Chapter 3 Developing Methods

Chapter Objectives

After studying this chapter, you should be able to:

- Use stepwise refinement to implement long or complex methods
- Explain the advantages to using stepwise refinement
- Use pseudocode to help design and reason about methods before code is written
- Use multiple objects to solve a problem
- Use inheritance to reduce duplication of code and increase flexibility
- Explain why some methods should not be available to all clients and how to appropriately hide them

In Chapter 2, we wrote new services such as `turnRight` and `turnAround`. These services were very simple, consisting of only a few steps to accomplish the task.

In this chapter, we will examine techniques for implementing much more complex services that require many steps to accomplish the task.

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.

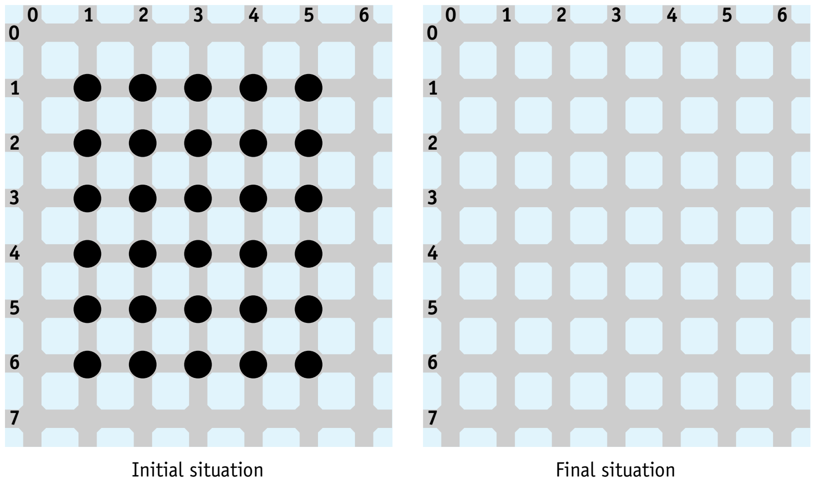
When algorithms are expressed as computer programs, algorithms are encoded in methods. Stepwise refinement encourages us to write each method in terms of other methods that implement one logical step in solving the problem. In this way, we can write programs that are more likely to be correct, simple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

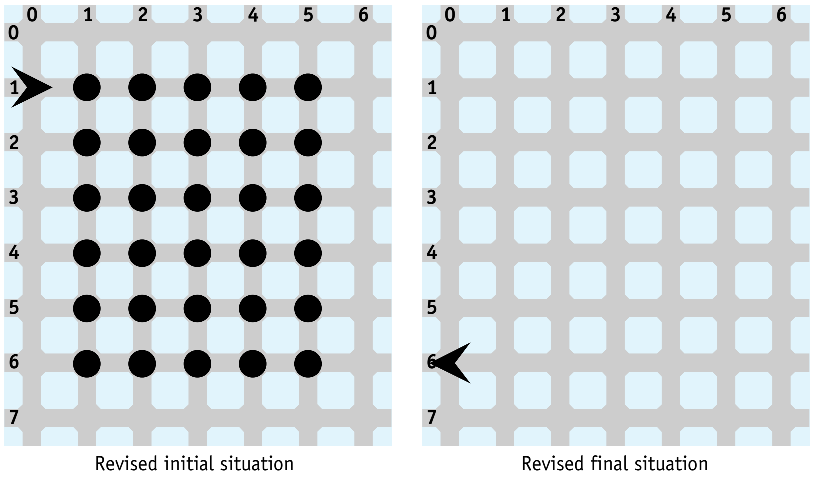
Question Where should `mark` start?

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.

LOOKING AHEAD

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

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.



(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

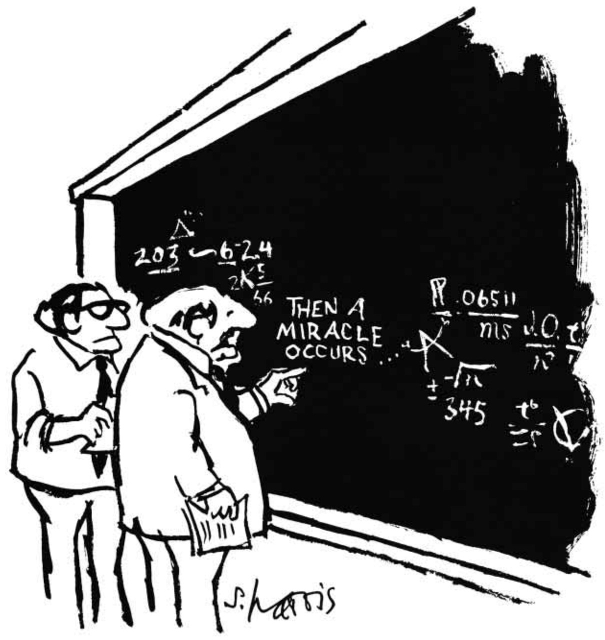
Write a long or complex method using helper methods.

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.

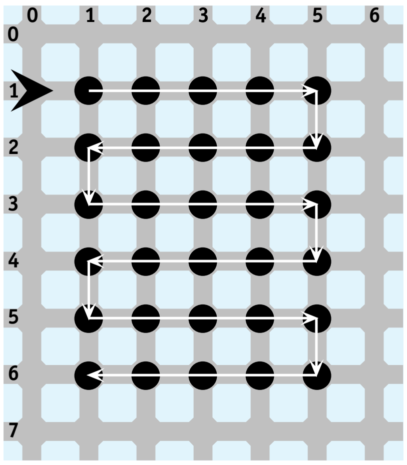
Expert Why is this bad?

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.

PATTERN

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*

```
import becker.robots.*;

/** A class of robot that can harvest a field of things. The field must be 5 things wide
 *   and 6 rows high.
 *
 *   @author Byron Weber Becker */
public class Harvester extends RobotSE
{
   /** Construct a new Harvester robot.
    *   @param aCity The city where the robot will be created.
    *   @param str      The robot's initial street.
    *   @param ave      The robot's initial avenue.
    *   @param dir      The initial direction, one of Direction.{NORTH, SOUTH, EAST, WEST}. */
   public Harvester(City aCity,
                    int str, int ave, Direction dir)
   { super(aCity, str, ave, dir);
   }

   /** Harvest a field of things. The robot is on the northwest corner of the field. */
   public void harvestField()
   { this.harvestTwoRows();
     this.positionForNextHarvest();
     this.harvestTwoRows();
     this.positionForNextHarvest();
     this.harvestTwoRows();
   }

   /** Harvest two rows of the field, returning to the same avenue but one street
    *  farther south. The robot must be facing east. */
   public void harvestTwoRows()
   {  // Incomplete.
   }

   /**  Go one row south and face east. The robot must be facing west. */
   public void positionForNextHarvest()
   {  // Incomplete.
   }
}
```

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.

```
/** Harvest two rows of the field, returning to the same avenue but one street
 *   farther south. The robot must be facing east. */
public void harvestTwoRows()
{ this.harvestOneRow();
  this.goToNextRow();
  this.harvestOneRow();
}

/** Harvest one row of five things. */
public void harvestOneRow()
{ // incomplete
}

/** Go one row south and face west. The robot must be facing east. */
public void goToNextRow()
{ // incomplete
}
```

PATTERN

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*

```
import becker.robots.*;

/** A class of robot that can harvest a field of things. The field must be 5 things wide
 *  and 6 rows high.
 *
 *  @author Byron Weber Becker */
public class Harvester extends RobotSE
{
  /** Construct a new Harvester robot.
   *  @param aCity The city where the robot will be created.
   *  @param str     The robot's initial street.
   *  @param ave     The robot's initial avenue.
   *  @param dir     The initial direction, one of Direction.{NORTH, SOUTH, EAST, WEST}. */
  public Harvester(City aCity,
                   int str, int ave, Direction dir)
  { super(aCity, str, ave, dir);
  }

  /** Harvest a field of things. The robot is on the northwest corner of the field. */
  public void harvestField()
  { this.harvestTwoRows();
    this.positionForNextHarvest();
    this.harvestTwoRows();
    this.positionForNextHarvest();
    this.harvestTwoRows();
  }
```

