Chapter 8: Stepwise Refinement

compedu.stanford.edu/karel-reader/docs/python/en/chapter8.html

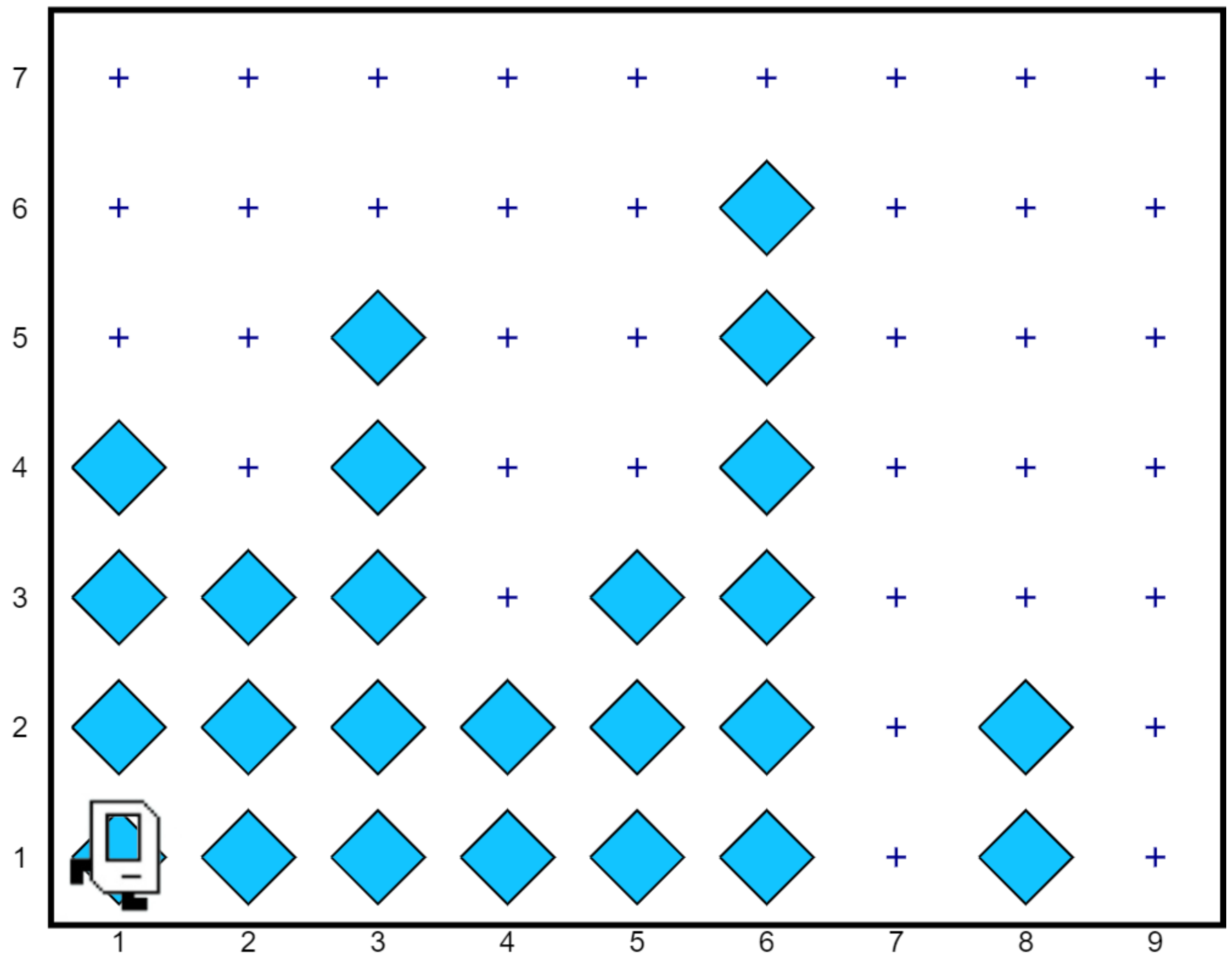
To a large extent, programming is the science of solving problems by computer. Because problems are often difficult, solutions—and the programs that implement those solutions—can be difficult as well. In order to make it easier for you to develop those solutions, you need to adopt a methodology and discipline that reduces the level of that complexity to a manageable scale.

