4 Decomposing Code into Components

4.1 Procedural Decomposition

4.1.1 Printing Rectangles

In this chapter, we learn how to decompose a source code into multiple methods.

We previously studied programs that draw shapes using multiple `System.out.println` statements. Consider, this time, drawing a rectangle. Suppose we draw a rectangle with 3 × 3 white space characters and surrounding borders, as shown next:

```
+---+
|   |
|   |
|   |
+---+
```

We can accomplish the task using five `System.out.println` statements corresponding to the five horizontal strips of the shape, as shown next.[1] For clarification, the five `System.out.println` statements are marked with line comments indicating their correspondence to the strips.

```java
public class Rectangle
{
  public static void main( String[] args )
  {
    System.out.println( "+---+" );    // top line
    System.out.println( "|   |" );    // middle section 1
    System.out.println( "|   |" );    // middle section 2
    System.out.println( "|   |" );    // middle section 3
    System.out.println( "+---+" );    // bottom line
  }
}
```

Listing 4.1 The source code for a program that produces a 3 × 3 white space rectangle with encompassing borders

[1]The program name for this is Rectangle, not Square, although the size of the white-area is 3 × 3. This is because the shape does not look like a square, which is because the computer characters have longer height than width.

M. Ogihara, *Fundamentals of Java Programming*,
https://doi.org/10.1007/978-3-319-89491-1_4

Suppose, instead of just one rectangle, we want to print the same rectangle three times, on top of one another, as shown next.

```
+---+
|   |
|   |
|   |
+---+
+---+
|   |
|   |
|   |
+---+
+---+
|   |
|   |
|   |
+---+
```

We can accomplish the task by repeating the five statements two more times:

```java
public class Rectangle00
{
  public static void main( String[] args )
  {
    System.out.println( "+---+" );    // top line
    System.out.println( "|   |" );    // middle section 1
    System.out.println( "|   |" );    // middle section 2
    System.out.println( "|   |" );    // middle section 3
    System.out.println( "+---+" );    // bottom line
    System.out.println( "+---+" );    // top line
    System.out.println( "|   |" );    // middle section 1
    System.out.println( "|   |" );    // middle section 2
    System.out.println( "|   |" );    // middle section 3
    System.out.println( "+---+" );    // bottom line
    System.out.println( "+---+" );    // top line
    System.out.println( "|   |" );    // middle section 1
    System.out.println( "|   |" );    // middle section 2
    System.out.println( "|   |" );    // middle section 3
    System.out.println( "+---+" );    // bottom line
  }
}
```

Listing 4.2 A program that produces three rectangles

Alternatively, the same output can be generated using a source code that uses, three times, a method that prints just one rectangle.

A method that prints a single rectangle can be defined as follows (where we use the name `oneRectangle` for the method):

```java
  public static void oneRectangle()
  {
    System.out.println( "+---+" );
    System.out.println( "|   |" );
    System.out.println( "|   |" );
    System.out.println( "|   |" );
    System.out.println( "+---+" );
  }
```

The method declaration conforms to the format we saw in Chap. 1:

```
ATTRIBUTES RETURN_TYPE METHOD_NAME( PARAMETERS )
```

`oneRectangle` is a public executable method requiring no parameters, with the return type of `void`.

As mentioned earlier, one can define multiple methods in a source code. If `oneRectangle` is the name of a method defined in a source code, all the methods appearing in the same source code can execute the code `oneRectangle`. This is done by stating the name, and then attaching a pair of parentheses followed by a semicolon:

```
public static ... fooBar( ... )
{
  ...
  oneRectangle();
  ...
}
```

We call the action of executing a code (written in a method) by stating its name a **method call**.

Unlike the concept of method calls that we saw in Chap. 3, the method call `oneRectangle` stands alone and does not require an instantiation of an object.

When a method call occurs, the present execution of the code is suspended temporarily, and the code of the method that has been called is executed. Once the execution of the method is completed, the suspended execution resumes.

Returning to the task of printing the three rectangles: now that we have written `oneRectangle` as he method for printing one rectangle, the task can be accomplished by three successive calls of `oneRectangle`, as shown next:

```
public static void main( String[] args )
{
  oneRectangle();
  oneRectangle();
  oneRectangle();
}
```

Remember that the declaration of a method in a class appears at depth 1 of a source code. If a class has some *k* methods, the source code of the class will look like:

```
public class Foo
{
  ATTRIBUTES_1 METHOD_1(...)
  {
    ...
  }
  ATTRIBUTES_2 METHOD_2(...)
  {
    ...
  }
  ...
  ATTRIBUTES_k METHOD_k(...)
  {
    ...
  }
}
```

Here, `ATTRIBUTES_i` is a series of attributes for `Method_i`. These methods may appear in any order.