FIND THE CODE

ch03/harvest/

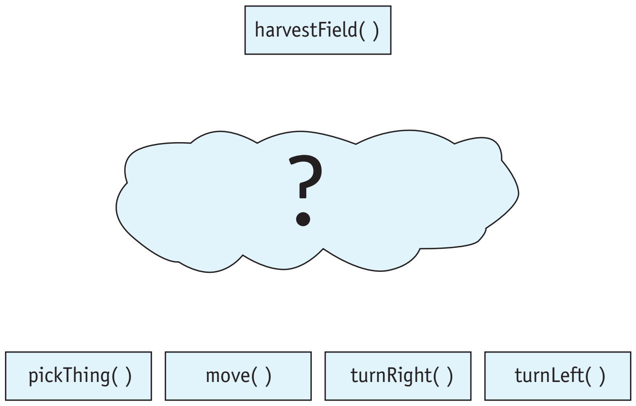
Listing 3-3: *The complete* `Harvester` *class* (continued)

```

   /** Harvest two rows of the field, returning to the same avenue but one
   *   street farther south. The robot must be facing east. */
   public void harvestTwoRows()
   { this.harvestOneRow();
     this.goToNextRow();
     this.harvestOneRow();
   }

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

   /** Go one row south and face west. The robot must be facing east. */
   public void goToNextRow()
   { this.turnRight();
     this.move();
     this.turnRight();
   }

   /** Go one row south and face east. The robot must be facing west. */
   public void positionForNextHarvest()
   { this.turnLeft();
     this.move();
     this.turnLeft();
   }

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

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`.

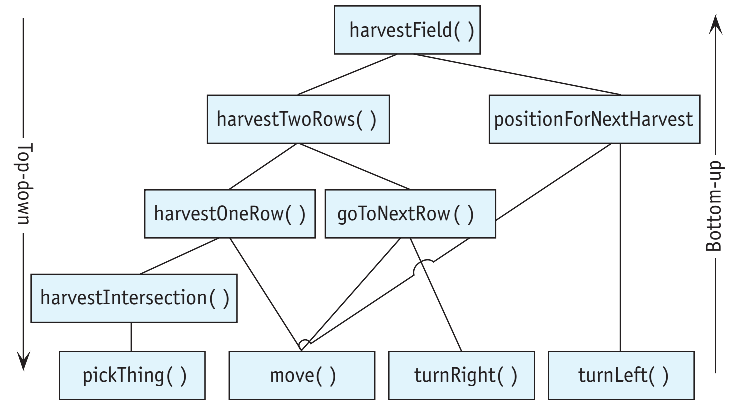
Figure 3-5 shows the situation near the beginning of the design process. We know we want a method to harvest a field and we know that robots can move, pick things up, turn left, and so on. The question is, how do we bridge the gap between them? Stepwise refinement helps fill in intermediate methods,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

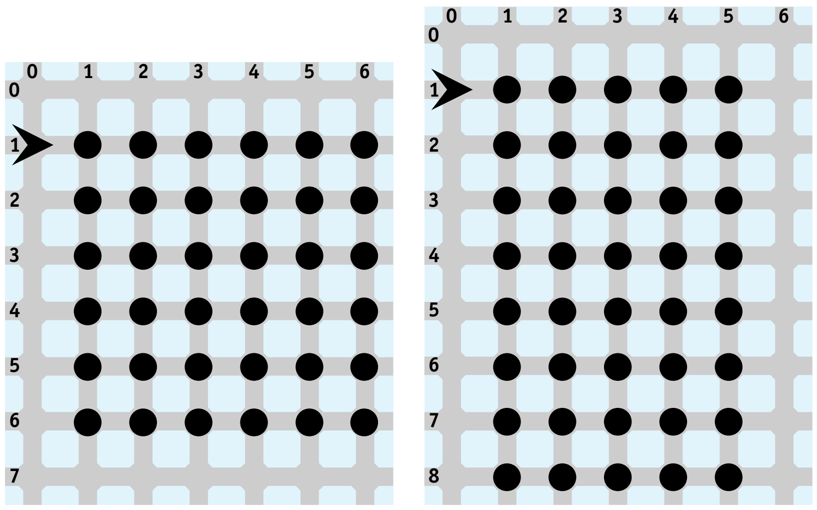
The first situation is one in which the original task has really changed, and it therefore makes sense to change the `Harvester` class itself. In this case, harvesting longer rows can be easily accommodated by adding the following statements to the `harvestOneRow` method:

```
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*

```
import becker.robots.*;

/** A kind of Harvester robot that harvests fields with 6 things per row rather than just 5.
 *
 *  @author Byron Weber Becker */
public class LongRowHarvester extends Harvester
{ /** Construct the harvester. */
  public LongRowHarvester(City acity,
                                     int str, int ave, Direction dir)
  { super(acity, str, ave, dir);
  }
```

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

```

    /** Override the harvestOneRow method to harvest the longer row. */
    public void harvestOneRow()
    { super.harvestOneRow();              // harvest first 5 intersections
      this.move();                        // harvest one more
      this.harvestIntersection();
    }
}
```

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.

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.

KEY IDEA

Pseudocode is a blend of natural and programming languages.

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.

There are several advantages to using pseudocode:

- Pseudocode helps us think more abstractly. As we discussed briefly in Section 1.1.1, abstractions allow us to "chunk" information together into higher level pieces so that we don't need to remember as much. In this case, pseudocode enables us to chunk together many lower-level steps into a single higher-level step, such as `pick all the things in one row`. Such higher-level thinking, however, comes at a cost: less precision. This lack of precision may allow us to accidentally slip in a "miracle" (see the cartoon in Figure 3-3), but overall, the benefits of using pseudocode outweigh the costs.
- Pseudocode allows us to simulate, or trace, our program very early in its development. We can trace the program after only scratching out a few lines on paper. If we find a bug, it is much easier to change and fix it than if we had invested all the time and energy into obeying the many details of the Java language.
- If we are working with other people, even nontechnical users, pseudocode can provide a common language. With it, we can describe the algorithm to others. They might see a special case we missed or a more efficient approach, or even help implement it in a programming language.
- Algorithms expressed with pseudocode can be converted into any computer programming language, not just Java.

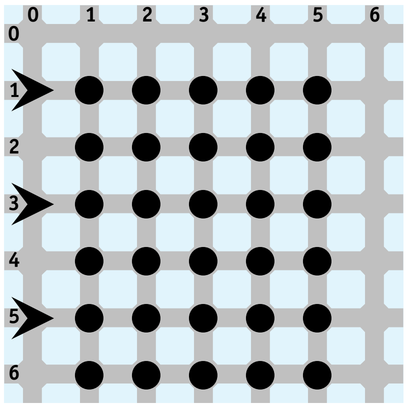
Pseudocode's usefulness increases as the complexity of the algorithm you are designing increases. In the next chapter, we will introduce Java constructs that allow us to choose whether to execute some statements. Other constructs allow us to repeat statements. These constructs are very powerful and vital to writing interesting programs—but they also add complexity, a complexity that pseudocode can help manage in the early stages of programming.

3.5 Variations on the Theme

Consider again the field harvesting task discussed in Section 3.2. There are many variations. Perhaps several robots are available to perform the task, or instead of harvesting things, the robot needs to plant things. This section explores these variations. In the process, we will see that early in the development of the `Harvester` class we made a key assumption that we should use a single robot. This assumption needlessly complicated the program; we should have explored more alternatives. We will also see how to make the computer appear to do several things at once, such as six robots all harvesting a row of things simultaneously.

3.5.1 Using Multiple Robots

One approach to solving the harvesting problem is to use several robots, which we briefly considered early in the process. In this approach, each robot harvests only a part of the field. For example, our `main` method could be modified to instantiate three robots, each of which harvests two rows. The initial situation is shown in Figure 3-8 and in the program in Listing 3-5. `mark` will harvest the first two rows; `lucy` the middle two rows; and `greg` the last two rows. Of course, the work does not need to be divided evenly. If there were only two robots, one could harvest two rows, and the other could harvest four rows.



