3.1 Solving Problems

Writing programs involves solving problems. One model[1] describes problem solving as a process that has four activities: defining the problem, planning the solution, implementing the plan, and analyzing the solution.

When programming, the solution is called an **algorithm**. An algorithm is a finite set of step-by-step instructions that specifies a process of moving from the initial situation to the final situation. That is, an algorithm is the "solution" spelled out in a step-by-step manner.

We find many algorithms in our lives. A recipe for lasagna is an algorithm, as are the directions for assembling a child's wagon. Even bottles of shampoo have algorithms printed on them:

> *wet hair with warm water*
> *gently work in the first application of shampoo*
> *rinse thoroughly and repeat*

While people may have no trouble interpreting this algorithm, it is not precise enough for computers. How much warm water? How much shampoo? What does it mean to "gently work in?" How many times should it be repeated? Once? A hundred times? Indefinitely? Is it necessary to wet the hair (again) for the repeated applications?

Not all algorithms are equally effective. Good algorithms share five qualities. Good algorithms are:

- correct
- easy to read and understand
- easy to debug
- easy to modify to solve variations of the original task
- efficient[2]

This chapter is about designing algorithms, particularly algorithms that can be encoded as computer programs and executed by a computer. This concept is not new—from the very beginning of this text, we have been writing algorithms and turning them into programs. Now we will focus more deliberately on the process.

[1] G. Polya, *How to Solve It,* Princeton University Press, 1945, 1973.

[2] Meaning that the algorithm does not require performing more steps than necessary. Efficiency should never compromise the first guideline, and only rarely should it compromise the other three.

3.2 Stepwise Refinement

Stepwise refinement is a method of constructing algorithms. An algorithm to solve a complex problem may be written by decomposing the problem into smaller, simpler subproblems, each with its own algorithm. Each sub-problem solves a logical step in the larger problem. The problem as a whole is solved by solving all of the subproblems.

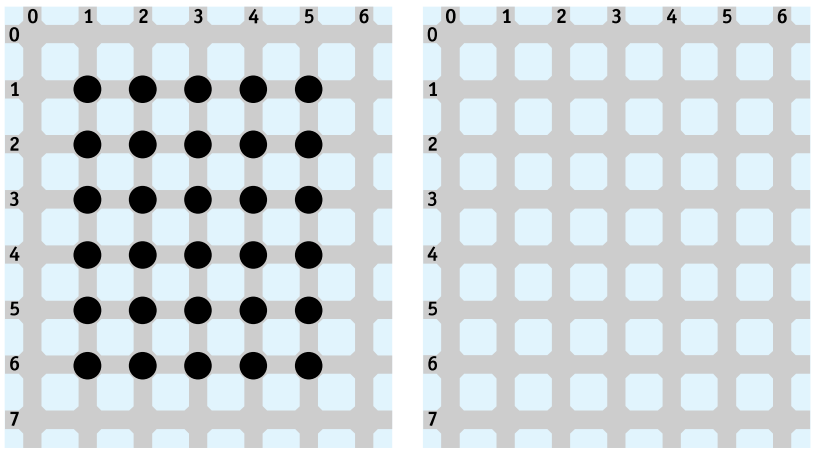
When algorithms are expressed as computer programs, algorithms are encoded in methods. Stepwise refinement encourages us to write each method in terms of other methods that implement one logical step in solving the problem. In this way, we can write programs that are more likely to be correct, simple to read, and easy to understand.

It may appear natural to define all the new classes and services needed for a task first, and then write the program using these services. But how can we know what robots and which new services are needed before we write the program? Stepwise refinement tells us to first write the program using any robots and service names we desire, and then define these robots and their services. That is, we write the `main` method first, and then we write the definitions of the new services we used. Finally, we assemble the class containing `main` and any new classes we wrote into a complete program.

We will explore this process more concretely by writing a program for the task shown in Figure 3-1. The initial situation represents a harvesting task that requires one or more robots to pick up a rectangular field of `Things`. The robot(s) may start and finish wherever is most convenient.

(figure 3-1)

Harvesting task



The first step is to develop an overall plan to guide us in writing a robot program to perform the given task. Planning is often best done as a group activity. Sharing ideas

in a group allows members to present different plans that can be thoughtfully examined for strengths and weaknesses. Even if we are working alone, we can think in a question-and-answer pattern, such as the following:

Question How many robots do we need to perform this task?

Answer We could do it with one robot that walks back and forth over all of the rows to be harvested, or we could do it with a team of robots, where each robot picks some of the rows.

Question How many shall we use?

Answer Let's try it with just one robot, named `mark`, for now. That seems simpler than using several robots.

