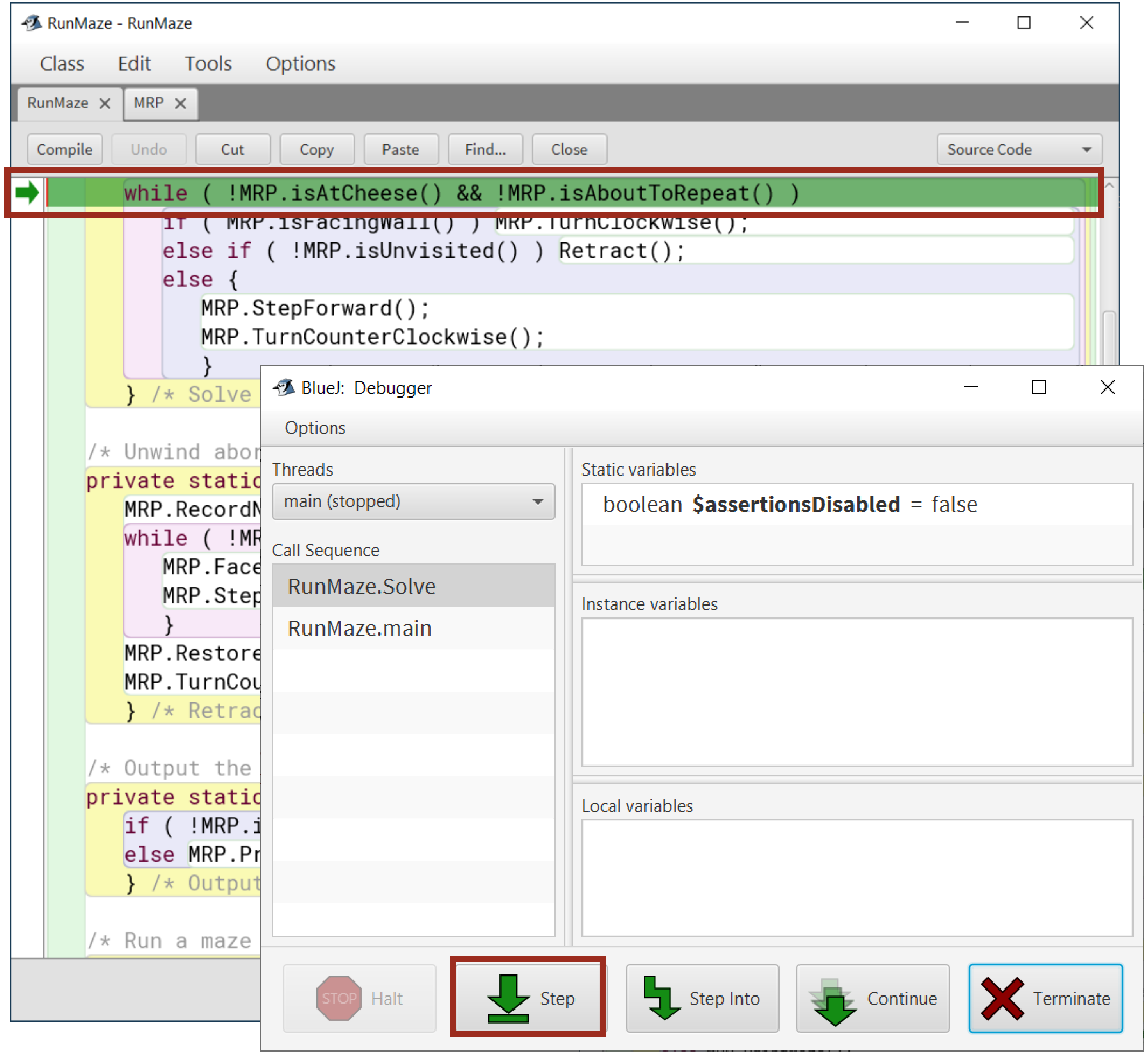
- First, from facing up to facing right.



Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

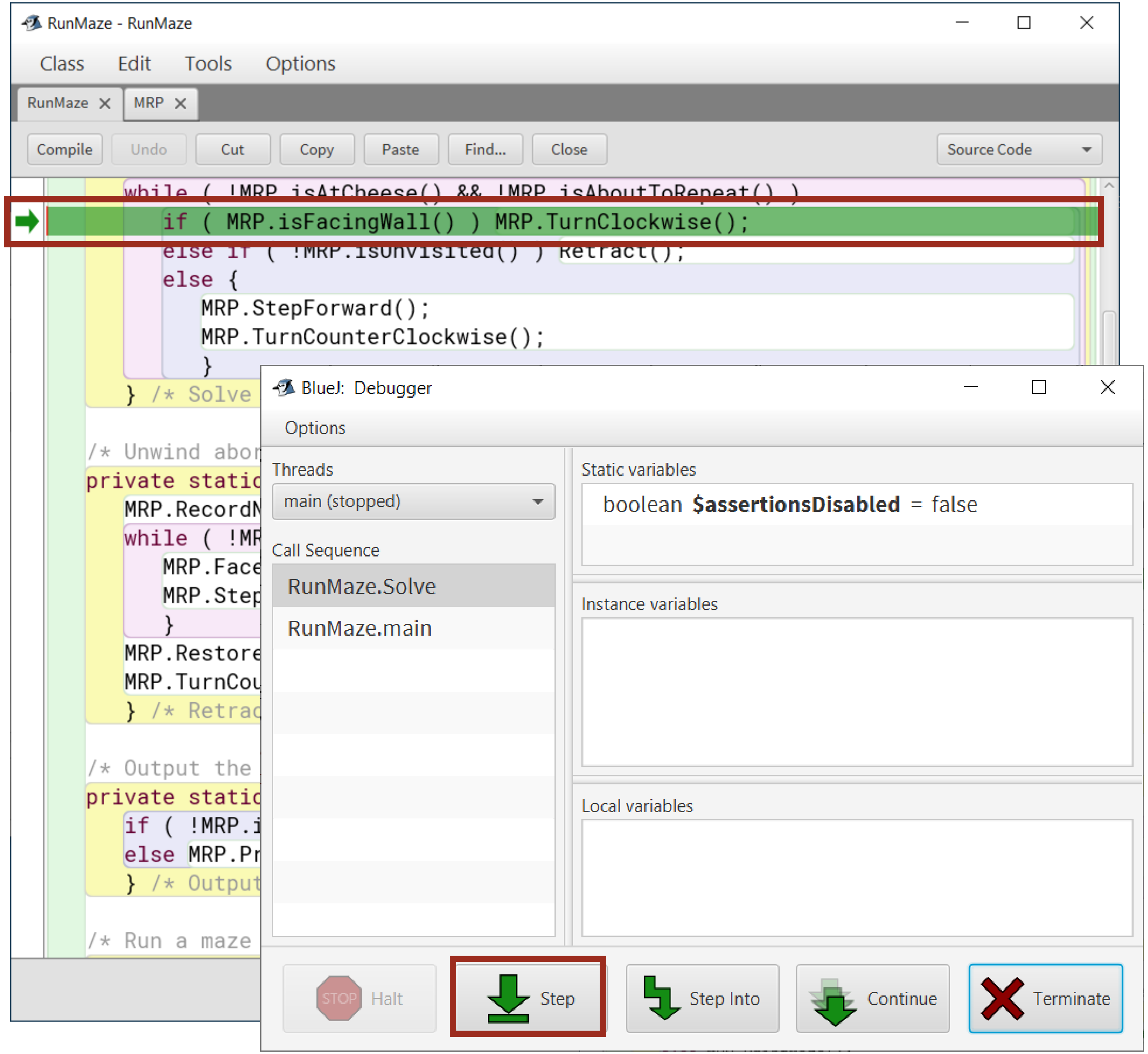
- First, from facing up to facing right.



Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

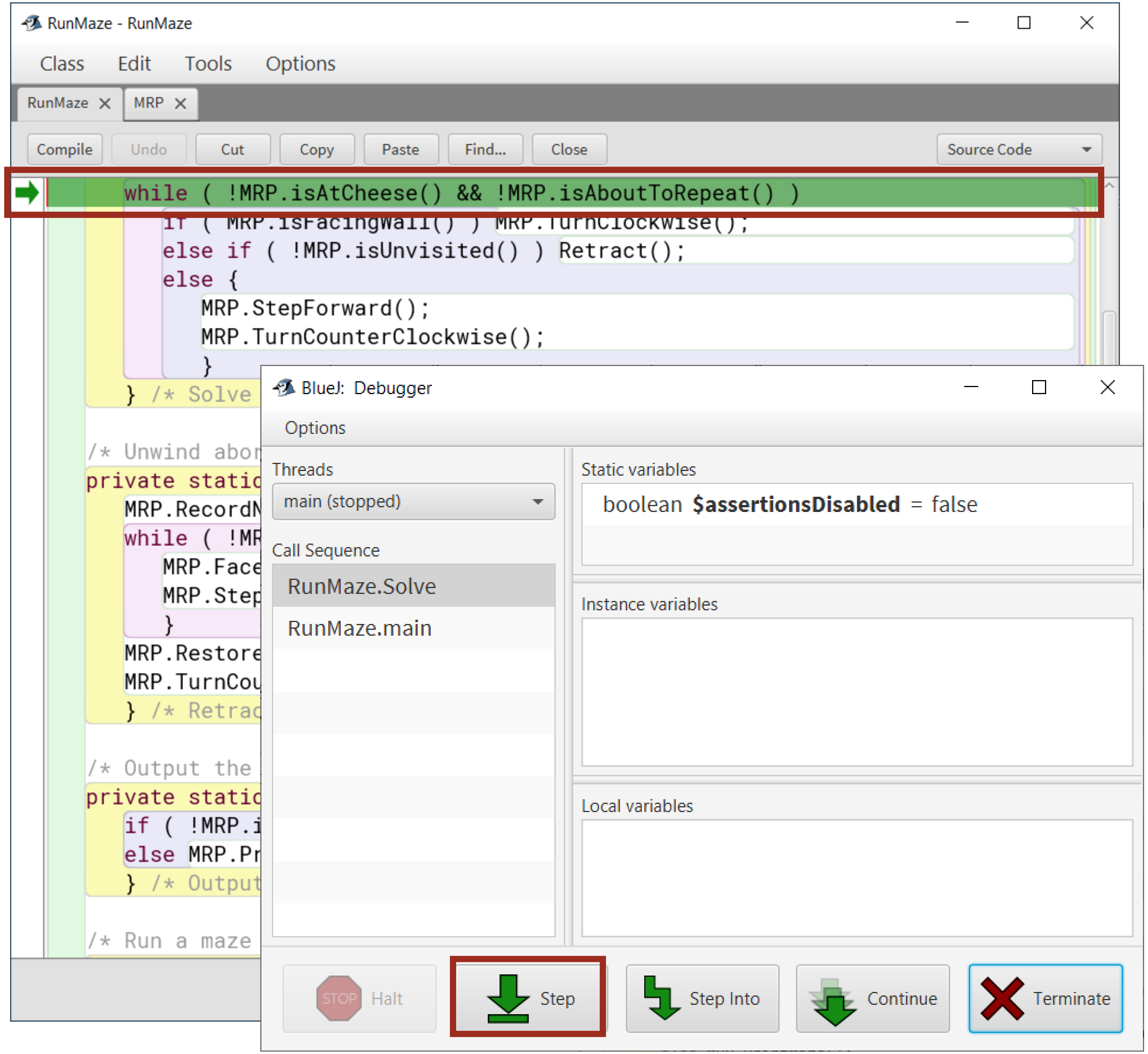
- First, from facing up to facing right.
- Second, from facing right to facing down.



Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

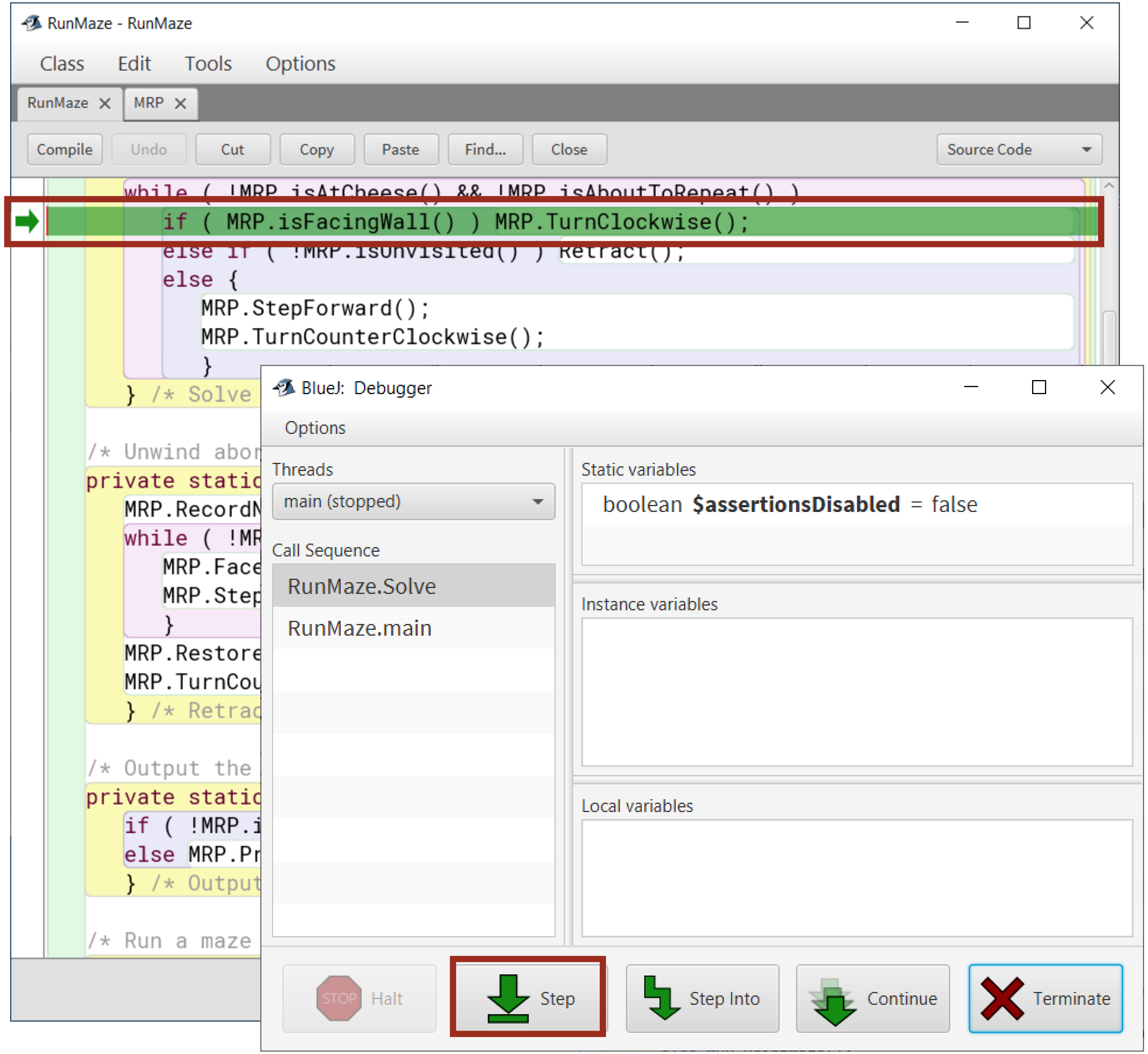
- First, from facing up to facing right.
- Second, from facing right to facing down.



Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

- First, from facing up to facing right.
- Second, from facing right to facing down.
- Third, from facing down to facing left.

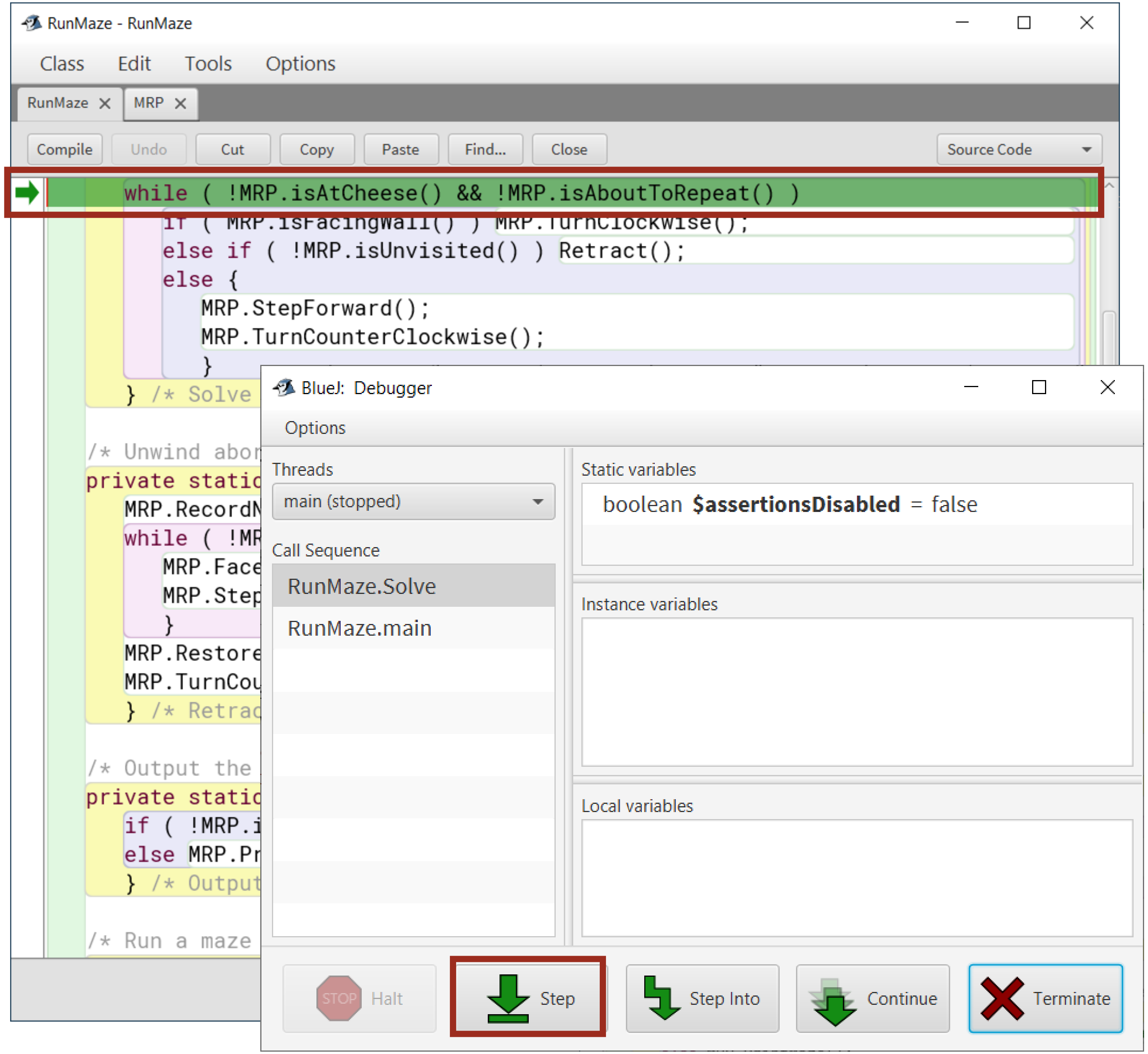


Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

- First, from facing up to facing right.
- Second, from facing right to facing down.
- Third, from facing down to facing left.