(figure 3-8)

Harvesting a field with three robots each harvesting two rows

Listing 3-5: *The `main` method for harvesting a field with three robots*

FIND THE CODE

ch03/ harvestWithThree/

```
import becker.robots.*;

/** Harvest a field of things using three robots.
 *
 * @author Byron Weber Becker */
public class HarvestTask extends Object
{
  public static void main(String[] args)
  {
    City stLouis = new City("Field.txt");
    Harvester mark = new Harvester(
                          stLouis, 1, 0, Direction.EAST);
    Harvester lucy = new Harvester(
                          stLouis, 3, 0, Direction.EAST);
    Harvester greg = new Harvester(
                          stLouis, 5, 0, Direction.EAST);
```

Listing 3-5: *The* `main` *method for harvesting a field with three robots* (continued)

```

      mark.move();
      mark.harvestTwoRows();
      mark.move();

      lucy.move();
      lucy.harvestTwoRows();
      lucy.move();

      greg.move();
      greg.harvestTwoRows();
      greg.move();
   }
}
```

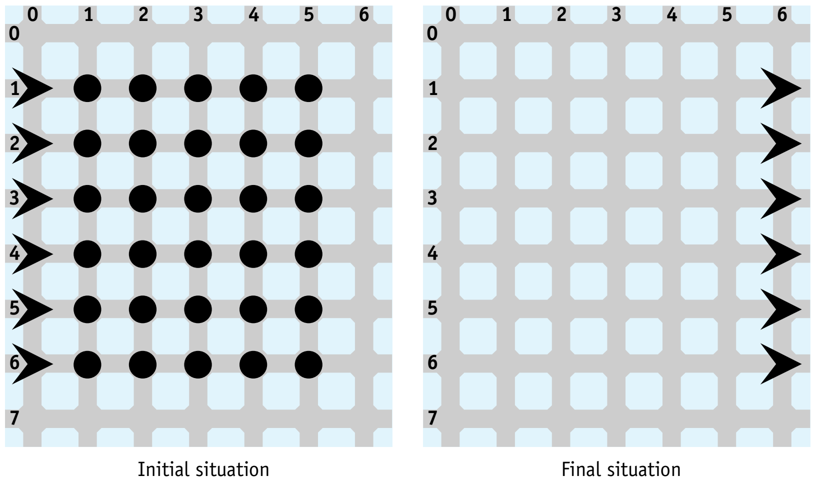
FIND THE CODE

ch03/harvestWithSix/

In fact, the original problem does not specify the number of robots to use, where they start, or where they finish. Perhaps the simplest solution is to have six robots each harvesting one row, and ending on the opposite side of the field. The initial and final situations are shown in Figure 3-9. If we had chosen this solution, the `Harvester` class would have consisted of only `harvestOneRow` and `harvestIntersection`—much simpler than what we actually implemented.

(figure 3-9)

Harvesting with six robots



3.5.2 Multiple Robots with Threads (advanced)

In the previous example, which uses six robots, one robot finishes its entire row before the next one begins to harvest its row. The entire task takes about six times as long as harvesting a single row, even though we have six robots.

If we were paying a group of people an hourly wage to perform this task, we would be pretty upset with this strategy. We would want them working simultaneously so that the entire job is done in about the same amount of time it takes one person to harvest one row.

In this section, we'll explore how to make the robots (appear to) do their work simultaneously. This material is normally considered advanced, but robots provide a clear introduction to these ideas, and it's a fun way to stimulate your thinking about other ways to do things. Check with your instructor to find out if he or she expects you to know this material.

Example: ThreadedRowHarvester

When you have several robots working simultaneously, each robot must be self-contained. The `main` method will start each robot, after which your robots will perform their tasks independently. This approach implies that each robot must be instantiated from a subclass of `Robot`, which "knows" what to do without further input from the program. We'll call this subclass `ThreadedRowHarvester`.

KEY IDEA

The `run` method contains the instructions the thread will execute.

The instructions each robot should execute after it's started are placed in a specially designated method named `run` in the `ThreadedRowHarvester`. The `run` method is free to call other methods to get the job done. In our case, we call the `HarvestOneRow` and `move` methods, as shown in the following method. The `run` method should be inserted in the `ThreadedRowHarvester` class. `harvestOneRow` is defined as in the `Harvester` class.

```
/** What the robot does after its thread is started. */
public void run()
{ this.move();
  this.harvestOneRow();
  this.move();
}
```

PATTERN

Multiple Threads

In the `main` method, we need to construct six `ThreadedRowHarvester` robots, one for each row. However, instead of instructing each robot to harvest a row, we start each robot's thread. The `run` method defined earlier then instructs the robot what to do. A thread is started with two statements, one to create a `Thread` object and one to call its `start` method. For a robot named `karel`, use the following statements:

FIND THE CODE

ch03/ harvestWithSix Threads/

```
ThreadedRowHarvester karel = new ThreadedRowHarvester(...);
...
Thread karelThread = new Thread(karel);
karelThread.start();
```

The `start` method in the last statement invokes the `run` method, which contains the instructions for the robot. For this strategy to work, the `Thread` class must be assured that the `ThreadedRowHarvester` class actually has a `run` method. You do so by adding `implements Runnable` to the line defining the class:

```
public class ThreadedRowHarvester extends Robot
                                  implements Runnable
```

This statement is your promise to the compiler that `ThreadedRowHarvester` will include all of the methods listed in the `Runnable` interface. The `run` method is the only method listed in the documentation for `Runnable`.

In summary, three things need to be completed to start a thread:

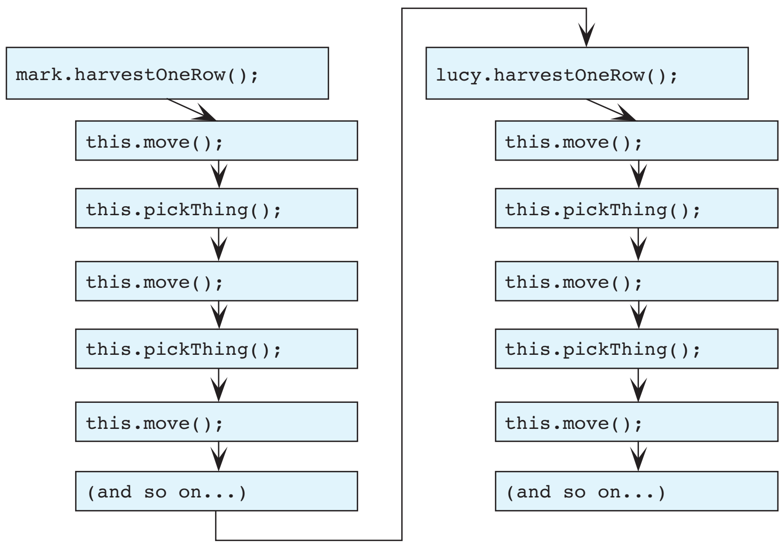
- Include the instructions for the robot in a specially designated method called `run`.
- Implement the interface `Runnable` to tell Java that your class is set up to run in a thread.
- Start the thread.

In this example, each thread performs identical tasks, which need not be the case. We could, for instance, set up two threads with robots harvesting two rows each, and two more threads with robots harvesting one row each.

About Threads

A thread starts a new flow of control. We learned in the Sequential Execution pattern that each flow of control is a sequence of statements, one after the other, where each statement finishes before the next one begins.

The `main` method begins execution in its own thread. As long as we don't start any new threads, execution proceeds one statement after another, as shown in Figure 3-10. This figure supposes that we have two robots named `mark` and `lucy`. The `main` method first calls `mark.harvestOneRow();` and then `lucy.harvestOneRow();`. Between these calls, many other statements are executed, one after the other.