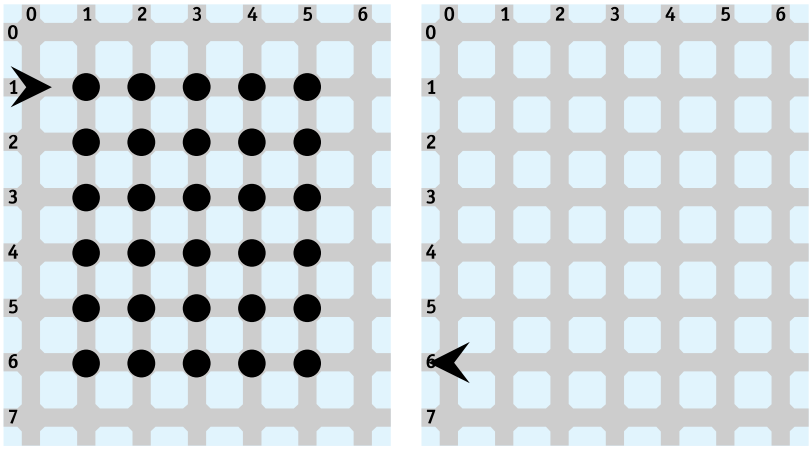
LOOKING AHEAD

In Section 3.5, we'll find that these are not well-founded assumptions.

Question Where should `mark` start?

Answer Probably at one of the corners. Then it doesn't need to go back to harvest rows behind it. Let's pick intersection (1, 0), facing the first row it will pick.

With these decisions made about how many robots to use and where to start, we can be more definite about the initial situation. The revised version appears in Figure 3-2.



Revised initial situation — Revised final situation

(figure 3-2)

Revised situations

3.2.1 Identifying the Required Services

Now that the initial situation is complete, we turn our attention to identifying the services `mark` must offer.

Question What do we want `mark` to do?

Answer Harvest all the things in the field.

Question So it sounds like we need a new service, perhaps called `harvestField`. Does `mark` need to have any other services?

Answer Well, the initial situation doesn't actually put `mark` in the field. We could either adjust the initial situation so it starts at (1, 1) or simply call `move` before it harvests the field. Other than that, `harvestField` seems to be the only service required.

Once the services required have been identified, we can make use of them in writing the `main` method. At this point, we won't worry about the fact that they don't exist yet.

We briefly move from planning to implementing our plan. We will call the new class of robots `Harvester` and implement the `main` method in a class named `HarvestTask`.

Defining a city with 30 `Things` would clutter Listing 3-1 significantly. To avoid this, the `City` class has a constructor, used in line 10, that can read a file to determine where `Things` are positioned. The requirements for such a file are described in the online documentation for the `City` class.

FIND THE CODE

ch03/harvest/

Listing 3-1: *The* `main` *method for harvesting a field of things*

```
1  import becker.robots.*;
2
3  /** A program to harvest a field of things 5 columns wide and 6 rows high.
4  *
5  *  @author Byron Weber Becker */
6  public class HarvestTask
7  {
8    public static void main (String[] args)
9    {
10     City stLouis = new City("Field.txt");
11     Harvester mark = new Harvester(
12                      stLouis, 1, 0, Direction.EAST);
13
14     mark.move ();
15     mark.harvestField();
16     mark.move ();
17   }
18 }
```

3.2.2 Refining `harvestField`

We now know that the `Harvester` class must offer a service named `harvestField` that harvests a field of things. As we develop this service, we will follow the same pattern as before—asking ourselves questions about what it must do and what services we want to use to implement the `harvestField` service.

Using other services to implement `harvestField` builds on the observation we made in Section 2.2.4 when we implemented `turnRight`: methods may use other methods within the same class. Recall the declaration of `turnRight`:

```
public void turnRight()
{ this.turnAround();
  this.turnLeft();
}
```

When we implemented `turnRight`, we noticed that `turnAround`, a method we had already written, would be useful. However, to implement `harvestField`, we are turning that process around. We need to write a method, `harvestField`, and begin by asking which methods we need to help make writing `harvestField` easier. These methods are called **helper methods**. We will write `harvestField` as if those methods already existed. Helper methods are used frequently enough to qualify as a pattern.

KEY IDEA

Write a long or complex method using helper methods.

Eventually, of course, we will have to write each of the helper methods. It may be that we will have to follow the same technique for them as well: defining the helper methods in terms of other services that we wish we had. Each time we do so, the helper methods should be simpler than the method we are writing. Eventually, they will be simple enough to be written without helper methods.

