Chapter 6

A Case Study: The Eight Queens Puzzle

This chapter presents the first of several case studies (or paradigms, in the original sense of the word) of programs written in an object-oriented style. The programs in this Chapter will be rather small so that we can present versions in several different langauges. Later case studies will be presented in only one language.

After first describing the problem, we will discuss how an object-oriented solution would differ from another type of solution. The chapter then concludes with a solution written in each language.

6.1 The Eight-Queens Puzzle

In the game of chess, the queen can attack any piece that lies on the same row, on the same column, or along a diagonal. The *eight-queens* is a classic logic puzzle. The task is to place eight queens on a chessboard in such a fashion that no queen can attack any other queen. A solution is shown in Figure 6.1, but this solution is not unique. The eight-queens puzzle is often used to illustrate problem-solving or backtracking techniques.

How would an object-oriented solution to the eight-queens puzzle differ from a solution written in a conventional imperative programming language? In a conventional solution, some sort of data structure would be used to maintain the positions of the pieces. A program would then solve the puzzle by systematically manipulating the values in these data structures, testing each new position to see whether it satisfied the property that no queen can attack any other.

We can provide an amusing but nevertheless illustrative metaphor for the difference between a conventional and an object-oriented solution. A conventional program is like a human being sitting above the board, and moving the chess pieces, which have no animate life of their own. In an object-oriented solution,

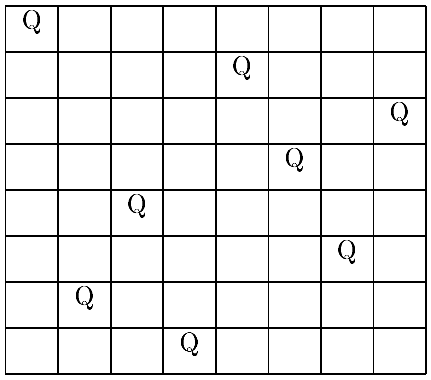


Figure 6.1: – One solution to the eight-queens puzzle.

on the other hand, we will empower the *pieces* to solve the problem themselves. That is, instead of a single monolithic entity controlling the outcome, we will distribute responsibility for finding the solution among many interacting agents. It is as if the chess pieces themselves are animate beings who interact with each other and take charge of finding their own solution.

Thus, the essence of our object-oriented solution will be to create objects that represent each of the queens, and to provide them with the abilities to discover the solution. With the computing-as-simulation view of Chapter 1, we are creating a model universe, defining behavior for the objects in this universe, then setting the universe in motion. When the activity of the universe stabilizes, the solution has been found.

6.1.1 Creating Objects That Find Their Own Solution

How might we define the behavior of a queen object so that a group of queens working together can find a solution on their own? The first observation is that, in any solution, no two queens can occupy the same column, and consequently no column can be empty. At the start we can therefore assign a specific column to each queen and reduce the problem to the simpler task of finding an appropriate row.

To find a solution it is clear that the queens will need to communicate with each other. Realizing this, we can make a second important observation that will greatly simplify our programming task–namely, each queen needs to know only about the queens to her immediate left. Thus, the data values maintained for each queen will consist of three values: a column value, which is *immutable*; a row value, which is altered in pursuit of a solution; and the neighboring queen to the immediate left.

Let us define an *acceptable solution for column n* to be a configuration of columns 1 through n in which no queen can attack any other queen in those columns. Each queen will be charged with finding acceptable solutions between herself and her neighbors on her left. We will find a solution to the entire puzzle by asking the right most queen to find an acceptable solution. A CRC-card description of the class Queen, including the data managed by each instance (recall that this information is described on the back side of the card), is shown in Figure 6.2.