A few more clicks and the loop terminates, the call to Solve terminates, and we are done trying to find a path to the cheese.

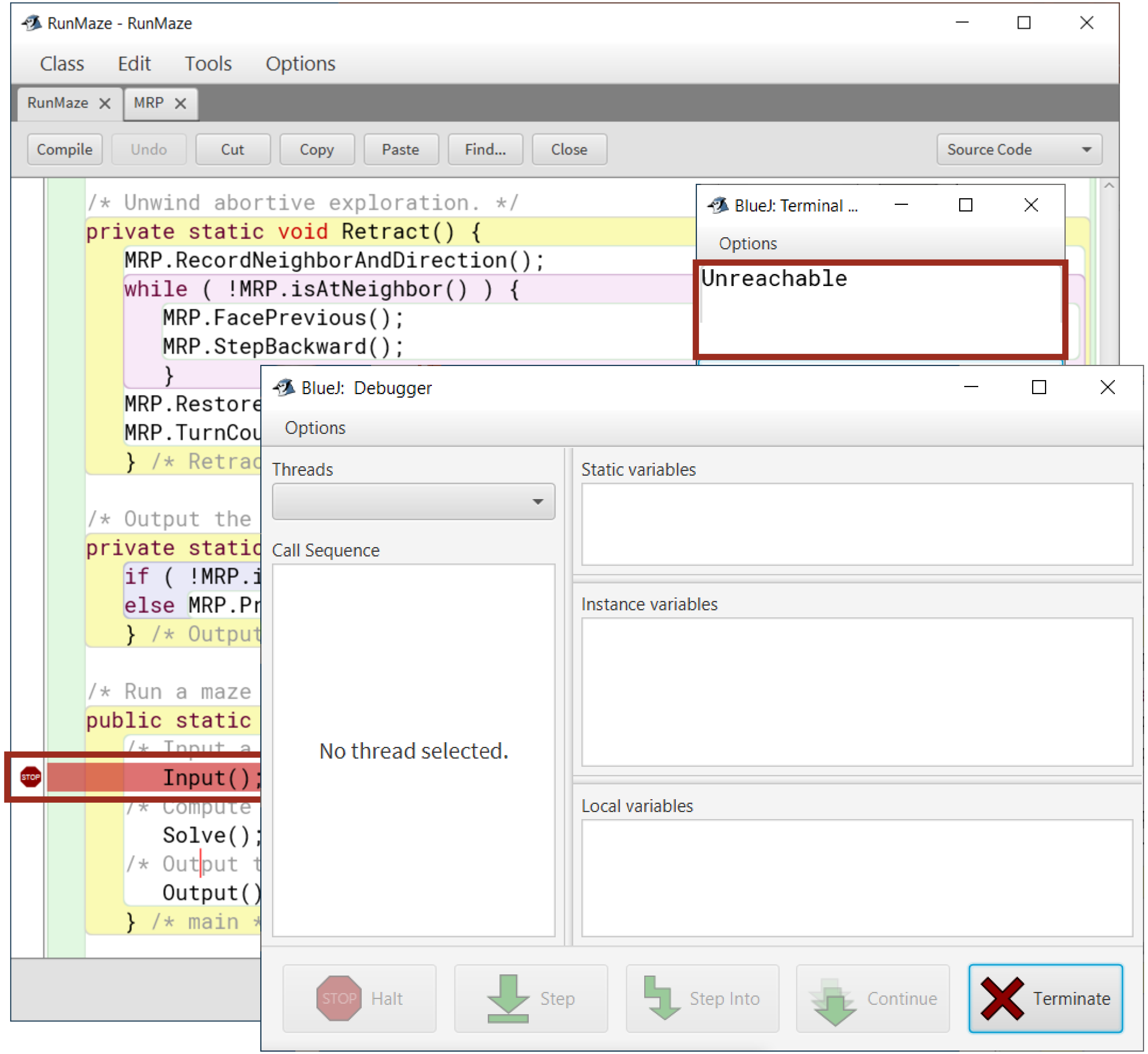


Bug A

Each time that we are not at the cheese, and are not about to repeat the traversal all over again, we ask whether we are facing a wall, and seeing none, make a clockwise turn:

- First, from facing up to facing right.
- Second, from facing right to facing down.
- Third, from facing down to facing left.