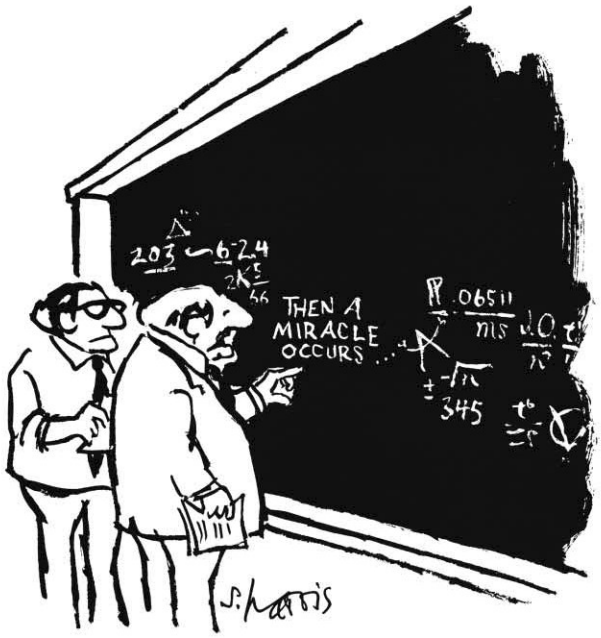
PATTERN

Helper Method

We must be realistic when imagining which helper methods would be useful to implement `harvestField`. Step 2 in Figure 3-3—"then a miracle occurs"—would *not* be an appropriate helper method.

(figure 3-3)

A rather vague step



First Refinement Attempt

If you are working in a group to develop a program, a conversation between a Java expert and a novice to define the helper methods might proceed as follows. Even if you are working alone, it is still helpful to hold a "conversation" like this with yourself.

Expert So, what does a `Harvester` robot need to do to pick a field of things?

Novice Harvest all the things in each of the rows of the field.

Expert How could a `Harvester` robot harvest just one row?

Novice It could move west to east across the northern-most unharvested row of things, picking each thing as it moves.

Expert How could it harvest the entire field?

KEY IDEA

Use a helper method when doing the same thing several times.

Novice At the end of each row, the robot could turn around and move back to the western side of the field, move south one block, face east, and repeat the actions listed earlier. It could do so for each row of things in the field. Since the field has six rows, the robot needs to repeat the actions six times.

Expert If you were to write this down in an abbreviated form, what would it look like?

Novice *pick all the things in one row*
return to the start of the row
move south one block

pick all the things in one row
return to the start of the row
move south one block

pick all the things in one row
return to the start of the row
move south one block

Performing the actions in these nine lines would harvest the first three rows of the field. They need to be repeated to harvest the last three rows.

Analysis of the First Refinement Attempt

Before we continue with this plan, we should analyze it, looking at its strengths and weaknesses. Are we asking for the right helper methods? Are there other ways of solving the problem that might work better? Our analysis might proceed as follows:

Expert What are the strengths of this plan?

Novice The plan simplifies the `harvestField` method by defining three simpler methods, using each one several times.

Expert What are the weaknesses of the plan?

Novice The same three lines are repeated over and over. Maybe we should have `harvestField` defined as

harvest one row
harvest one row
harvest one row

and so on. The method to harvest one row could be defined using the helper methods mentioned earlier.

Expert That's easy enough to do. Any other weaknesses in this plan?

Novice The `Harvester` robot makes some "empty trips."

Expert What do you mean by "empty trips?"

Novice The robot returns to the starting point on the row that was just harvested.

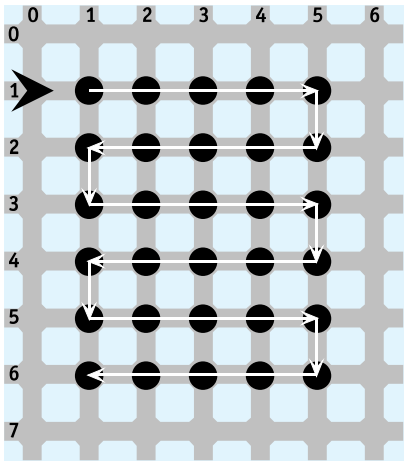
Expert Why is this bad?

Novice It seems like a better solution to have the robot doing productive work (as opposed to just moving) in both directions. I know that if I were picking that field personally, I'd look for every efficiency I could find!

Instead of harvesting only one row and then turning around and returning to the start, the `Harvester` robot could pick all the things in one row, move south one row, and come back to the west, harvesting a second row. It could then move one row south to begin the entire process over for the next two rows. If `mark` repeats these steps one more time, the entire field of things will be harvested, as shown in Figure 3-4.

(figure 3-4)

Harvesting the field in two directions



Expert How would you write that in an abbreviated form?

Novice Well, `harvestField` would be defined as follows:

harvest two rows
position for next harvest
harvest two rows
position for next harvest
harvest two rows

Again we analyze this new plan for its strengths and weaknesses.

Expert What advantage does this offer over the first plan?

Novice Now the robot makes only six trips across the field instead of 12. There are no empty trips.

Expert What are the weaknesses of this new plan?

Novice The robot harvests two rows at a time. If the field had an odd number of rows, we would have to think of something else.

When we are planning solutions, we should be very critical and not just accept the first plan as the best. We now have two different plans, and you can probably think of several more. Let's avoid the empty trips and implement the second plan.

Implementing `harvestField`

Recall the brief form of the idea:

harvest two rows
position for next harvest
harvest two rows
position for next harvest
harvest two rows

LOOKING AHEAD

This brief form is called pseudocode. We'll learn more about it in Section 3.4.