In the early years of programming, the concept of computing as a science was more or less an experiment in wishful thinking. No one knew much about programming in those days, and few thought of it as an engineering discipline in the conventional sense. As programming matured, however, such a discipline began to emerge. The cornerstone of that discipline is the understanding that programming is done in a social environment in which programmers must work together. If you go into industry, you will almost certainly be one of many programmers working to develop a large program. That program, moreover, is almost certain to live on and require maintenance beyond its originally intended application. Someone will want the program to include some new feature or work in some different way. When that occurs, a new team of programmers must go in and make the necessary changes in the programs. If programs are written in an individual style with little or no commonality, getting everyone to work together productively is extremely difficult.

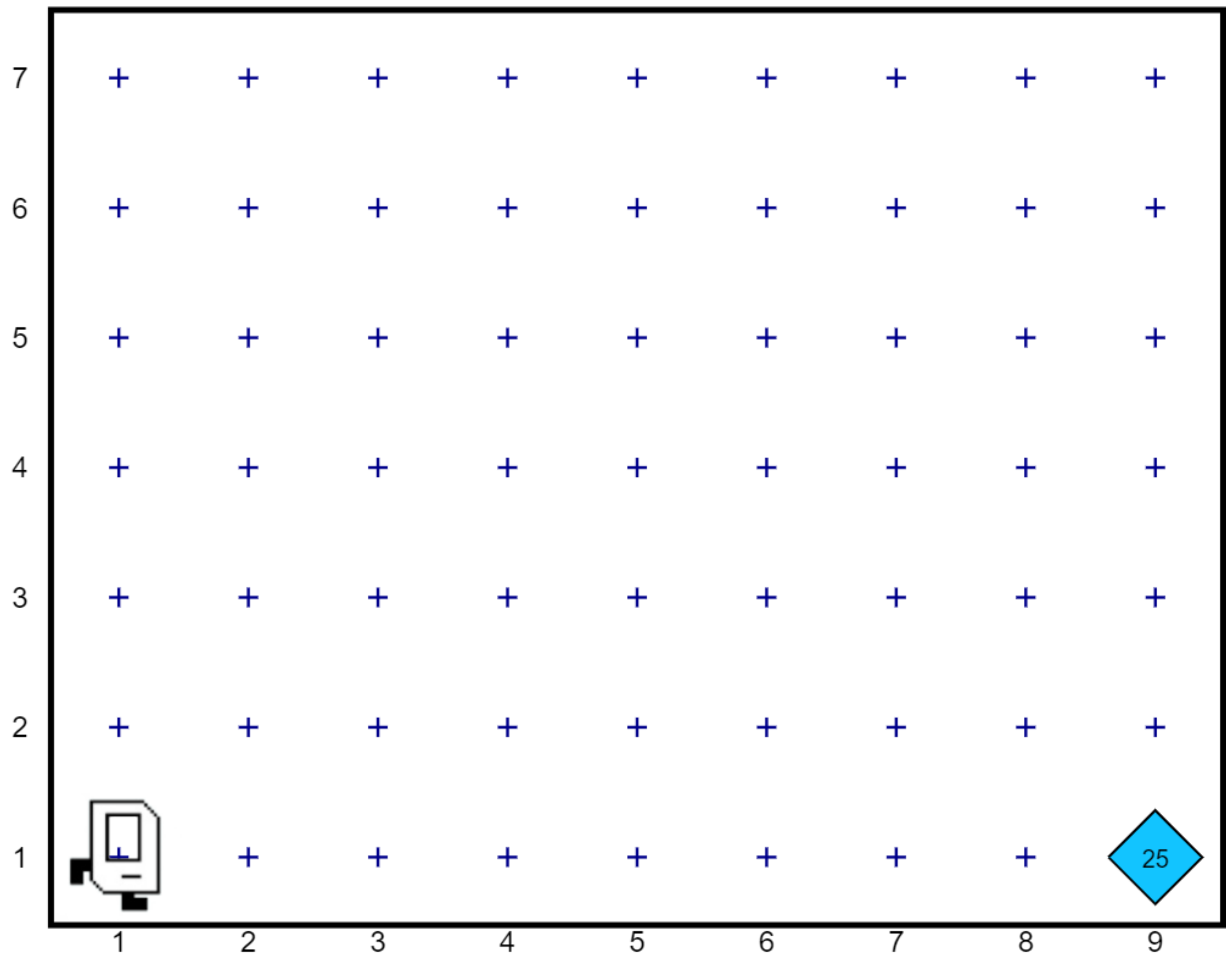
To combat this problem, programmers began to develop a set of programming methodologies that are collectively called **software engineering**. Using good software engineering skills not only makes it easier for other programmers to read and understand your programs, but also makes it easier for you to write those programs in the first place. One of the most important methodological advances to come out of software engineering is the strategy of **top-down design** or **stepwise refinement**, which consists of solving problems by starting with the problem as a whole. You break the whole problem down into pieces, and then solve each piece, breaking those down further if necessary. This top down strategy is complemented with **iterative testing** where you make sure that the smaller pieces of the solution are working before moving on.