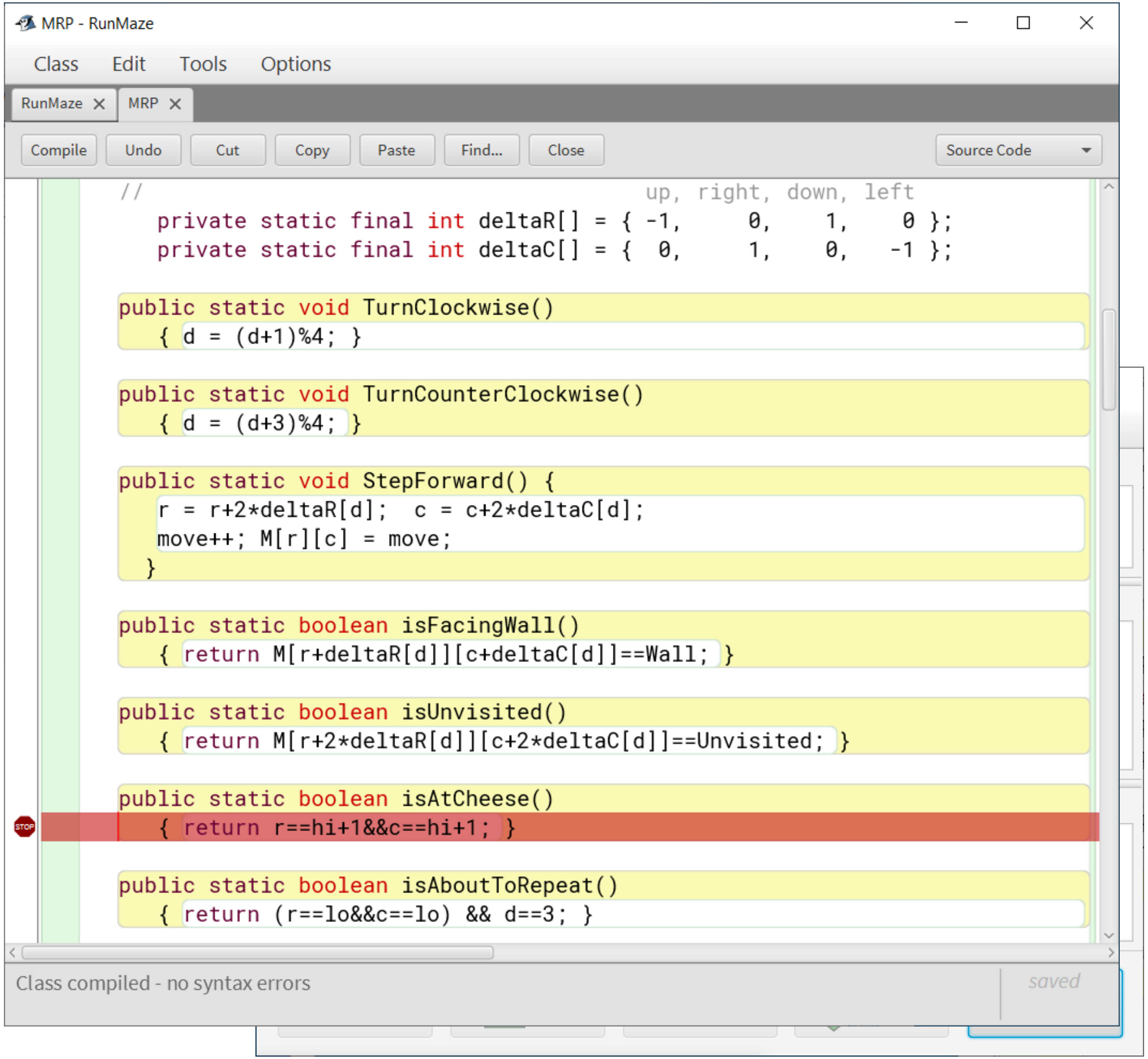
A few more clicks and the loop terminates, the call to Solve terminates, and we are done trying to find a path to the cheese. The program prints “Unreachable”, and stops.



Bug B

Recall that Bug B caused the rat to blow right by the cheese in the lower right cell, and eventually return to the upper-left cell, whereupon as in Bug A it prints “Unreachable” and stops.

Fine-grained single-step execution in this case gets tedious. We can accelerate it by setting a breakpoint at method `isAtCheese`, and then execution just stop there.