6.2 Using Generators

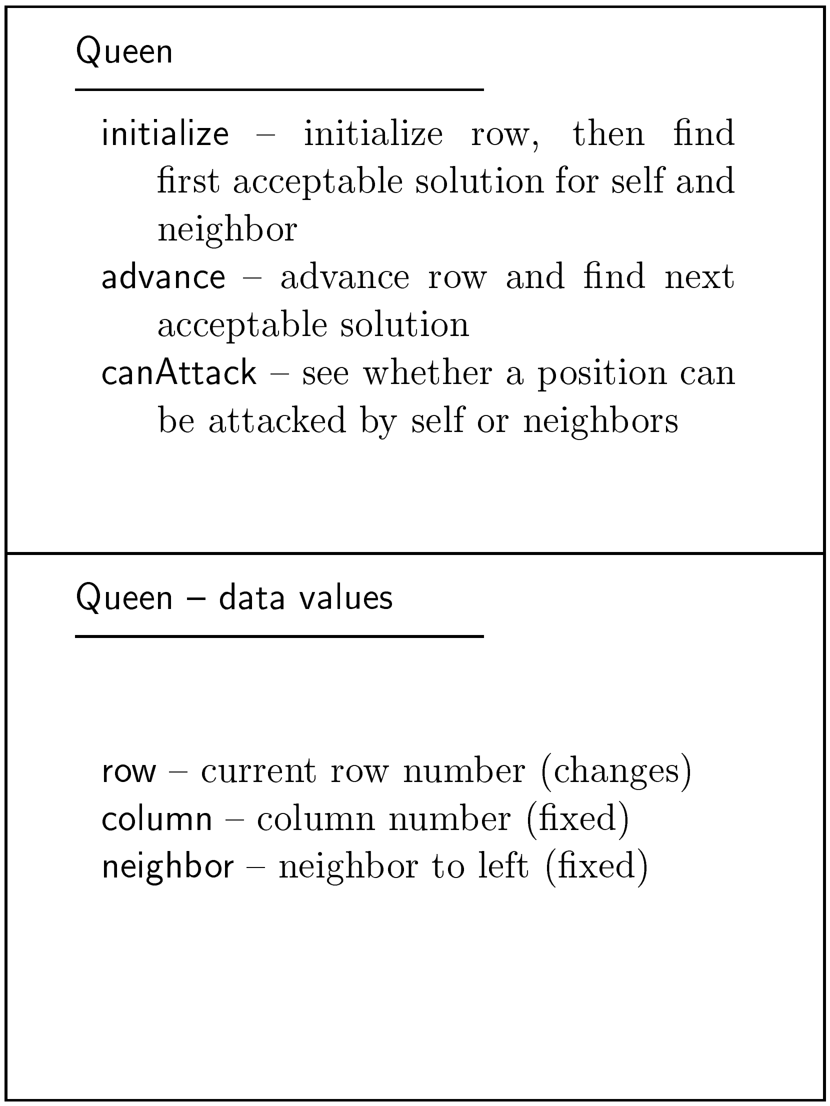
As with many similar problems, the solution to the eight-queens puzzle involves two interacting steps: *generating* possible partial solutions and *filtering out* solutions that fail to satisfy some later goal. This style of problem solving is sometimes known as the *generate and test* paradigm.

Let us consider the filter step first, as it is easier. For the system to test a potential solution it is sufficient for a queen to take a coordinate (row-column) pair and produce a Boolean value that indicates whether that queen, or any queen to her left, can attack the given location. A pseudo code algorithm that checks to see whether a queen can attack a specific position is given below. The procedure canAttack uses the fact that, for a diagonal motion, the differences in rows must be equal to the differences in columns.

```
function queen.canAttack(testRow , testColumn) -> boolean
    /* test for same row */
    if row = testRow then
        return true

    /* test diagonals */
    columnDifference := testColumn - column
    if (row + columnDifference = testRow) or
        (row - columnDifference = testRow)
            then return true

    /* we can't attack, see if neighbor can */
    return neighbor.canAttack(testRow, testColumn)
end
```

6.2.1 Initialization

We will divide the task of finding a solution into parts. The method initialize establishes the initial conditions necessary for a queen object, which in this case simply means setting the data values. This is usually followed immediately by a call on findSolution to discover a solution for the given column. Because such a



Queen

initialize – initialize row, then find first acceptable solution for self and neighbor
advance – advance row and find next acceptable solution
canAttack – see whether a position can be attacked by self or neighbors

Queen – data values

row – current row number (changes)
column – column number (fixed)
neighbor – neighbor to left (fixed)

Figure 6.2: – Front and back sides of the Queen CRC card.

solution will often not be satisfactory to subsequent queens, the message advance is used to advance to the next solution.