An exercise in stepwise refinement

To illustrate the concept of stepwise refinement, let's teach Karel to solve a new problem. Imagine that Karel is now living in a world that looks something like this:



On each of the columns, there is a tower of beepers of an unknown height, although some columns (such as the 7th, and 9th in the sample world) may be empty. Karel's job is to collect all the beepers in each of these towers, put them back down on the easternmost corner of 1st row, and then return to its starting position. Thus, when Karel finishes its work in the example above, all 25 beepers currently in the towers should be stacked on the corner of 9th column and 1st row, as follows:



Importantly, you may assume that Karel initially *starts* with zero beepers in its bag. Each beeper picked up is added to its bag. When putting beepers in the corner, Karel can use the `beepers_in_bag()` test. We can also assume that the columns do not reach all the way up to the northmost wall.

The key to solving this problem is to decompose the program in the right way, while still being able to test as you go. This task is more complex than the others you have seen, which makes choosing appropriate subproblems more important to obtaining a successful solution.

The principle of top-down design

The key idea in stepwise refinement is that you should start the design of your program from the top, which refers to the level of the program that is conceptually highest and most abstract. At this level, the beeper tower problem is clearly divided into three independent phases. First, Karel has to collect all the beepers. Second, Karel has to deposit them on the last intersection. Third, Karel has to return to its home position. This conceptual decomposition of the problem suggests that the `main()` function for this program will have the following structure:

```
def main():
    collect_all_beepers()
    drop_all_beepers()
    return_home()
```

At this level, the problem is easy to understand. Of course, there are a few details left over in the form of functions that you have not yet written. Even so, it is important to look at each level of the decomposition and convince yourself that, as long as you believe that the functions you are about to write will solve the subproblems correctly, you will then have a solution to the problem as a whole.

Iterative testing as you go

Now that you have defined the structure for the program as a whole, it is time to move on to the first subproblem, which consists of collecting all the beepers. This task is itself more complicated than the simple problems from the preceding chapters. Collecting all the beepers means that you have to pick up the beepers in every tower until you get to the final corner. The fact that you need to repeat an operation for each tower suggests that you need a `while` loop here. The `while` loop will repeat the process of `collect_one_tower()` and then moving.

Caution: It is dangerous to try to write the entire program without **testing** it as you go. If you make a mistake it will be hard to find the mistake. We know that we are going to repeat the process of collecting one tower. Let us write and **test** collecting a single tower before we put the `collect_one_tower()` process in a for loop. Thus *temporarily* we can start with the following definition of `collect_all_beepers()` :

```
def collect_all_beepers() :
    # temporary implementation for testing purposes
    collect_one_tower()
    move()
```

As a guiding principle, if you have a complex loop, test the *body* of the loop before you write the entire loop.

Refining collect tower

When `collect_one_tower()` is called, Karel is either standing at the base of a tower of beepers or standing on an empty corner. In the former case, you need to collect the beepers in the tower. In the latter, you can simply move on. This situation sounds like an application for the `if` statement, in which you would write something like this:

```
if beepers_present():
    collect_actual_tower()
```

Before you add such a statement to the code, you should think about whether you need to make this test. Often, programs can be made much simpler by observing that cases that at first seem to be special can be treated in precisely the same way as the more general situation. In the current problem, what happens if you decide that there is a tower of beepers on every avenue but that some of those towers are zero beepers high? Making use of this insight simplifies the program because you no longer have to test whether there is a tower on a particular avenue.

The `collect_one_tower()` function is still complex enough that an additional level of decomposition is in order. To collect all the beepers in a tower, Karel needs to undertake the following steps:

1. Turn left to face the beepers in the tower.
2. Collect all the beepers in the tower, stopping when no more beepers are found.
3. Turn around to face back toward the bottom of the world.
4. Return to the wall that represents the ground.
5. Turn left to be ready to move to the next corner.

Once again, this outline provides a model for the `collect_one_tower()` function, which looks like this:

```
    def collect_one_tower():
        turn_left()
        collect_line_of_beepers()
        turn_around()
        move_to_wall()
        turn_left()
```

Function preconditions and postconditions

The `turn_left()` commands at the beginning and end of the `collect_one_tower()` function are both critical to the correctness of this program. When `collect_one_tower()` is called, Karel is always somewhere on 1st row facing east. When it completes its operation, the program as a whole will work correctly only if Karel is again facing east at that same corner. Conditions that must be true before a function is called are referred to as **preconditions**; conditions that must apply after the function finishes are known as **postconditions**.