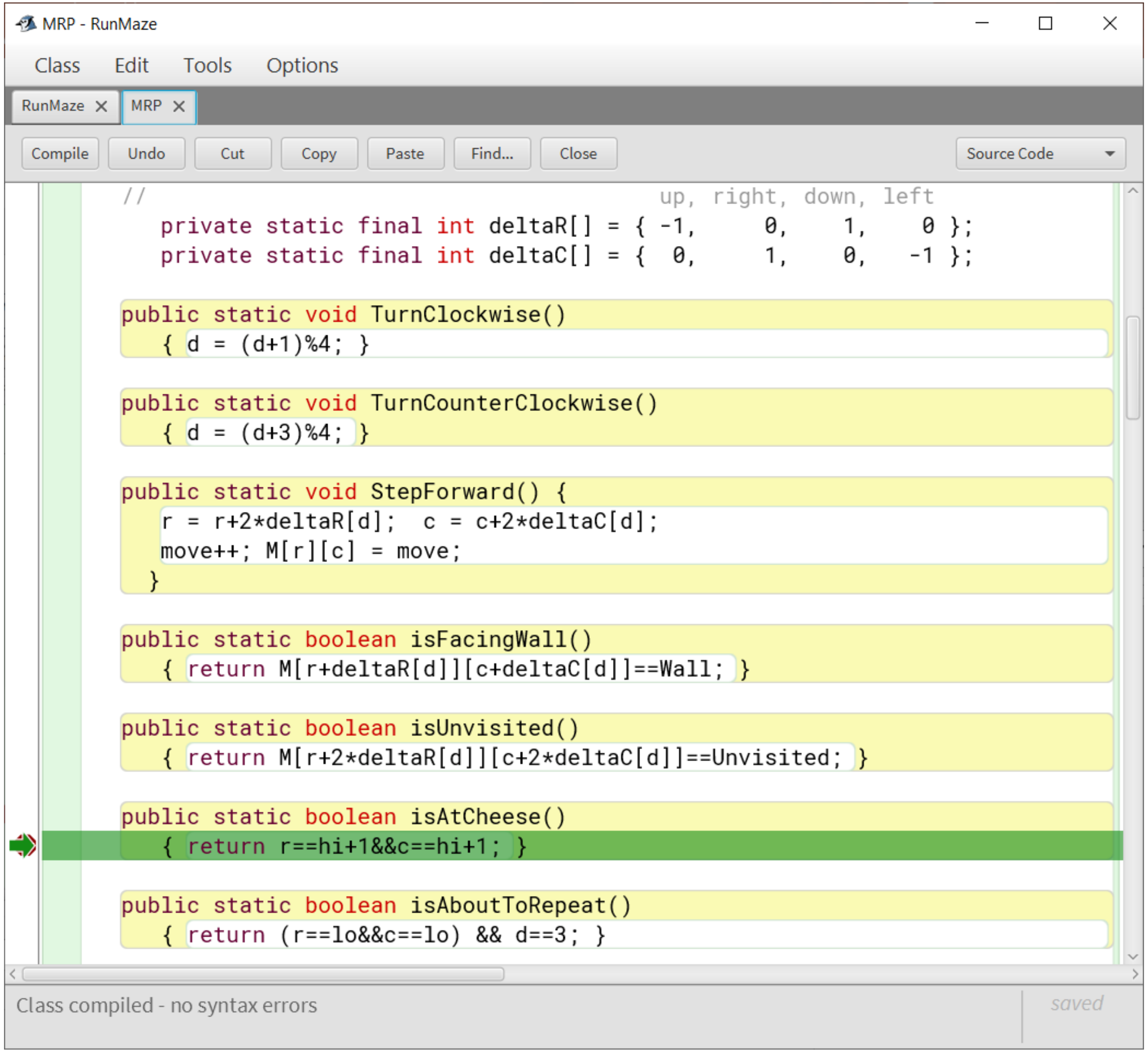


Bug B

Recall that Bug B caused the rat to blow right by the cheese in the lower right cell, and eventually return to the upper-left cell, whereupon as in Bug A it prints “Unreachable” and stops.

Fine-grained single-step execution in this case gets tedious. We can accelerate it by setting a breakpoint at method `isAtCheese`, and then execution just stop there.

For each step, we just click **Continue**, and execution resumes until hitting the breakpoint again.