A queen in column n is initialized by being given a column number, and the neighboring queen (the queen in column $n - 1$). At this level of analysis, we will leave unspecified the actions of the leftmost queen, who has no neighbor. We will explore various alternative actions in the example problems we subsequently present. We will assume the neighbor queens (if any) have already been initialized, which includes their having found a mutually satisfactory solution. The queen in the current column simply places herself in row 1. A pseudo-code description of the algorithm is shown below.

```
function queen.initialize(col, neigh) -> boolean

        /* initialize our column and neighbor values */
    column := col
    neighbor := neigh

        /* start in row 1 */
    row := 1
    return findSolution;
end
```

6.2.2 Finding a Solution

To find a solution, a queen simply asks its neighbors if they can attack. If so, then the queen advances herself, if possible (returning failure if she cannot). When the neighbors indicate they cannot attack, a solution has been found.

```
function queen.findSolution -> boolean

        /* test positions */
    while neighbor.canAttack (row, column) do
        if not self.advance then
            return false

        /* found a solution */
    return true
end
```

As we noted in Chapter 5, the pseudo-variable self denotes the receiver for the current message. In this case we want the queen who is being asked to find a solution to pass the message advance to herself.

6.2.3 Advancing to the Next Position

The procedure advance divides into two cases. If we are not at the end, the queen simply advances the row value by 1. Otherwise, she has tried all positions and not found a solution, so nothing remains but to ask her neighbor for a new solution and start again from row 1.

```
function queen.advance -> boolean

        /* try next row */
    if row < 8 then begin
        row := row + 1
        return self.findSolution
    end

        /* cannot go further */
        /* move neighbor to next solution */
    if not neighbor.advance then
        return false

        /* start again in row 1 */
    row := 1
    return self.findSolution
end
```

The one remaining task is to print out the solution. This is most easily accomplished by a simple method, print that is rippled down the neighbors.

```
procedure print
    neighbor.print
    write row, column
end
```

6.3 The Eight-Queens Puzzle in Several Languages

In this section we present solutions to the eight-queens puzzle in several of the programming languages we are considering. Examine each variation, and compare how the basic features provided by the language make subtle changes to the final solution. In particular, examine the solutions written in Smalltalk and Objective-C, which use a special class for a sentinel value, and contrast this with the solutions given in Object Pascal, C++, or Java, all of which use a null pointer for the leftmost queen and thus must constantly test the value of pointer variables.

6.3.1 The Eight-Queens Puzzle in Object Pascal

The class definition for the eight-queens puzzle in Apple Object Pascal is shown below. A subtle but important point is that this definition is recursive; objects of type Queen maintain a data field that is itself of type Queen. This is sufficient to indicate that declaration and storage allocation are not necessarily linked; if they were, an infinite amount of storage would be required to hold any Queen value. We will contrast this with the situation in C++ when we discuss that language.

```
type
    Queen = object
            (* data fields *)
        row :        integer;
        column :    integer;
        neighbor :    Queen;

            (* initialization *)
        procedure initialize (col : integer; ngh : Queen);

            (* operations *)
        function     canAttack
                    (testRow, testColumn : integer) : boolean;
        function     findSolution : boolean;
        function     advance : boolean;
        procedure    print;
    end;
```

The class definition for the Delphi language differs only slightly, as shown below. The Borland language allows the class declaration to be broken into public and private sections, and it includes a constructor function, which we will use in place of the initialize routine.

```
TQueen = class (TObject)
public
    constructor Create (initialColumn : integer; nbr : TQueen);
    function findSolution : boolean;
    function advance : boolean;
    procedure print;

private
    function canAttack (testRow, testColumn : integer) : boolean;
    row : integer;
    column : integer;
    neighbor : TQueen;
```

```
end;
```

The pseudo-code presented in the earlier sections is reasonably close to the Pascal solution, with two major differences. The first is the lack of a return statement in Pascal, and the second is the necessity to first test whether a queen has a neighbor before passing a message to that neighbor. The functions findSolution and advance, shown below, illustrate these differences. (Note that Delphi Pascal differs from standard Pascal in permitting short-circuit interpretation of the and and or directives, in the fashion of C++. Thus, the code for the Delphi language could in a single expression combine the test for neighbor being non-null and the passing of a message to the neighbor).

```
function Queen.findSolution : boolean;
var
    done : boolean;
begin
    done := false;
    findsolution := true;

        (* test positions *)
    if neighbor <> nil then
        while not done and neighbor.canAttack(row, column) do
            if not self.advance then begin
                findSolution := false;
                done := true;
            end;
end;

function Queen.advance : boolean;
begin
    advance := false;
        (* try next row *)
    if row < 8 then begin
        row := row + 1;
        advance := self.findSolution;
    end
    else begin
            (* cannot go further *)
            (* move neighbor to next solution *)
        if neighbor <> nil then
            if not neighbor.advance then
                advance := false
            else begin
                row := 1;
                advance := self.findSolution;
        end;
    end;
end;
```