Let's turn each of these statements into invocations of methods named `harvestTwoRows` and `positionForNextHarvest`. We can then begin implementation of the `Harvester` class and `harvestField` in particular, as shown in Listing 3-2.

Helper Method

The listing includes the complete implementation of `harvestField` as well as stubs for `harvestTwoRows` and `positionForNextHarvest`. A method that has just enough code to compile, but not to actually do its job is called a **stub**. Stubs are useful for at least three reasons:

- Stubs serve as placeholders for work that must still be completed. The associated documentation records our ideas for what the methods should do, helping to jog our memory when we come back to actually implement the methods. In large programs with many methods, a span of days or even months might elapse before you have a chance to complete the method. If you are part of a team, perhaps someone else can implement the method based on the stub and its documentation.
- A stub allows the program to be compiled even though it is not finished. When we compile the program, the compiler may catch errors that are easier to find and fix now rather than later. Waiting to compile until the entire program is written may result in so many interrelated errors that debugging becomes very difficult.
- A compiled program can be run, which may allow some early testing to be performed that validates our ideas (or uncovers bugs that are easier to fix now rather than later). We might run the program to verify that the initial situation is correctly set up, for instance.

FIND THE CODE

ch03/harvest/

Listing 3-2: *An incomplete implementation of the* `Harvester` *class*

```
1  import becker.robots.*;
2
3  /** A class of robot that can harvest a field of things. The field must be 5 things wide
4  *  and 6 rows high.
5  *
6  *  @author Byron Weber Becker */
7  public class Harvester extends RobotSE
8  {
9     /** Construct a new Harvester robot.
10    *  @param aCity The city where the robot will be created.
11    *  @param str   The robot's initial street.
12    *  @param ave   The robot's initial avenue.
13    *  @param dir   The initial direction, one of Direction.{NORTH, SOUTH, EAST, WEST}. */
14    public Harvester(City aCity,
15                     int str, int ave, Direction dir)
16    { super(aCity, str, ave, dir);
17    }
18
19    /** Harvest a field of things. The robot is on the northwest corner of the field. */
20    public void harvestField()
21    { this.harvestTwoRows();
22      this.positionForNextHarvest();
23      this.harvestTwoRows();
24      this.positionForNextHarvest();
25      this.harvestTwoRows();
26    }
27
28    /** Harvest two rows of the field, returning to the same avenue but one street
29    *  farther south. The robot must be facing east. */
30    public void harvestTwoRows()
31    { // Incomplete.
32    }
33
34    /**  Go one row south and face east. The robot must be facing west. */
35    public void positionForNextHarvest()
36    { // Incomplete.
37    }
38 }
```

PATTERN

Helper Method

We must now begin to think about planning the instructions `harvestTwoRows` and `positionForNextHarvest`.

3.2.3 Refining `harvestTwoRows`

Our plan contains two subtasks: one harvests two rows and the other positions the robot to harvest two more rows. The planning of these two subtasks must be just as thorough as the planning was for the overall task. Let's begin with `harvestTwoRows`.

Expert What does `harvestTwoRows` do?

Novice `harvestTwoRows` must harvest two rows of things. One is harvested as the `Harvester` robot travels east and the second is harvested as it returns to the west.

Expert What does the robot have to do?

Novice It must pick things and move as it travels east. At the end of the row of things, it must move south one block, face west, and return to the western edge of the field, picking things as it travels west. In an abbreviated form, it must complete the following tasks:

harvest one row while moving east
go south to the next row
harvest one row while moving west

We analyze this plan as before, looking for strengths and weaknesses.

Expert What are the strengths of this plan?

Novice It seems to solve the problem.

Expert What are the weaknesses of this plan?

Novice Possibly one—we have two different instructions that harvest a single row of things.

Expert Do we really need two different harvesting instructions?

Novice We need one for going east and one for going west.

Expert Do we really need a separate method for each direction?

Novice Harvesting is just a series of `pickThings` and `moves`. The direction the robot is moving does not matter. If we plan `goToNextRow` carefully, we can use one instruction to harvest a row of things when going east and the same instruction for going west.

By reusing a method, we make the program smaller and potentially easier to understand. The new plan is as follows:

harvest one row
go to the next row
harvest one row

Translating this idea to Java, we arrive at the following method and stubs, which should be added to the code in Listing 3-2.

```
28  /** Harvest two rows of the field, returning to the same avenue but one street
29  *  farther south. The robot must be facing east. */
30  public void harvestTwoRows()
31  { this.harvestOneRow();
32    this.goToNextRow();
33    this.harvestOneRow();
34  }
35
36  /** Harvest one row of five things. */
37  public void harvestOneRow()
38  { // incomplete
39  }
40
41  /** Go one row south and face west. The robot must be facing east. */
42  public void goToNextRow()
43  { // incomplete
44  }
```

Helper Method

This doesn't look good! Every time we implement a method, we end up with even more methods to implement. We now have three outstanding methods, `positionForNextHarvest`, `harvestOneRow`, and `goToNextRow`, all needing to be finished. Rest assured, however, that these methods are getting more and more specific. Eventually, they will be implemented only in terms of already existing methods such as `move`, `turnLeft`, and `pickThing`. Then the number of methods left to implement will begin to decrease until we have completed the entire program.