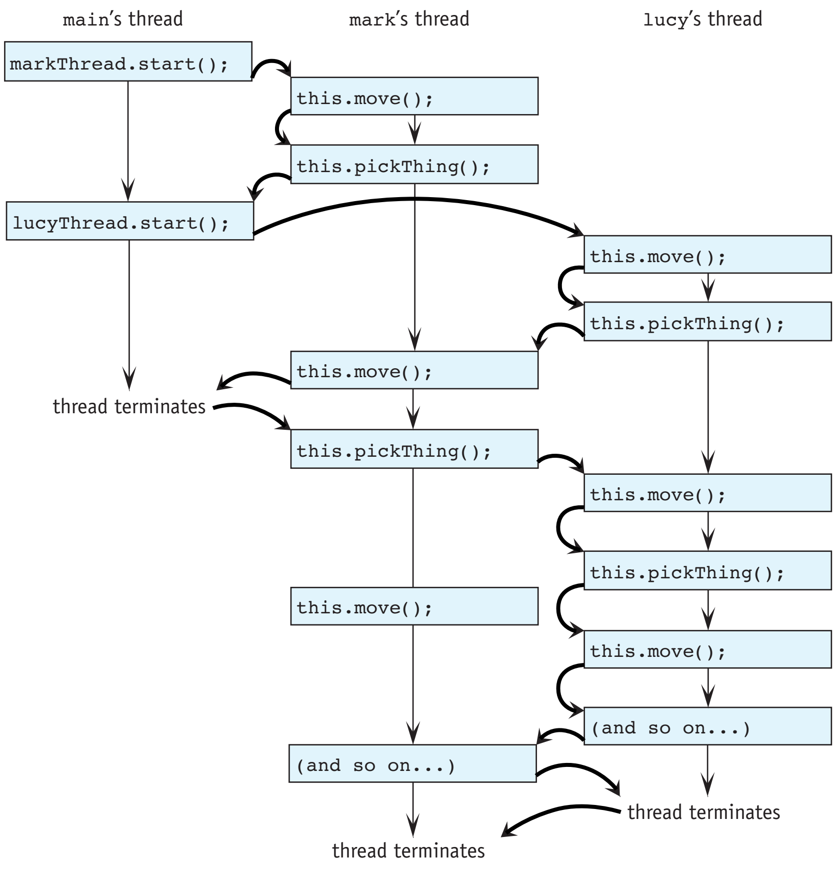
(figure 3-10)

Flow of control with only one thread

When we have two or more flows of control, execution switches among them. The statements *within* each flow of control still execute in order with respect to each other, but statements from a different thread might execute between them. This concept is illustrated in Figure 3-11.

The `main` method's flow of control starts a thread for `mark` and then for `lucy`, represented by the light arrow between the two left-most boxes.

But now that we have three threads of control (one for `main`, one for `mark`, and one for `lucy`), the execution switches between all three threads, as represented by the heavier arrows. Execution switches among the threads so quickly that it *appears* that all the robots are moving simultaneously, though they are not (unless you are fortunate enough to have a computer with at least as many processors as the program has threads). The computer's operating system ensures that each thread runs at least a little bit before stopping it and starting another thread. It also ensures that every thread is eventually run.



(figure 3-11)

One possible flow of control with three threads

Notice that although execution switches among the threads, the statements within each thread are still executed in the same order as before. The only difference is that statements from another thread might be executed between the statements.

Complexities

This simple example glosses over some complexities. For instance, each robot's task in these examples is independent of the tasks performed by the other robots. If a seventh robot collected all the things harvested by the first six robots, it would need a way to wait for those robots to finish their task before starting.

In the next chapter, we will explore ways that programs can make decisions. For example, a robot can check if a `Thing` is present on the intersection. Suppose `mark` is programmed to check for a `Thing` on the current intersection. If there is one, `mark` picks it up; otherwise, `mark` goes on to the next intersection. But the check is in one program statement and the call to `pickThing` is in another. `lucy`, running in another thread, might come along and snatch the thing between those two statements. So the thing `mark` thought was there disappears, and `mark` breaks when it executes `pickThing`.

In spite of these and other complexities, threads are a useful tool in many applications. For example, animations run in their own threads. Many word processors figure out page breaks in a separate thread so that the user can continue typing at the same time. Printing usually has a separate thread so that the user can do other work instead of waiting for a slow printer. Graphical user interfaces usually run in one or more threads so that they can continue to respond to the user even while the program is carrying out a time-consuming command.

LOOKING AHEAD

In Section 10.7, we'll learn how to use a thread to perform animation in a user interface.

3.5.3 Factoring Out Differences

Suppose that instead of picking one thing from each intersection in the field, we want to plant a thing at each intersection. Other alternatives include picking two things or counting the total number of things in the field.

Each of these programs is similar to the harvesting task. In particular, the part that controls the movement of the robot over the field is the same for all of these problems; it is only the task at each intersection that differs. The original task of harvesting things is only one example of a much more general problem: traversing a rectangular area and performing a task at each intersection.

KEY IDEA

Think about variations of the problem early in the design.

If we started with this view of the problem, we might design the program differently. Instead of solving the harvesting problem directly, we could design a `TraverseAreaRobot` that traverses a rectangular area. At each intersection, it calls a method named `visitIntersection` that is defined to do nothing, as follows:

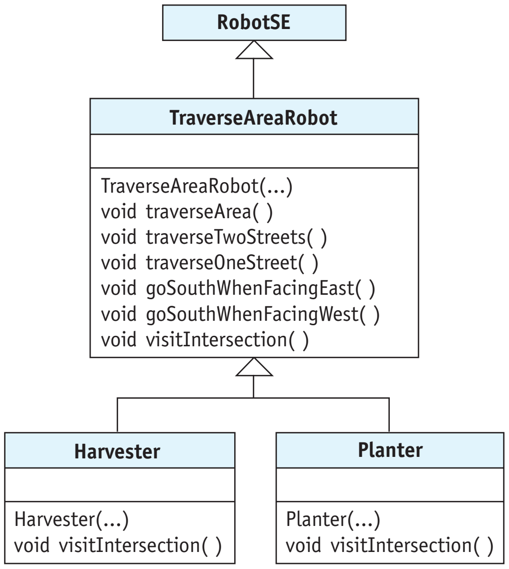
```
public void visitIntersection()
{
}
```

By overriding this method in different subclasses, we can create robots that harvest each intersection or plant each intersection, and so on. A class diagram illustrating this approach is shown in Figure 3-12.

It may seem strange to include a method like `visitIntersection` that does nothing. However, this method must be present in `TraverseAreaRobot` because other methods in that class call it. On the other hand, we don't know what to put in the method because we don't know if the task is harvesting or planting the field, and so we simply leave it empty, ready to be overridden to perform the appropriate action.

(figure 3-12)

Class diagram for a group of classes for working with fields



RobotSE
TraverseAreaRobot
TraverseAreaRobot(...)
void traverseArea( )
void traverseTwoStreets( )
void traverseOneStreet( )
void goSouthWhenFacingEast( )
void goSouthWhenFacingWest( )
void visitIntersection( )
Harvester
Harvester(...)
void visitIntersection( )
Planter
Planter(...)
void visitIntersection( )

Of course, we could have solved the planting problem by extending the `Harvester` class and overriding `harvestIntersection`. The approach shown in Figure 3-12 differs from that in two ways. The first difference is that we planned for various tasks to occur at each intersection and named the methods accordingly. It is confusing to override a method named `harvestIntersection` so that it plants something instead of harvesting.

PATTERN

Template Method

The second difference is that the `TraverseAreaRobot` class deliberately does nothing when it visits an intersection. Instead, `visitIntersection` serves as an intentional point where subclasses can modify the behavior of `traverseArea`. In fact, the documentation for `visitIntersection` and `traverseArea` should explicitly describe the possibilities of overriding the method. In a sense, `traverseArea` is a template for a common activity, which is modified by overriding `visitIntersection`.

3.6 Private and Protected Methods

The `TraverseAreaRobot` class makes available six new services: `traverseArea`, `traverseTwoStreets`, `traverseOneStreet`, `goSouthWhenFacingEast`, `goSouthWhenFacingWest`, and `visitIntersection`. Should these all be available to all clients? For example, should a client such as `main` be able to invoke the `goSouthWhenFacingEast` method? After all, it was developed as a helper method, not as a service to be offered by a `TraverseAreaRobot`. Perhaps a client should not be allowed to invoke it.