The main program allocates space for each of the eight queens and initializes the queens with their column number and neighbor value. Since during initialization the first solution will be discovered, it is only necessary for the queens to print their solution. The code to do this in Apple Object Pascal is shown below. Here, neighbor and i are temporary variables used during initialization and lastQueen is the most recently created queen.

```
begin
    neighbor := nil;
    for i := 1 to 8 do begin
            (* create and initialize new queen *)
        new (lastQueen);
        lastQueen.initial (i, neighbor);
        if not lastQueen.findSolution then
            writeln('no solution');
            (* newest queen is next queen neighbor *)
        neighbor := lastQueen;
    end;

        (* print the solution *)
    lastQueen.print;

    end;
end.
```

By providing explicit constructors that combine new object creation and initialization, the Delphi language allows us to eliminate one of the temporary variables. The main program for the Delphi language is as follows:

```
begin
    lastQueen := nil;
    for i := 1 to 8 do begin
            // create and initialize new queen
        lastQueen := Queen.create(i, lastQueen);
        lastQueen.findSolution;
        end;

        // print the solution
    lastQueen.print;
end;
```

6.3.2 The Eight-Queens Puzzle in C++

The most important difference between the pseudo-code description of the algorithm presented earlier and the eight-queens puzzle as actually coded in C++ is the explicit use of pointer values. The class description for the class Queen is shown below. Each instance maintains, as part of its data area, a pointer to another queen value. Note that, unlike the Object Pascal solution, in C++ this value must be declared explicitly as a pointer rather than an object value.

```
class Queen {
public:
        // constructor
    Queen (int, Queen *);

        // find and print solutions
    bool findSolution();
    bool advance();
    void print();

private:
        // data fields
    int row;
    const int column;
    const Queen * neighbor;

        // internal method
    bool canAttack (int, int);
};
```

As in the Delphi Pascal solution, we have subsumed the behavior of the method initialize in the constructor. We will describe this shortly.

There are three data fields. The integer data field column has been marked as const. This identifies the field as an immutable value, which cannot change during execution. The third data field is a pointer value, which either contains a null value (that is, points at nothing) or points to another queen.

Since initialization is performed by the constructor, the main program can simply create the eight queen objects, and then print their solution. The variable lastQueen will point to the most recent queen created. This value is initially a null pointer–it points to nothing. A loop then creates the eight values, initializing each with a column value and the previous queen value. When the loop completes, the leftmost queen holds a null value for its neighbor field while every other queen points to its neighbor, and the value lastQueen points to the rightmost queen.

```
void main() {
    Queen * lastQueen = 0;

    for (int i = 1; i <= 8; i++) {
        lastQueen = new Queen(i, lastQueen);
        if (! lastQueen->findSolution())
            cout << "no solution\n";
        }

    lastQueen->print();
}
```

We will describe only those methods that illustrate important points. The complete solution can be examined in Appendix A.

The constructor method must use the initialization clauses on the heading to initialize the constant value column, as it is not permitted to use an assignment operator to initialize instance fields that have been declared const. An initialization clause is also used to assign the value neighbor, although we have not declared this field as constant.

```
Queen::Queen(int col, Queen * ngh) : column(col), neighbor(ngh)
{
    row = 1;
}
```

Because the value of the neighbor variable can be either a queen or a null value, a test must be performed before any messages are sent to the neighbor. This is illustrated in the method findSolution. The use of short-circuit evaluation in the logical connectives and the ability to return from within a procedure simplify the code in comparison to the Object Pascal version, which is otherwise very similar.

```
bool Queen::findSolution()
{
    while (neighbor && neighbor->canAttack(row, column))
            if (! advance())
                return false;
    return true;
}
```

The advance method must similarly test to make certain there is a neighbor before trying to advance the neighbor to a new solution. When passing a message to oneself, as in the recursive message findSolution, it is not necessary to specify a receiver.

```
bool Queen::advance()
```

```
{
    if (row < 8) {
        row++;
        return findSolution();
        }

    if (neighbor && ! neighbor->advance())
        return false;

    row = 1;
    return findSolution();
}
```

6.3.3 The Eight-Queens Puzzle in Java

The solution in Java is in many respects similar to the C++ solution. However, in Java the bodies of the methods are written directly in place, and public or private designations are placed on the class definitions themselves. The following is the class description for the class Queen, with some of the methods omitted.