When you define a function, you will get into far less trouble if you write down exactly what the pre- and postconditions are. Once you have done so, you then need to make sure that the code you write always leaves the postconditions satisfied, assuming that the preconditions were satisfied to begin with. For example, think about what happens if you call `collect_one_tower()` when Karel is on 1st row facing east. The first `turn_left()` command leaves Karel facing north, which means that Karel is properly aligned with the column of beepers representing the tower. The `collect_line_of_beepers()` function-which has yet to be written but nonetheless performs a task that you understand conceptually— simply moves without turning. Thus, at the end of the call to `collect_line_of_beepers()`, Karel will still be facing north. The `turn_around()` call therefore leaves Karel facing south. Like `collect_line_of_beepers()`, the `move_to_wall()` function does not involve any turns but instead simply moves until it hits the boundary wall. Because Karel is facing south, this boundary wall will be the one at the bottom of the screen, just below 1st row. The final `turn_left()` command therefore leaves Karel on 1st row facing east, which satisfies the postcondition.

Repeating the process

You run your program and it successfully clears one tower and leaves Karel in the promised postcondition. Wahoo! You have just hit a milestone in solving this hard task! We now have to repeat the process of clearing one tower using a `while` loop.

But what does this `while` loop look like? First of all, you should think about the conditional test. You want Karel to stop when it hits the wall at the end of the row. Thus, you want Karel to keep going as long as the space in front is clear. Thus, you know that the `collect_all_beepers()` function will include a `while` loop that uses the `front_is_clear()` test. At each position, you want Karel to collect all the beepers in the tower beginning on that corner. If you give that operation a name, which might be something like `collect_one_tower()`, you can go ahead and write a definition for the `collect_all_beepers()` function even though you haven't yet filled in the details.

You do, however, have to be careful. The code for `collect_all_beepers()` does not look like this:

```
def collect_all_beepers():
    # buggy loop!
    while front_is_clear():
        collect_one_tower()
        move()
```

This implementation is buggy for exactly the same reason that the first version of the general `BeeperLine` program from chapter 6 failed to do its job. There is a fencepost error in this version of the code, because Karel needs to test for the presence of a beeper tower on the last avenue. The correct implementation is:

```
def collect_all_beepers():
    while front_is_clear():
        collect_one_tower()
        move()
    collect_one_tower()
```

Note that this function has precisely the same structure as the main program from the PlaceBeeperLine program presented in chapter 6. The only difference is that this program calls `collect_one_tower()` where the other called `put_beeper()`. These two programs are each examples of a general strategy that looks like this:

```
def collect_all_beepers():
    while front_is_clear():
```
perform some operation.
```
        move()
```
perform the same operation for the final corner.

You can use this strategy whenever you need to perform an operation on every corner as you move along a path that ends at a wall. If you remember the general structure of this strategy, you can use it whenever you encounter a problem that requires such an operation. Reusable strategies of this sort come up frequently in programming and are referred to as **programming idioms** or **patterns**. The more patterns you know, the easier it will be for you to find one that fits a particular type of problem.