Recall that a client is an object that uses the services of another object, called the server. The client uses the server's services by invoking its corresponding method with the Command Invocation pattern described in Section 1.7.3:

```
«objectReference».«methodName»(«parameterList»);
```

The client is the class that contains code, such as `karel.move()`, `joe.traverseArea()`, or even `this.goSouthWhenFacingEast()`. In these cases, `karel`, `joe`, and `this` are the *«objectReference»*s.

Java has a set of **access modifiers** that control which clients are allowed to invoke a method. The access modifier is placed as the first keyword before the method signature.

KEY IDEA

Public methods may be invoked by any client.

So far, we have used the access modifier `public`, as in `public void traverseArea()`. The keyword `public` allows any client to access the method. Like a public telephone, anyone who comes by can use it.

KEY IDEA

Private methods can only be invoked by methods defined in the same class.

The access modifier `private` is at the other end of the scale. It says that no one except clients who belong to the same class, may invoke the method, and that the method may not be overridden. Staying private is what we want for many helper methods. `goSouthWhenFacingEast`, for example, was designed to help `traverseTwoStreets` do its work; it should not be called from outside of the class where it was declared. It should therefore be declared as follows:

```
private void goSouthWhenFacingEast()
```

KEY IDEA

Protected methods can be used from subclasses.

A middle ground is to use the `protected` access modifier. Protected methods may be invoked from clients that are also subclasses. Like all methods, protected methods can also be invoked from within the class defining them.

Using `protected` on the `traverseOneStreet` and `visitIntersection` methods would be appropriate. It would allow us to override and use those methods in a subclass to traverse longer streets. We also did this in Section 3.3.4 when we overrode `harvestOneRow` to harvest a longer row. This approach is shown in Listing 3-6 and Listing 3-7. Listing 3-8 shows code that does *not* compile because it attempts to use `protected` and `private` methods.

Listing 3-6: *Using* `protected` *and* `private` *access modifiers in* `TraverseAreaRobot`

```
public class TraverseAreaRobot extends RobotSE
{ public TraverseAreaRobot(...)          {     ...     }

  public void traverseArea()             {     ...     }

  private void traverseTwoStreets()      {     ...     }

```

Listing 3-6: *Using* `protected` *and* `private` *access modifiers in* `TraverseAreaRobot` (continued)

```
  protected void traverseOneStreet()     {     ...     }

  private void goSouthWhenFacingEast()   {     ...     }

  private void goSouthWhenFacingWest()   {     ...     }

  protected void visitIntersection()     {     ...     }
}
```

Listing 3-7: *Using* `protected` *methods in a subclass of* `TraverseAreaRobot`

```
public class TraverseWiderAreaRobot extends TraverseAreaRobot
{ public TraverseWiderAreaRobot(...)     {     ...     }

  protected void traverseOneStreet()
  { super.traverseOneStreet();       // traverse first 5 intersections
    this.move();                     // traverse one more
    this.visitIntersection();
  }
}
```

Listing 3-8: *A program that fails to compile because it attempts to use* `private` *and* `protected` *methods*

```
public class DoesNotWork
{ public static void main(String[] args)
  { ...
    TraverseAreaRobot karel = new TraverseAreaRobot(...);
    ...
    karel.traverseArea();             // works—method is public
    karel.traverseTwoStreets();       // compile error
                                      // traverseTwoStreets is private
    karel.visitIntersection();        // compile error
                                      // visitIntersection is protected
   }
}
```

It is also possible to omit the access modifier. The result is called "package" access. It restricts the use of the method to classes in the same package. The `becker.robots` package sometimes uses package access to make services available within all classes in the package that should not be available to students. For example, `Robot` actually has a `turnRight` method (contrary to what you read in Section 1.2.3), but it has package access, so most clients can't use it. `RobotSE`, however, is in the same package and thus has access to it. It makes `turnRight` publicly available with the following method, which overrides `turnRight` with a less restrictive access modifier.

```
public void turnRight()
{ super.turnRight();
}
```

Students should not need to use package access.

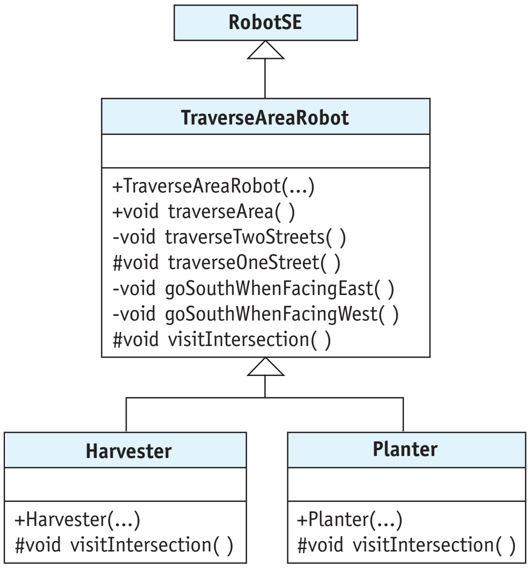
KEY IDEA

Declare methods to be `private` unless you have a specific reason to do otherwise.

As a rule of thumb, beginning programmers should declare methods as `private` except in the following cases:

- The method is specifically designed to be a public service. In this case, you should declare it as `public`.
- The method is used only by a subclass. In this case, you should declare it as `protected`.

Access modifiers are often shown in class diagrams with the symbols `+`, `#`, and `-`. They stand for `public`, `protected`, and `private` access, respectively. Figure 3-13 shows a class diagram for the `Harvester` class that includes these symbols.


RobotSE
TraverseAreaRobot
+TraverseAreaRobot(...)
+void traverseArea( )
-void traverseTwoStreets( )
#void traverseOneStreet( )
-void goSouthWhenFacingEast( )
-void goSouthWhenFacingWest( )
#void visitIntersection( )
Harvester
+Harvester(...)
#void visitIntersection( )
Planter
+Planter(...)
#void visitIntersection( )

(figure 3-13)

Showing the accessibility of the helper methods in the `TraverseAreaRobot` *class*

3.7 GUI: Using Helper Methods

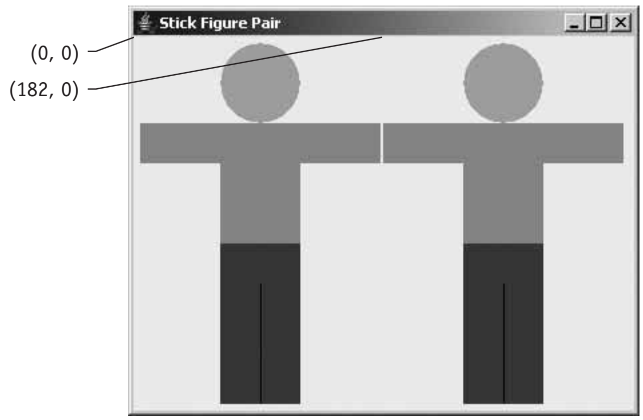
KEY IDEA

Helper methods are a powerful way to help organize a class.

Stepwise refinement and helper methods are useful in graphics programs, too. For example, consider the pair of stick figures in Figure 3-14. They are based on the stick figure program written in Section 2.7. The `paintComponent` method from that program is reproduced in Listing 3-9, but lines 17 to 29 need to somehow be executed twice to draw both figures. Simply executing the same code twice isn't enough—that would just draw one figure on top of the other. We also need to offset the second figure so that they stand side-by-side.

(figure 3-14)

Pair of stick figures



FIND THE CODE

ch02/stickFigure/

Listing 3-9: *The code to draw a single stick figure at a predetermined location*

```
12 // Paint a stick figure.
13 public void paintComponent(Graphics g)
14 { super.paintComponent(g);
15
16    // Paint the head.
17    g.setColor(Color.YELLOW);
18    g.fillOval(60, 0, 60, 60);
19
20    // Paint the shirt.
21    g.setColor(Color.RED);
22    g.fillRect(0, 60, 180, 30);
23    g.fillRect(60, 60, 60, 90);
24
25    // Paint the pants.
26    g.setColor(Color.BLUE);
27    g.fillRect(60, 150, 60, 120);
```

Listing 3-9: *The code to draw a single stick figure at a predetermined location* (continued)

```
28    g.setColor(Color.BLACK);
29    g.drawLine(90, 180, 90, 270);
30  }
```

One approach is to duplicate lines 17 to 29 inside the `paintComponent` method and adjust the arguments to offset the second figure. A much better approach is to place lines 17 to 29 inside a helper method. The `paintComponent` method calls the method twice to draw the two figures—except that we once again have the problem of offsetting the second figure to stand beside the first one. We could make two helper methods, one for each figure, but they would be almost identical.