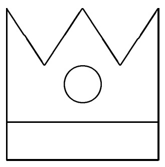
```
class Queen {
        // data fields
    private int row;
    private int column;
    private Queen neighbor;

        // constructor
    Queen (int c, Queen n) {
            // initialize data fields
        row = 1;
        column = c;
        neighbor = n;
        }

    public boolean findSolution() {
        while (neighbor != null &&
                neighbor.canAttack(row, column))
            if (! advance())
                return false;
        return true;
        }

    public boolean advance() { ...  }
    private boolean canAttack(int testRow, int testColumn) { ... }

    public void paint (Graphics g) { ... }
}
```

Unlike in C++, in Java the link to the next queen is simply declared as an object of type Queen and not as a pointer to a queen. Before a message is sent to the neighbor instance variable, an explicit test is performed to see if the value is null.

Since Java provides a rich set of graphics primitives, this solution will differ from the others in actually drawing the final solution as a board. The method paint will draw an image of the queen, then print the neighbor images.

```
class Queen {
    ...
    public void paint (Graphics g) {
            // x, y is upper left corner
            // 10 and 40 give slight margins to sides
        int x = (row - 1) * 50 + 10;
        int y = (column - 1) * 50 + 40;
        g.drawLine(x+5, y+45, x+45, y+45);
        g.drawLine(x+5, y+45, x+5, y+5);
        g.drawLine(x+45, y+45, x+45, y+5);
        g.drawLine(x+5, y+35, x+45, y+35);
        g.drawLine(x+5, y+5, x+15, y+20);
        g.drawLine(x+15, y+20, x+25, y+5);
        g.drawLine(x+25, y+5, x+35, y+20);
        g.drawLine(x+35, y+20, x+45, y+5);
        g.drawOval(x+20, y+20, 10, 10);
            // then draw neighbor
        if (neighbor != null)
            neighbor.paint(g);
        }
}
```

The graphics routines draw a small crown, which looks like this:



Java does not have global variables nor functions that are not member functions. As we will describe in more detail in Chapter 22, a program is created by the defining of a subclass of the system class JFrame, and then the overriding of certain methods. Notably, the constructor is used to provide initialization for the application, while the method paint is used to redraw the screen. Mouse events and window events are handled by creating *listener objects* that will execute when their associated event occurs. We will describe listeners in much greater detail in later sections. We name the application class QueenSolver and define it as follows:

```
public class QueenSolver extends JFrame {

    public static void main(String [ ] args) {
        QueenSolver world = new QueenSolver();
        world.show();
    }

    private Queen lastQueen = null;

    public QueenSolver() {
        setTitle("8 queens");
        setSize(600, 500);
        for (int i = 1; i <= 8; i++) {
            lastQueen = new Queen(i, lastQueen);
            lastQueen.findSolution();
            }
        addMouseListener(new MouseKeeper());
        addWindowListener(new CloseQuit());
        }

    public void paint(Graphics g) {
        super.paint(g);
            // draw board
        for (int i = 0; i <= 8; i++) {
            g.drawLine(50 * i + 10, 40, 50*i + 10, 440);
            g.drawLine(10, 50 * i + 40, 410, 50*i + 40);
        }
        g.drawString("Click Mouse for Next Solution", 20, 470);
            // draw queens
        lastQueen.paint(g);
        }

    private class CloseQuit extends WindowAdapter {
        public void windowClosing (WindowEvent e) {
            System.exit(0);
        }
    }

    private class MouseKeeper extends MouseAdapter {
        public void mousePressed (MouseEvent e) {
            lastQueen.advance();
            repaint();
        }
    }
}
```

Note that the application class must be declared as public, because it must be accessible to the main program.

6.3.4 The Eight-Queens Puzzle in Objective-C

The interface description for our class Queen is as follows:

```
@interface Queen : Object
{    /* data fields */
    int row;
    int column;
    id neighbor;
}

    /* methods */
- (void) initialize: (int) c neighbor: ngh;
- (int)  advance;
- (void) print;
- (int)  canAttack: (int) testRow column: (int) testColumn;
- (int)  findSolution;

@end
```

Each queen will maintain three data fields: a row value, a column, and the neighbor queen. The last is declared with the data type id. This declaration indicates that the value being held by the variable is an object type, although not necessarily a queen.