KEY IDEA

Implement the methods in execution order.

We have a choice of which of the three methods to refine next. One good strategy is to choose the first uncompleted method we enter while tracing the program. This strategy allows us to run the program to verify that the work done thus far is correct. Applying this strategy indicates that we should work on `harvestOneRow` next.

3.2.4 Refining `harvestOneRow`

We now focus our efforts on `harvestOneRow`.

Expert What does `harvestOneRow` do?

Novice Starting on the first thing and facing the correct direction, the robot must harvest each of the intersections that it encounters, stopping on the location of the last thing in the row.

Expert What does the `Harvester` robot have to do?

Novice It must harvest the intersection it's on and then move to the next intersection. It needs to do that five times, once for each thing in the row.

harvest an intersection
move
harvest an intersection
move
harvest an intersection
move
harvest an intersection
move
harvest an intersection
move

Expert Are you sure? It seems to me that it moves right out of the field.

Novice Right! The last time it doesn't need to move to the next intersection. It can just go to the next row of the field.

We can implement `harvestOneRow` and `harvestIntersection` as follows.

```
/** Harvest one row of five things. */
public void harvestOneRow()
{ this.harvestIntersection();
  this.move();
  this.harvestIntersection();
  this.move();
  this.harvestIntersection();
  this.move();
  this.harvestIntersection();
  this.move();
  this.harvestIntersection();
}

/** Harvest one intersection. */
public void harvestIntersection()
{ this.pickThing();
}
```

PATTERN
Helper Method

It may seem silly to define a method such as `harvestIntersection` that contains only one method. There are two reasons why it is a good idea:

- The language of the problem has been about "harvesting," not "picking." This method carries that language throughout the program, making the program easier to understand.

- What it means to harvest an intersection may change. By isolating the concept of harvesting an intersection in this method, we provide a natural place to make future changes. For example, suppose a future field requires harvesting two things on each intersection. With the helper method, we need to add just one `pickThing` to the `harvestIntersection` method. Without the helper method, we would need to change the program at five places in the `harvestOneRow` method.

3.2.5 Refining `goToNextRow`

Let's now plan `goToNextRow`.

Expert What does `goToNextRow` do?

Novice It moves the `Harvester` robot south one block to the next row and faces it in the opposite direction. I think we can implement this one without creating any new helper methods, like this:

turn right
move
turn right

3.2.6 Refining `positionForNextHarvest`

At last, we have only one stub to complete, `positionForNextHarvest`.

Expert What does `positionForNextHarvest` do?

Novice It moves the `Harvester` robot south one block to the next row.

Expert Didn't we do that already? Why can't we use the instruction `goToNextRow`?

Novice The robot isn't in the correct situation. When executing `goToNextRow`, the robot is on the eastern edge of the field facing east. When it executes `positionForNextHarvest`, it has just finished harvesting two rows and is on the western edge of the field facing west.

Take a moment to simulate the `goToNextRow` instruction on paper. Start with a `Harvester` robot facing west and see where the robot is when you finish simulating the instruction.

Expert What does the robot have to do?

Novice It must turn left, not right, to face south, move one block, and turn left again to face east.

The implementation of this new method follows:

```
/** Position the robot for the next harvest by moving one street south and facing west. */
public void positionForNextHarvest()
{ this.turnLeft();
  this.move();
  this.turnLeft();
}
```

All of the method stubs have now been completed.

3.2.7 The Complete Program

Because we have spread this class out over several pages, the complete program is printed in Listing 3-3 so that you will find it easier to read and study.

Listing 3-3: *The complete* `Harvester` *class*

FIND THE CODE

ch03/harvest/

```
1  import becker.robots.*;
2
3  /** A class of robot that can harvest a field of things. The field must be 5 things wide
4  *  and 6 rows high.
5  *
6  *  @author Byron Weber Becker */
7  public class Harvester extends RobotSE
8  {
9     /** Construct a new Harvester robot.
10    *  @param aCity The city where the robot will be created.
11    *  @param str   The robot's initial street.
12    *  @param ave   The robot's initial avenue.
13    *  @param dir   The initial direction, one of Direction.{NORTH, SOUTH, EAST, WEST}. */
14    public Harvester(City aCity,
15                     int str, int ave, Direction dir)
16    { super(aCity, str, ave, dir);
17    }
18
19    /** Harvest a field of things. The robot is on the northwest corner of the field. */
20    public void harvestField()
21    { this.harvestTwoRows();
22      this.positionForNextHarvest();
23      this.harvestTwoRows();
24      this.positionForNextHarvest();
25      this.harvestTwoRows();
26    }
```