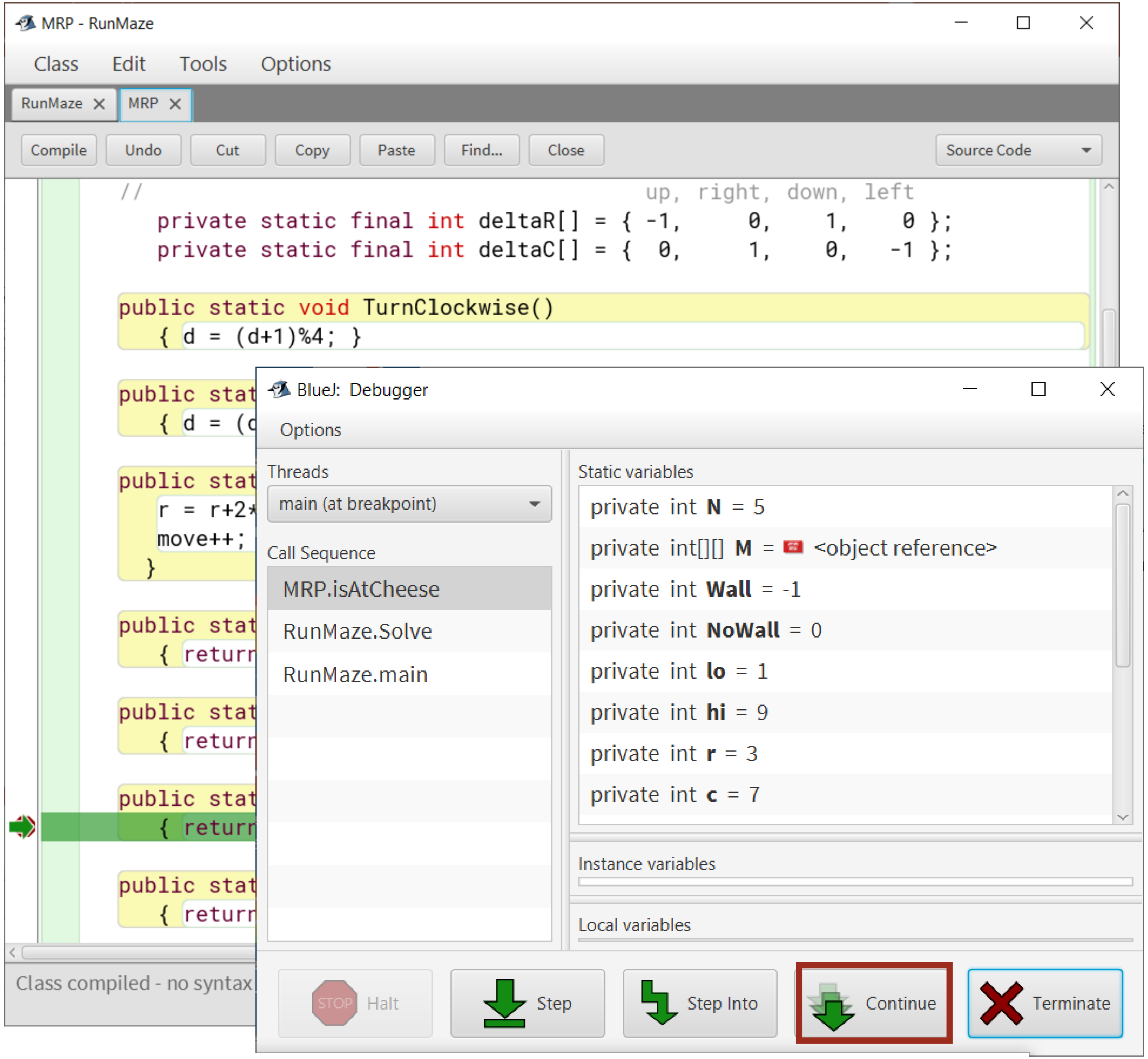


Bug B

The static variables of class MRP are displayed in the debugger's control panel each time we stop at the breakpoint in `isAtCheese`. For example, we can see that the rat has only advanced to ⟨r,c⟩ = ⟨3,7⟩, so we have a ways to go.

Each click of **Continue** resumes execution.

And so it goes.



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. 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 can do so by the aggressive use of assertions.

Assert statements were first introduced in Chapter 3 when we had scant use for them. In Chapter 15, we introduced the idea of self-checking code, and used an **assert** to signal failure of the program to meet its specification. We now advocate self-checking on a fine-grained basis (rather than just at the end of execution) in the hope of nipping bugs in the bud.

Defensive Programming: Stay in Control.

We illustrate aggressive use of asserts in our program for Running a Maze by implementing Boolean method `isValid` to check the data representation invariants 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.isFacingUnvisited()) Retract();
      else {
         MRP.StepForward();
         MRP.TurnCounterClockwise();
         }
      }
   } /* Solve */
```

Defensive Programming: Stay in Control.

Method `isValid` can be defined in MRP as:

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

where method `isValidPath` is the routine introduced into MRP in Chapter in 15 to validate the solution path, and method `isValidRat` is defined now in MRP to validate the rat's representation invariant:

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

Defensive Programming: Stay in Control.

by:

```
# Return false iff rat's representation invariant is violated.
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.

In addition to the validity check once per iteration in `Solve`, we can scatter calls to `isValid()` generously throughout the program, e.g., at the end of each method that modifies state. Were we to have done so in the flawed routine of Bug C:

```
34 public static void TurnClockwise()
35    { d = (d+1)    ; assert isValid(): "Invalid MRP representation."; }
```

the mistake would have immediately “self-reported”:

```
java.lang.AssertionError: Invalid MRP representation.
    at MRP.TurnClockwise(MRP.java:35)
    at RunMaze.Solve(RunMaze.java:11)
    at RunMaze.main(RunMaze.java:41)
```

Defensive Programming: Stay in Control.

In general, each place in code at which an assumption is made is a candidate for defensive self-checking. Those places include the following:

- 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).

The biggest drawback of aggressive validity checking is degraded performance, but during program development your time is valuable. Once you have found all the bugs, you can disable **assert** statements using the appropriate compiler option, at which point they cost you nothing.