In fact, we can use this typeless nature of variables in Objective-C to our advantage. We will employ a technique that is not possible, or at least not as easy, in a more strongly typed language such as C++ or Object Pascal. Recall that the leftmost queen does not have any neighbor. In the C++ solution, this was indicated by the null, or empty value, in the neighbor pointer variable in the leftmost queen. In the current solution, we will instead create a new type of class, a *sentinel value*. The leftmost queen will point to this sentinel value, thereby ensuring that every queen has a valid neighbor.

Sentinel values are frequently used as endmarkers and are found in algorithms that manipulate linked lists, such as our linked list of queen values. The difference between an object-oriented sentinel and a more conventional value is that an object-oriented sentinel value can be *active*–it can have *behavior*–which means it can respond to requests.

What behaviors should our sentinel value exhibit? Recall that the neighbor links in our algorithm were used for two purposes. The first was to ensure that a given position could not be attacked; our sentinel value should always respond negatively to such requests, since it cannot attack any position. The second use of the neighbor links was in a recursive call to print the solution. In this case our sentinel value should simply return, since it does not have any information concerning the solution.

Putting these together yields the following implementation for our sentinel queen.

```
@implementation SentinelQueen : Object
- (int) advance
{
    /* do nothing */
    return 0;
}

- (int) findSolution
{
    /* do nothing */
    return 1;
}

- (void) print
{
    /* do nothing */
}

- (int) canAttack: (int) testRow column: (int) testColumn;
{
    /* cannot attack */
    return 0;
}
@end
```

In the full solution there is an implementation section for SentinelQueen, but no interface section. This omission is legal, although the compiler will provide a warning since it is somewhat unusual.

The use of the sentinel allows the methods in class Queen to simply pass messages to their neighbor without first determining whether or not she is the leftmost queen. The method for canAttack, for example, illustrates this use:

```
- (int) canAttack: (int) testRow column: (int) testColumn
{    int columnDifference;

    /* can attack same row */
    if (row == testRow)
        return 1;

    columnDifference = testColumn - column;
    if ((row + columnDifference == testRow) ||
        (row - columnDifference == testRow))
            return 1;

    return [ neighbor canAttack:testRow column: testColumn ];
}
```

Within a method, a message sent to the receiver is denoted by a message sent to the pseudo-variable self.

```
- (void) initialize: (int) c neighbor: ngh
{
    /* set the constant fields */
    column = c;
    neighbor = ngh;
    row = 1;
}

- (int) findSolution
{
    /* loop until we find a solution */
    while ([neighbor canAttack: row and: column ])
        if (! [self advance])
            return 0; /* return false */
    return 1; /* return true */
}
```

Other methods are similar, and are not described here.

6.3.5 The Eight-Queens Puzzle in Smalltalk

The solution to the eight-queens puzzle in Smalltalk is in most respects very similar to the solution given in Objective-C. Like Objective-C, Smalltalk handles the fact that the leftmost queen does not have a neighbor by defining a special *sentinel* class. The sole purpose of this class is to provide a target for the messages sent by the leftmost queen.

The sentinel value is the sole instance of the class SentinelQueen, a subclass of class Object, which implements the following three methods:

```
advance

        " sentinels do not attack "
    ↑ false


canAttack: row column: column
        " sentinels cannot attack "
    ↑ false

result
        " return empty list as result "
    ↑ List new
```

One difference between the Objective-C and Smalltalk versions is that the Smalltalk code returns the result as a list of values rather than printing it on the output. The techniques for printing output are rather tricky in Smalltalk and vary from implementation to implementation. By returning a list we can isolate these differences in the calling method.