The best solution is one helper method that uses parameters to specify the location of the figure. We have already made extensive use of parameters. For example, consider the method calls in lines 17 to 29 of Listing 3-9. They each pass arguments to the method's parameters indicating the location and size of the shape to draw. We will use the same strategy except that instead of drawing a simple oval or rectangle, our method will draw an entire stick figure. We will use parameters only for the location of the stick figure. Using such a helper method, the `paintComponent` method is simplified to the following:

```
1  /** Paint two stick figures
2   *   @param g The graphics context to do the painting. */
3  public void paintComponent(Graphics g)
4  { super.paintComponent(g);
5    this.paintStickFig(g, 0, 0);
6    this.paintStickFig(g, 182, 0);
7  }
```

Line 5 causes a stick figure to be drawn with its upper-left corner placed at (0, 0)—that is, the upper-left corner of the component. Figure 3-14 is annotated with this location. Line 6 causes the second figure to be painted at (182, 0), or 182 pixels from the left and 0 pixels down from the top. This location is also noted in Figure 3-14. The value of 182 was picked because each stick figure is 180 pixels wide, plus two pixels for a tiny gap between them.

Lines 5 and 6 also pass `g`, the `Graphics` object used for painting, as an argument because `paintStickFig` will need it to draw the required shapes.

3.7.1 Declaring Parameters

PATTERN

Parameterized Method

To use arguments such as `g`, `182`, and `0` inside our helper method, we need to declare corresponding parameters. These should look familiar because we have been declaring parameters in `Robot` constructors since the beginning of Chapter 2. The first line of the `paintStickFig` method should be:

```
    private void paintStickFig(Graphics g2, int x, int y)
```

The first part of this line, `private void paintStickFig`, is the same as our Parameterless Command and Helper Method patterns.

LOOKING BACK

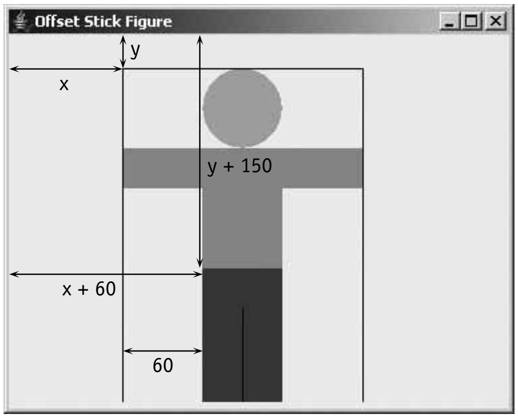
***Type** was defined in Section 1.3.1 as specifying a valid set of values for an attribute. Here it specifies the set of values for the parameter.*

Next come the three parameters. Each specifies a type and a name, and is separated from the next parameter with a comma. `Graphics` is the name of a class and specifies that the first argument to `paintStickFig` must be a reference to a `Graphics` object. This is similar to our `Robot` constructors. There, the first parameter has a type of `City`; consequently, we always pass a `City` object as the first argument. The next two parameters must always be passed integer arguments because they are declared with `int`.

Inside the method, the values passed as arguments will be given the name of the corresponding parameter. If the method is called with `this.paintStickFig(g, 182, 0)`, then inside `paintStickFig`, every time we use the name `x` it will be interpreted as `182`—the value passed to it.

3.7.2 Using Parameters

With this background, we can rewrite the method to use the parameters to specify the stick figure's position. Each time we refer to an *x* or a *y* location in drawing the stick figure, we add the appropriate `x` or `y` parameter. This action offsets the figure, as shown in Figure 3-15. Adding two numbers together uses the plus sign, and if one of the "numbers" happens to be a parameter, Java will use the number it represents (in this case, the number passed to it as an argument). The revised code for `paintStickFig` appears in Listing 3-10.



(figure 3-15)

Offsetting the location of the stick figure with the `x` *and* `y` *parameters*

Consider line 36 to paint the rectangle used for the pants. In the original code, we wrote `g.fillRect(60, 150, 60, 120)` to draw a rectangle 60 pixels from the left side and 150 pixels down from the top. The last two arguments specify that it should be 60 pixels wide and 120 pixels high. In line 36, this is changed to `g2.fillRect(x+60, y+150, 60, 120)`. Now the rectangle starts 60 pixels to the right of `x`. If `x` is passed 0, the pants are painted 60 pixels from the left side of the panel. If `x` is passed 182, the pants are painted 242 (182 + 60) pixels from the left side.

FIND THE CODE

ch03/stickFigure/

Listing 3-10: *A component that paints two stick figures, one beside the other*

```
import java.awt.*;                    // Graphics, Dimension, Color
import javax.swing.*;                 // JComponent

public class StickFigurePair extends JComponent
{
  public StickFigurePair()
  { super ();
    Dimension prefSize = new Dimension(2*180+5, 270);
    this.setPreferredSize(prefSize);
  }

  /** Paint two stick figures
  *   @param g The graphics context to do the painting. */
  public void paintComponent(Graphics g)
  { super.paintComponent(g);
    this.paintStickFig(g, 0, 0);
    this.paintStickFig(g, 182, 0);
  }

  /** Paint one stick figure at the given location.
```

Listing 3-10: *A component that paints two stick figures, one beside the other* (continued)

```
     *  @param g2   The graphics context to do the painting.
     *  @param x    The x coordinate of the upper-left corner of the figure.
     *  @param y    The y coordinate of the upper-left corner of the figure. */
    private void paintStickFig(Graphics g2, int x, int y)
    { // Paint the head.
      g2.setColor(Color.YELLOW);
      g2.fillOval(x+60, y+0, 60, 60);

      // Paint the shirt.
      g2.setColor(Color.RED);
      g2.fillRect(x+0, y+60, 180, 30);
      g2.fillRect(x+60, y+60, 60, 90);

      // Paint the pants.
      g2.setColor(Color.BLUE);
      g2.fillRect(x+60, y+150, 60, 120);
      g2.setColor(Color.BLACK);
      g2.drawLine(x+90, y+180, x+90, y+270);
    }
  }
```

Using a helper method helps keep the `paintComponent` method to a reasonable size. By adding parameters to the helper method, we allow the method to be used more flexibly with the result that we only need one helper method instead of two.

3.8 Patterns

This chapter introduced four patterns: Helper Method, Multiple Threads, Template Method, and Parameterized Method.

3.8.1 The Helper Method Pattern

Name: Helper Method

Context: You have a long or complex method to implement. You want your code to be easy to develop, test, and modify.

Solution: Look for logical steps in the solution of the method. Put the code to solve this step in a well-named helper method. For example, if the problem is for a robot to travel in a square pattern, the problem could be decomposed like this:

```
public void squareMove()
{ this.sideMove();
  this.sideMove();
  this.sideMove();
  this.sideMove();
}
```

where `sideMove` is defined as follows:

```
private void sideMove()
{ this.move();
  this.move();
  this.move();
  this.turnLeft();
}
```

Of course, the problem may involve writing several different helper methods. Because helper methods are usually not services the class provides, they should generally be declared `private` or at least `protected`, depending on whether subclasses need to access or override them.

Consequences: Long or complex methods are easier to read, develop, test, and modify when you break them into smaller steps and use helper methods.

Related Patterns: This pattern is almost identical to the Parameterless Command pattern and other method-related patterns we will see in future chapters. The difference is in the intent: helper methods designate methods that exist to perform a piece of a larger operation whereas the Parameterless Command, for example, does not have that connotation.

3.8.2 The Multiple Threads Pattern

Name: Multiple Threads

Context: You have multiple objects such as robots that should appear to carry out their tasks simultaneously.