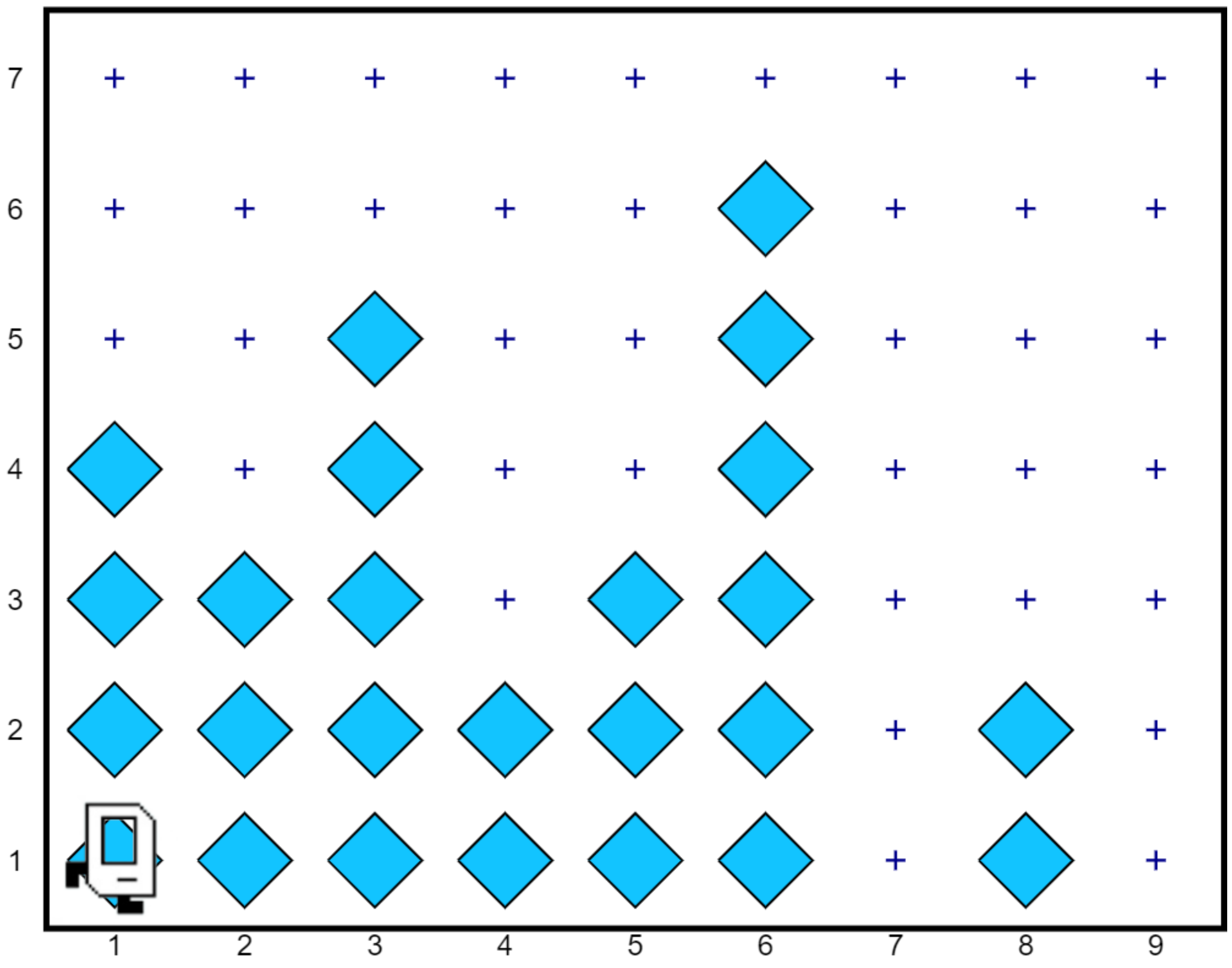
Finishing up

Although the hard work has been done, there are still several loose ends that need to be resolved. The main program calls two functions— `drop_all_beepers()` and `return_home()` — that are as yet unwritten. Similarly, `collect_one_tower()` calls `collect_line_of_beepers()` and `move_to_wall()` . Fortunately, all four of these functionss are simple enough to code without any further decomposition, particularly if you use `move_to_wall()` in the definition of `return_home()` . Here is the complete implementation:

```
# File: BeeperCollectingKarel.py

# --------------------------------

# The BeeperCollectingKarel class collects all the beepers

# in a series of vertical towers and deposits them at the

# eastmost corner on 1st row.

from karel.stanfordkarel import *

def main():

collect_all_beepers()

drop_all_beepers()

return_home()

# Collects the beepers from every tower by moving along 1st

# row, calling collect_one_tower at every corner. The

# postcondition for this function is that Karel is in the

# easternmost corner of 1st row facing east.

def collect_all_beepers():

while front_is_clear():

collect_one_tower()

move()

collect_one_tower()
```

```
# Collects the beepers in a single tower. When collect_one_tower

# is called, Karel must be on 1st row facing east. The

# postcondition for collect_one_tower is that Karel must again

# be facing east on that same corner.

def collect_one_tower():

turn_left()

collect_line_of_beepers()

turn_around()

move_to_wall()

turn_left()

# Collects a consecutive line of beepers. The end of the beeper

# line is indicated by a corner that contains no beepers.

def collect_line_of_beepers():

while beepers_present():

pick_beeper()

if front_is_clear():

move()

# Drops all the beepers on the current corner.

def drop_all_beepers() :

while beepers_in_bag():

put_beeper()

# Returns Karel to its initial position at the corner of 1st

# Avenue and 1st row, facing east. The precondition for this

# function is that Karel must be facing east somewhere on 1st

# row, which is true at the conclusion of collect_all_beepers.

def return_home():

turn_around()
```

```
move_to_wall()

turn_around()

# Moves Karel forward until it is blocked by a wall.

def move_to_wall():

while front_is_clear():

move()

# Turns Karel 180 degrees around

def turn_around():

turn_left()

turn_left()
```



Next Chapter