The class Queen is a subclass of class Object. Instances of class Queen maintain three instance variables: a row value, a column value, and a neighbor. Initialization is performed by the method setColumn:neighbor:

```
setColumn: aNumber neighbor: aQueen
        " initialize the data fields "
    column := aNumber.
    neighbor := aQueen.
    row := 1.
```

The canAttack method differs from the Objective-C counterpart only in syntax:

```
canAttack: testRow column: testColumn | columnDifference |
    columnDifference := testColumn - column.
    (((row = testRow) or:
        [ row + columnDifference = testRow]) or:
        [ row - columnDifference = testRow])
            ifTrue: [ ↑ true ].
    ↑ neighbor canAttack: testRow column: testColumn
```

Rather than testing for the negation of a condition, Smalltalk provides an explicit ifFalse statement, which is used in the method advance:

```
advance
        " first try next row "
    (row < 8)
        ifTrue: [ row := row + 1. ↑ self findSolution ].
        " cannot go further, move neighbor "
    (neighbor advance) ifFalse: [ ↑ false ].
        " begin again in row 1 "
    row := 1.
    ↑ self findSolution
```

The while loop in Smalltalk must use a block as the condition test, as in the following:

```
findSolution
    [ neighbor canAttack: row column: column ]
        whileTrue: [ self advance ifFalse: [ ↑ false ] ].
    ↑ true
```

A recursive method is used to obtain the list of answer positions. Recall that an empty list is created by the sentinel value in response to the message result.

```
result
    ↑ neighbor result; addLast: row
```

A solution can be found by invocation of the following method, which is not part of class Queen but is instead attached to some other class, such as Object.

```
solvePuzzle | lastQueen |
    lastQueen := SentinelQueen new.
    1 to: 8 do: [:i | lastQueen := (Queen new)
        setColumn: i neighbor: lastQueen.
        lastQueen findSolution ].
    ↑ lastQueen result
```

6.3.6 The Eight-Queens Puzzle in Ruby

Ruby is a recent scripting language, similar in spirit to Python or Perl. There are only functions in Ruby, every method returns a value, which is simply the value of the last statement in the body of the method. A feel for the syntax for

Ruby can be found by the definition of the sentinel queen, which can be written as follows:

```
class NullQueen

  def canAttack(row, column)
    false
  end

  def first?
    true
  end

  def next?
    false
  end

  def getState
    Array.new
  end

end
```

The class Queen handles all but the last case. In Ruby instance variables must begin with an at-sign. Thus the initialization method is written as follows:

```
class Queen

  def initialColumn(column, neighbour)
    @column = column
    @neighbour = neighbour
    nil
  end
  ...

end
```

Conditional statements are written in a curious form where the expression is given first, followed by the if keyword. This is illustrated by the method canAttack:

```
  def canAttack(row, column)
    return true if row == @row

    cd = (column - @column).abs
    rd = (row - @row).abs
    return true if cd == rd

    @neighbour.canAttack(row, column)
  end
```

The remainder of the Ruby solution can be found in the appendix.

Chapter Summary

In this first case study we have examined a classic puzzle, how to place eight queens on a chessboard in such a way that no queen can attack any of the others. While the problem is moderately intriguing, our interest is not so much in the problem itself, but in the way the solution to the problem has been structured. We have addressed the problem by making the queens into independent agents, who then work among themselves to discover a solution.

Further Reading

A solution to the eight-queens puzzle constructed without the use of a sentinel value was described in my earlier book on Smalltalk [Budd 1987].

The eight queens puzzle is found in many computing texts. See [Griswold 1983, Budd 1987, Berztiss 1990], for some representative examples.

For further information on the general technique termed generate and test, see [Hanson 1981], or [Berztiss 1990].

The solution in Ruby was written by Mike Stok. Further information on Ruby can be found in [Thomas 2001].

Self Study Questions

1. What is the eight queens puzzle?

2. In what way is the object-oriented solution presented here different from a conventional solution?

3. What is the generate and test approach to finding a solution in a space of various alternative possibilities?

4. What is a sentinel? (The term is introduced in the solution presented in Objective-C).

Exercises

1. Modify any one of the programs to produce all possible solutions rather than just one. How many possible solutions are there for the eight-queens puzzle? How many of these are rotations of other solutions? How might you filter out rotations?

2. Can you explain why the sentinel class in the Objective-C and Smalltalk versions of the eight-queens puzzle do not need to provide an implementation for the method findSolution, despite the fact that this message is passed to the neighbor value in the method advance?

3. Suppose we generalize the eight-queens problem to the N-queens problem, where the task is to place N queens on an N by N chessboard. How must the programs be changed?

   It is clear that there are values for N for which no solution exists (consider N=2 or N=3, for example). What happens when your program is executed for these values? How might you produce more meaningful output?

4. Using whatever graphics facilities your system has, alter one of the programs to display dynamically on a chessboard the positions of each queen as the program advances. What portions of the program need to know about the display?