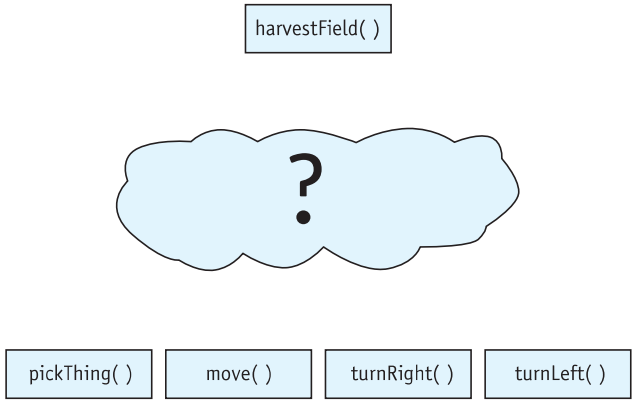
Listing 3-3: *The complete* `Harvester` *class* (continued)

```
27
28    /** Harvest two rows of the field, returning to the same avenue but one
29    *   street farther south. The robot must be facing east. */
30    public void harvestTwoRows()
31    { this.harvestOneRow();
32      this.goToNextRow();
33      this.harvestOneRow();
34    }
35
36    /** Harvest one row of five things. */
37    public void harvestOneRow()
38    { this.harvestIntersekction();
39      this.move();
40      this.harvestIntersection();
41      this.move();
42      this.harvestIntersection();
43      this.move();
44      this.harvestIntersection();
45      this.move();
46      this.harvestIntersection();
47    }
48
49    /** Go one row south and face west. The robot must be facing east. */
50    public void goToNextRow()
51    { this.turnRight();
52      this.move();
53      this.turnRight();
54    }
55
56    /** Go one row south and face east. The robot must be facing west. */
57    public void positionForNextHarvest()
58    { this.turnLeft();
59      this.move();
60      this.turnLeft();
61    }
62
63    /** Harvest one intersection. */
64    public void harvestIntersection()
65    { this.pickThing();
66    }
67  }
```

3.2.8 Summary of Stepwise Refinement

Stepwise refinement can be viewed as an approach to bridging the gap between the method we need (`harvestField`) and the methods we already have (`move`, `pickThing`, and so on). The methods we already have available are sometimes called the **primitives**. For drawing a picture, the primitives include `drawRect` and `drawLine`.

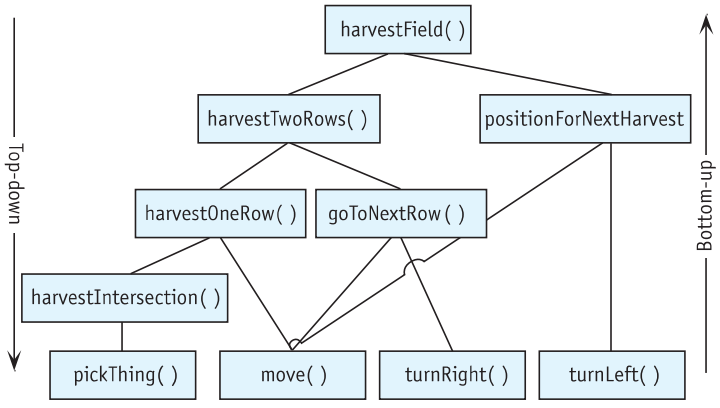
Figure 3-5 shows the situation near the beginning of the design process. We know we want a method to harvest a field and we know that robots can move, pick things up, turn left, and so on. The question is, how do we bridge the gap between them? Stepwise refinement helps fill in intermediate methods, as shown in Figure 3-6, in an orderly manner to help solve the problem.



(figure 3-5)

Gap between the method we need and the primitives we have available

Design is best performed starting at the top of the diagram and working down. This approach is often called **top-down design**. Stepwise refinement is simply another name for top-down design.



(figure 3-6)

Bridging the gap between the method we need and the primitives we have available

Sometimes we may have a flash of intuition and realize that harvesting one row would be a useful step in harvesting a field and that such a method could be easily constructed with the `move` and `pickThing` methods. When such an insight occurs before being derived in a top-down design, it's called **bottom-up design**. Bottom-up design happens within the context of top-down design.

It is also useful to make a distinction between top-down and **bottom-up implementation**. A top-down design may be done only on paper using pseudocode or even a diagram such as Figure 3-6. When actually writing the methods, we can start at the top and work down (as we did in this section of the book) or we can start at the bottom and work up. One advantage of the top-down approach is that it matches the design process. A significant advantage of the bottom-up approach is that methods can be implemented and tested before the entire program is complete. Testing methods as they are written almost always improves the correctness of the overall program.

3.3 Advantages of Stepwise Refinement

Developing programs using stepwise refinement has a number of advantages. The programs we create are more likely to be:

- Easy to understand
- Free of programming errors
- Easy to test and debug
- Easy to modify

All of these advantages follow from a few simple facts. First, as we noted in Section 1.1.1, most people can only manage about seven pieces of information at once. By breaking each problem into a small number of subproblems, the stepwise refinement technique helps us avoid information overload.