Solution: Start each of the tasks in its own thread of control. This requires three tasks:

- Write a method named `run`. It contains code to execute in a thread.
- Implement the `Runnable` interface so that Java knows your class is set up to run as a thread.
- Start the thread.

The first two steps are expressed in code according to the following template:

```
public class «className» extends «superclassName»
                             implements Runnable
{ ...
  public void run()
  { «statements to execute inside a separate thread»
  }
}
```

The third step is often included in an instance of the Java Program pattern but can also be used in other contexts.

```
public class «programClassName»
{ public static void main(String[] args)
  {  ...
     «className» «runnableObject» = new «className»(...);
     Thread «threadName» = new Thread(«runnableObject»);
     «threadName».start();
     ...
  }
}
```

The three lines to create the object, create the thread, and start the thread are repeated as many times as there are threads.

Consequences: A separate thread of control is started whose execution is interleaved with the execution of other threads. This is relatively easy as long as the threads cannot interfere with each other. If interference is a possibility, many problems can arise.

Related Patterns: This pattern makes use of common patterns, such as the following:

- Java Program
- Extended Class
- Object Instantiation
- Method Invocation
- Sequential Execution

3.8.3 The Template Method Pattern

Name: Template Method

Context: You have a set of similar classes. Each has a method that does almost the same thing as a corresponding method in the other classes, but not quite. You would like to avoid duplicating the common code so that you only need to write it, debug it, and maintain it once.

Solution: The method that shares the similar code among classes is called the **template method**. Write it using helper methods for the parts that are different from one version to another. However, instead of putting these helper methods in the same class as the template method, put them in subclasses. The subclasses provide the variations in the code that are used to solve the different problems.

LOOKING AHEAD

Methods can be declared `abstract` *instead. We'll learn more in Section 12.1.5.*

To compile the template method, you need to include empty methods with the same names as the helper methods.

For an example, see Section 3.5.3.

Consequences: Writing the common code once helps reduce the effort to write, debug, and maintain it. By explicitly identifying where the differences occur and writing methods for them, it's easier to add a new class that solves another variation of the same problem.

On the other hand, needing to look in a different class for part of the solution to the problem can be confusing.

Related Pattern: This pattern is a specialization of the Extended Class pattern where specific methods are provided for the express purpose of being overridden.

3.8.4 The Parameterized Method Pattern

Name: Parameterized Method

Context: A method might do many variations of its task if it only had some information from its client to say which variation to perform. The different variations are often quantified—how many pixels over to paint a figure, how many times to turn a robot, or how much money to deposit in a bank account.

Solution: Use one or more parameters to communicate information from the client to the method. Use this information to control which of many possible variations of the task to perform.

In general, the method will declare one or more parameters, each having a type and a name. Consecutive pairs are separated with commas, as shown in the following template:

```
public void «methodName»(«paramType1» «paramName1»,
                         «paramType2» «paramName2»,
                         ...
                         «paramTypeN» «paramNameN»)
{ «list of statements, at least some of which use paramName»
}
```

The method is used with a method invocation matching the following template:

```
    «objectReference».«methodName»(«arg1», «arg2», ..., «argN»);
```

where the type of each argument is compatible with the type of the corresponding parameter.

A concrete example of this pattern is illustrated in Listing 3-10.

Consequences: The method is much more flexible than a similar method written without parameters. Parameters give opportunities to the client to influence how the method carries out its task.

LOOKING AHEAD

This pattern will be discussed more fully in Chapters 4 and 6.

Related Patterns: The Parameterized Method pattern is a variation of the other following patterns:

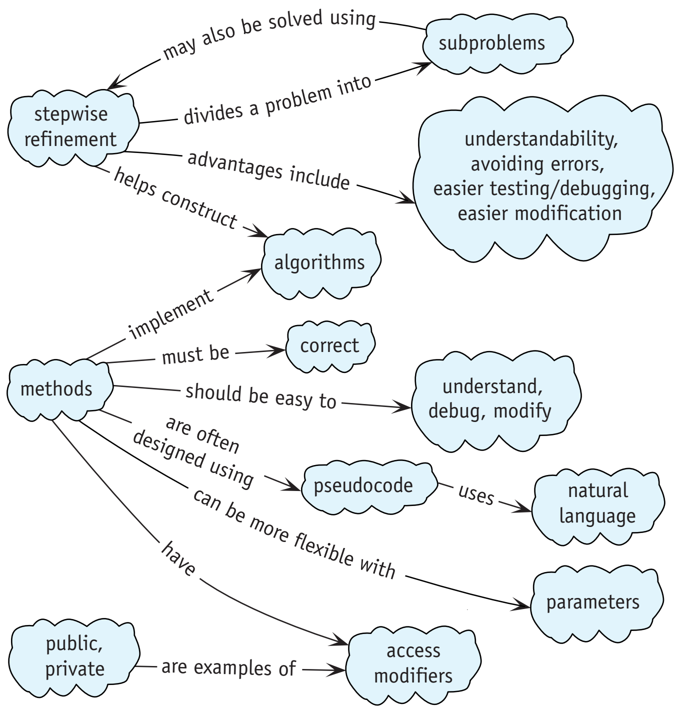
- Parameterless Command
- Helper Method

3.9 Summary and Concept Map

Stepwise refinement is a process of breaking a problem down into smaller and simpler steps until, ultimately, the problems are constrained enough to be directly solved with the primitives at hand. The advantages of using stepwise refinement include code that is easier to understand, test, debug, and modify, precisely because the solution is expressed in terms of logical subproblems discovered by the stepwise refinement process.

Stepwise refinement naturally results in helper methods: methods that exist to help another method by solving a subproblem. Helper methods are often `private`, but may also be `protected`. When they are protected, subclasses can often be used to easily solve variations of the same problem using the Template Method pattern.

Pseudocode is another tool for designing methods. It is a mixture of a natural language, such as English, and a programming language. It allows us to express our algorithms at a high level and reason about them without investing the time and overhead of coding them in a language such as Java.



3.10 Problem Set

Written Exercises

3.1 In Listing 3-3, all of the methods have public access. Is this appropriate? Should any of them have a different access modifier? If so, which ones?

3.2 Examine problem 2.13 again, in which `karel` wrote the message “Hello” using `Things`. What helper methods would you suggest?

3.3 Suppose the `TraverseAreaRobot` class is extended to create the `Harvester` class, as described in Section 3.5.3. What happens if the method in `Harvester` is misspelled vistIntersection?

3.4 Consider the choice of access modifiers for `TraverseAreaRobot` suggested in Listing 3-6. Explain their effects on the creation of a subclass to harvest every other row of the field.

Programming Exercises

3.5 If necessary, download the source code for the examples and find `ch03/debugging/`. It contains two kinds of robots, both of which perform the same task. However, `MonolithicBot` contains a single method named `doit`.

`StepRefineBot` also contains a method named `doit`, but it was developed using stepwise refinement. Run the program once to see what it is supposed to do.

Work with a partner for the remainder of this problem. Assign `MonolithicBot` to one partner and `StepRefineBot` to the other. Each of you makes one small, secret change to the other's robot. "One small change" is defined as deleting a statement, adding a statement, or substituting a new statement for an existing statement. Run each program and time how long it takes each person to find the mistake in their assigned robot. Repeat five times. Summarize your results.

Programming Projects

3.6 Rewrite the `harvestField` method using a different stepwise refinement. In particular, move the robot over the field in a spiral pattern, as shown in Figure 3-16.

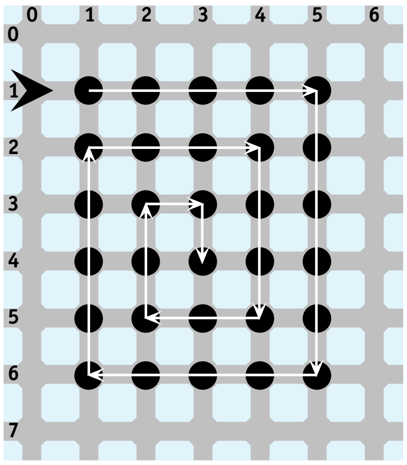
a. Write pseudocode to solve the problem using this idea.

b. Analyze the solution for strengths and weaknesses.