Furthermore, stepwise refinement imposes a structure on the problem. Related parts are kept together in methods; unrelated parts will be in different methods.

Finally, by identifying each of these related parts (methods) with well-chosen names, we can think at a higher level of abstraction; we can think about what the part does rather than how it does it.

We now investigate each of the four advantages of stepwise refinement.

3.3.1 Understandable Programs

Writing understandable programs is as important as writing correct ones; some say that it is even more important, since most programs initially have a few errors, and understandable programs are easier to debug. Successful programmers are distinguished from

ineffective ones by their ability to write clear and concise programs that someone else can read and quickly understand. What makes a program easy to understand? We present three criteria.

- Each method, including the `main` method, is composed of a few easily understood statements, including method calls.
- Each method has a single, well-defined purpose, which is succinctly described by the method's name.
- Each method can be understood by examining the statements it contains and understanding the purpose of the methods it calls. Understanding the method should not depend on knowing how other methods work. It should only depend upon the methods' purposes.

Each of these criteria help limit the number of details a person must keep in mind at one time.

If a method cannot correctly accomplish its purpose unless it begins in a certain situation, that fact should be documented. For example, an instruction directing a robot to always pick something up should indicate in a comment where that thing must appear:

```
public class Collector extends Robot
{
  /** Collects one thing from the next intersection. Breaks the robot if nothing is present. */
  public void collectOneThing()
  { this.move();
    this.pickThing();
  }
}
```

3.3.2 Avoiding Errors

Many novices think that all of the planning, analyzing, tracing, and simulating of programs shown in the `Harvester` example take too much time. They would rather start typing their programs into a computer immediately, without planning first.

What really takes time is correcting mistakes. These mistakes fall into two broad categories.

KEY IDEA

A T-shirt slogan: Days of programming can save you hours of planning.

The first category is planning mistakes. They result in execution and intent errors and happen when we write a program without an adequate plan. Planning mistakes can waste a lot of programming time. They are usually difficult to fix because large segments of the program may have to be modified or discarded. Careful planning and thorough analysis of the plan can help avoid planning mistakes.

The second category is programming mistakes. They result in compile-time errors and happen when we actually write the program. Programming mistakes can be spelling, punctuation, or other similar errors. Compiling the program each time we complete a

method helps find such errors so that they can be fixed. If we write the entire program before compiling it, we will undoubtedly have many errors to correct, some of which may be multiple instances of the same error. By using stubs and compiling often, we can both reduce the overall number of errors introduced at any one time and help prevent multiple occurrences of the same mistake.

Stepwise refinement is a tool that allows us to plan, analyze, and implement our plans in a way that should lead to a program containing a minimum of errors.

3.3.3 Testing and Debugging

Removing programming errors is easier in a program that has been developed using stepwise refinement. Removing errors has two components: identifying errors, and fixing the errors. Stepwise refinement helps in both steps.

LOOKING AHEAD

In Section 7.1.1, we will learn to write small programs designed to test single methods.

First, each method can be independently tested to identify errors that may be present. When writing a program, we should trace each method immediately after it is written until we are convinced that it is correct. Then we can forget how the method works and just remember what it does. Remembering should be easy if we name the method accurately, which is easiest if the method does only one thing.

Errors that are found by examining a method independently are the easiest ones to fix because the errors cannot have been caused by some other part of the program. When testing an entire program at once, this assumption cannot be made. If methods have not been tested independently, it is often the case that one has an error that does not become obvious until other methods have executed—that is, the signs of an error can first appear far from where the error actually occurs, making debugging difficult.

Second, stepwise refinement imposes a structure on our programs, and we can use this structure to help us find bugs in a completed program. When debugging a program, we should first determine which of the methods is malfunctioning. Then we can concentrate on debugging that method, while ignoring the other parts of the program, which are irrelevant to the bug. For example, suppose our robot makes a wrong turn and tries to pick up a thing from the wrong place. Where is the error? If we use helper methods to write our program, and each helper method performs one specific task (such as `positionForNextHarvest`) or controls a set of related tasks (such as `harvestTwoRows`), then we can usually determine the probable location of the error easily and quickly.

3.3.4 Future Modifications

Programs are often modified because the task to perform has changed in some way or there is an additional, related task to perform. Programs that have been developed using stepwise refinement are easier to modify than those that are not for the following reasons:

- The structure imposed on the program by stepwise refinement makes it easier to find the appropriate places to make modifications.
- Methods that have a single purpose and minimal, well-defined interactions with the rest of the program can be modified with less chance of creating a bug elsewhere in the program.
- Single-purpose methods can be overridden in subclasses to do something slightly different.

We can illustrate these points with an example modifying the `Harvester` class. Figure 3-7 shows two situations that differ somewhat from the original harvesting task. In the first one, each row has six things to harvest instead of just five. In the second, there are eight rows instead of six.