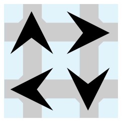
c. Write a program implementing your solution.

(figure 3-16)

Harvesting in a spiral



3.7 Program a robotic synchronized swimming team. The team has four members that begin their routine as shown in Figure 3-17 in the middle of the pool. Each swimmer goes through the same motions: a small counter-clockwise square, a large counter-clockwise square, turn around, a small clockwise square, and finally a large counter-clockwise square. Each square leaves the swimmer in the same position as when it started the square. Small squares involve moving once on each side; for large squares, the swimmers move twice. Start each swimmer in its own thread (see Section 3.5.2).



(figure 3-17)

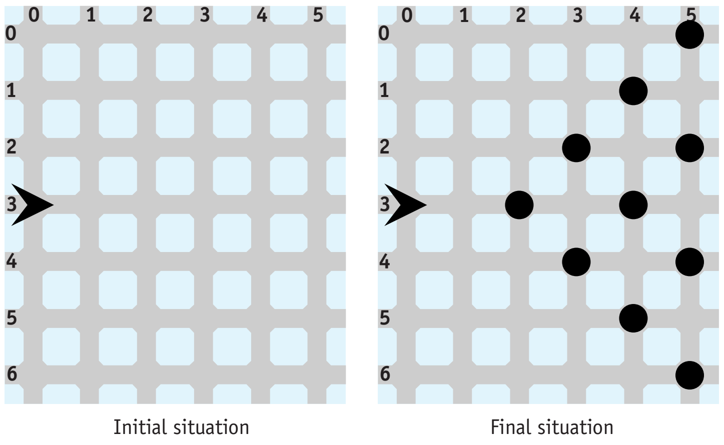
Initial situation for a team of synchronized swimmers

3.8 `karel` sometimes works as a pinsetter in a bowling alley. Examine the initial and final situations shown in Figure 3-18, and then complete the following tasks:

a. Develop pseudocode for two different refinements of a method named `setPins`.

b. Analyze both solutions for strengths and weaknesses.

c. Write a program that implements one of your solutions.



(figure 3-18)

Initial and final situations for `setPins`

3.9 The CEO of a highly successful local software company has a plus-shaped wall in her garden, as shown in Figure 3-19. She would like to use robots to plant one and only one `Thing` at each location around the wall. Robots will always start with enough `Things` to finish their task (look in the documentation for a constructor to specify how many `Things` a robot starts with).

a. Use a single robot to do the planting. It begins and ends at (0, 0).

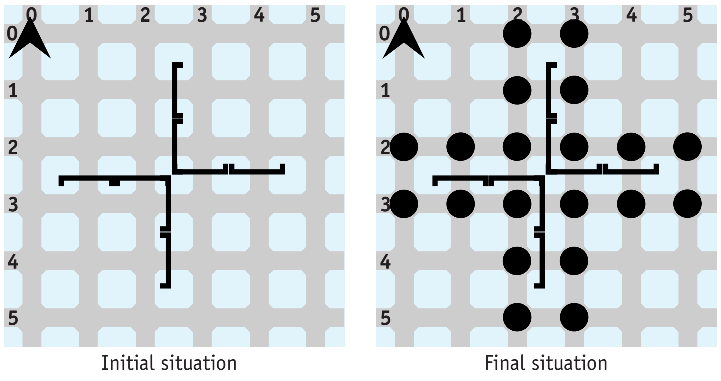
b. Use a team of four robots. You may choose their beginning and ending positions.

c. Use a team of eight robots. You may choose their beginning and ending positions.

d. Use threads so that a team of robots plants the garden simultaneously.

(figure 3-19)

Planting things in a garden



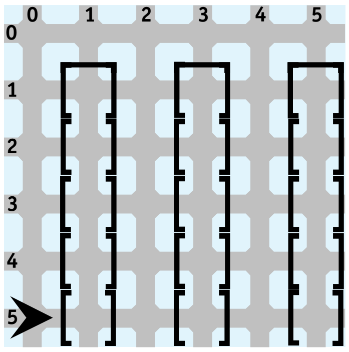
3.10 Spiderman has a new superhero rival: Spiderbot. Just like Spiderman, Spiderbot can climb tall buildings, as shown in Figure 3-20. However, Spiderbot must stay as close to a building as possible as it climbs, and it can't jump between buildings.

a. Write a `SpiderBot` class that has a `climbBuilding` method. Use it to instruct Spiderbot to climb over the three buildings. Use a file to place the walls of the buildings. Consult the online documentation for the `City` constructors for the file format.

b. Extend the `City` class to make `CityBuilder`. The `CityBuilder` class has a method named `placeBuilding` that takes one parameter: the avenue where the building should be placed. Use it to build the city.

(figure 3-20)

Series of skyscrapers for Spiderbot to climb



3.11 Section 3.5.3 describes how to use `TraverseAreaRobot` as a template for classes that do variations of the same task. Implement the `TraverseAreaRobot` class.

a. Extend `TraverseAreaRobot` to create a class named `Harvester`. Instances of `Harvester` will pick one `Thing` from each intersection of the area traversed.

b. Extend `TraverseAreaRobot` to create a class named `Planter`. Instances of `Planter` will put one `Thing` on each intersection of the area traversed.

c. Extend `TraverseAreaRobot` to create a class named `BumperCropHarvester`. Instances of this class will collect five `Things` from each intersection of the area traversed.

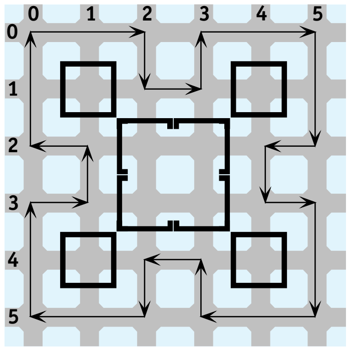
d. Extend `TraverseAreaRobot` to create a class named `SparseRowHarvester`. Instances of this class will harvest `Things` from every other row of the area traversed. The field should have 12 rows.

3.12 King Java's castle, shown in Figure 3-21, needs to be guarded. Write a `GuardBot` class to patrol the castle walls in the pattern shown. Be sure to use appropriate stepwise refinements. Choose an appropriate place for the guard to begin its duties.

a. Write a `main` method that uses a single `GuardBot` to guard the castle.

b. Write a `main` method that uses four `GuardBots` to patrol the castle, one on each side.

c. Modify your solution so that all the guards patrol their wall simultaneously.



(figure 3-21)

King Java's castle

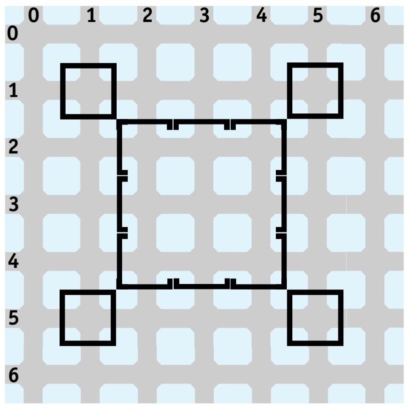
3.13 King Java's neighbor, King Caffeine, is impressed with the `GuardBots` developed in Problem 3.12. He wants to hire four `GuardBots` to patrol his castle. However, his castle is larger, as shown in Figure 3-22.

a. Refer to the class diagram in Figure 3-12, which discusses the Template Method pattern. Adapt it to this problem, showing the relationships and methods needed for three classes: `GuardBotTemplate`, `LongWallGuard`, and `ShortWallGuard`.

b. Using the Template Method pattern, develop three classes named `GuardBotTemplate`, `LongWallGuard`, and `ShortWallGuard`. Write a `main` method that creates castles for both King Caffeine and King Java and then uses four `LongWallGuards` to patrol King Caffeine's castle and four `ShortWallGuards` to patrol King Java's castle.

(figure 3-22)

King Caffeine's castle



3.14 Create a program that draws four copies of the Olympic rings, one in each corner of the component. The colors of the five rings, from left to right, are blue, yellow, black, green, and red. The rings may simply overlap rather than interlock, as in the official symbol.