Obviously, this problem is very similar to the original harvesting problem. It would be much simpler to modify the `Harvester` program than to write a completely new program.

LOOKING AHEAD

Repetition and parameters (Chapters 4 and 5) will help our code adapt to variations of the same problem.

How difficult would it be to modify the `Harvester` class to accomplish the new harvesting tasks? We have two different situations to consider.

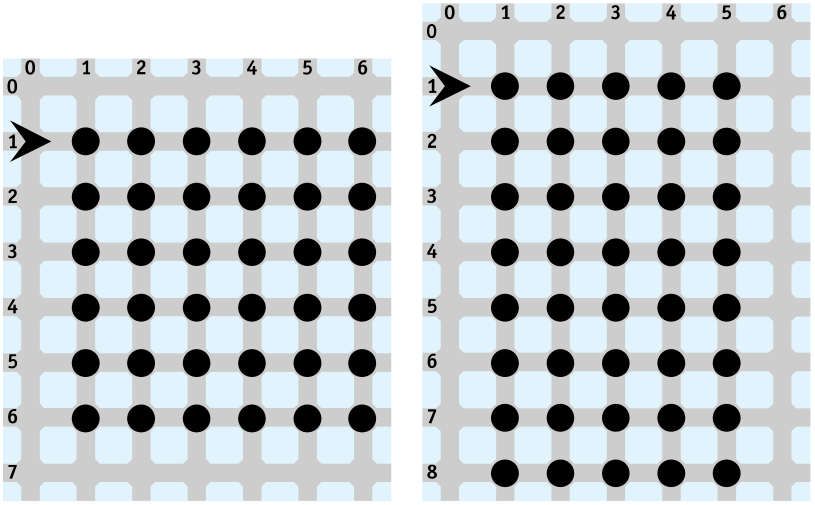
The first situation is one in which the original task has really changed, and it therefore makes sense to change the `Harvester` class itself. In this case, harvesting longer rows can be easily accommodated by adding the following statements to the `harvestOneRow` method:

```
this.move();
this.harvestIntersection();
```

A similar change to the `harvestField` method would solve the problem of harvesting additional rows.

(figure 3-7)

Two variations of the harvesting task



Our use of stepwise refinement in developing the original program aids this change tremendously. Stepwise refinement led us to logical subproblems. By naming them appropriately, it was easy to find where to change the program and how to change it. Furthermore, because the interactions between the methods were few and well defined, we could make the changes without creating a bug elsewhere in the program.

A second situation to consider is where we still need to solve the original problem—that is, it is inappropriate to change the original `Harvester` class, because it is still needed. We can then use inheritance to solve the new problem. By overriding `harvestOneRow`, we can make modifications to harvest longer rows, and by overriding `harvestField`, we can harvest more (or fewer) rows. A new robot class to harvest longer rows is shown in Listing 3-4.

FIND THE CODE

ch03/harvestLongRow/

Listing 3-4: *An extended version of* `Harvester` *that harvests longer rows*

```
1  import becker.robots.*;
2
3  /** A kind of Harvester robot that harvests fields with 6 things per row rather than just 5.
4  *
5  *  @author Byron Weber Becker */
6  public class LongRowHarvester extends Harvester
7  { /** Construct the harvester. */
8    public LongRowHarvester(City acity,
9                            int str, int ave, Direction dir)
10   { super(acity, str, ave, dir);
11   }
```

Listing 3-4: *An extended version of* `Harvester` *that harvests longer rows* (continued)

```
12
13    /** Override the harvestOneRow method to harvest the longer row. */
14    public void harvestOneRow()
15    { super.harvestOneRow();              // harvest first 5 intersections
16      this.move();                        // harvest one more
17      this.harvestIntersection();
18    }
19  }
```

3.4 Pseudocode

Sometimes it is useful to focus more on the algorithm than on the program implementing it. When we focus on the program, we also need to worry about many distracting details, such as placing semicolons appropriately, using consistent spelling, and even coming up with the names of methods. Those details can consume significant mental energy—energy that we would rather put into thinking about how to solve the problem.

KEY IDEA

Pseudocode is a blend of natural and programming languages.

Pseudocode is a technique that allows us to focus on the algorithms. Pseudocode is a blending of the naturalness of our native language with the structure of a programming language. It allows us to think about an algorithm much more carefully and accurately than we would with only **natural language**, the language we use in everyday speech, but without all the details of a full programming language such as Java. Think of it as your own personal programming language.

We've been using pseudocode for a long time without saying much about it. When planning our first program in Chapter 1, we presented the pseudocode for the algorithm before we wrote the program:

move forward until it reaches the thing,
pick up the thing
move one block farther
turn right
move a block
put the thing down
move one more block

Looking back for text set in this distinctive font, you'll also see that we used pseudocode in Chapter 2 when we developed the `Lamp` class and overrode `turnLeft` to make a faster-turning robot. We've also used it extensively in this chapter.