Here is a complete source code for printing three rectangles using the method `oneRectangle`:

```java
public class Rectangle01
{
  public static void oneRectangle() {
    System.out.println( "+---+" );
    System.out.println( "|   |" );
    System.out.println( "|   |" );
    System.out.println( "|   |" );
    System.out.println( "+---+" );
  }
  public static void main( String[] args )
  {
    oneRectangle();
    oneRectangle();
    oneRectangle();
  }
}
```

Listing 4.3 A program that produces three rectangles. An alternate version

Since the order in which the two methods, `oneRectangle` and `main`, appear in the source does not affect the way the source code works, the following code, in which the order of their appearances is reversed, behaves exactly the same:

```java
public class Rectangle01_Rev
{
  public static void main( String[] args )
  {
    oneRectangle();
    oneRectangle();
    oneRectangle();
  }
  public static void oneRectangle()
  {
    System.out.println( "+---+" );
    System.out.println( "|   |" );
    System.out.println( "|   |" );
    System.out.println( "|   |" );
    System.out.println( "+---+" );
  }
}
```

Listing 4.4 A program that produces three rectangles. The order of the two methods have been switched

Figure 4.1 shows how the two methods work together. Each method is visualized in a column, where the statements appearing in it are presented from top to bottom. When the first call of `oneRectangle` occurs, C is recorded as the return location after completion of `oneRectangle`, and then the execution of `oneRectangle` starts. When the execution of `oneRectangle` completes, the return location of C is retrieved, and from there the execution of `main` resumes (Fig. 4.2). We call the concept of using multiple methods with specific roles assigned to the methods a **procedural decomposition**. The procedural decomposition of the three-rectangle program has three benefits:

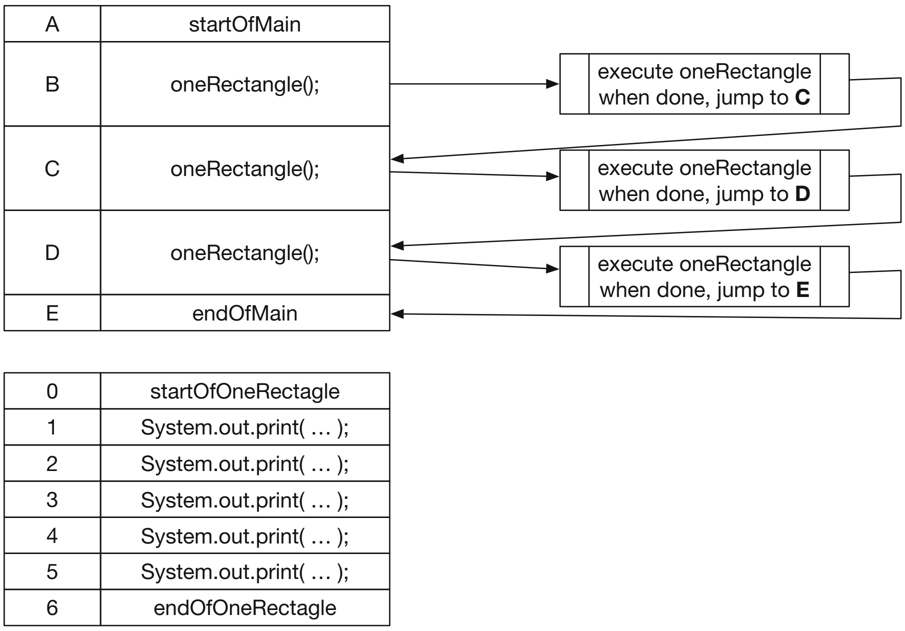

Fig. 4.1 The method calls in `Rectangle01`

1. In the source code of `main`, it is clear that some set of actions is repeated three times.
2. To change the shape, only one shape (i.e., the shape printed with `oneRectangle`) needs to be modified.
3. To change the number of times the shape is printed, only the number of calls of `oneRectangle()` needs to be modified.

Procedural decompositions improve the understanding of the code and make modifications easy. Procedural decompositions can be made in a bottom-up manner, building a new method out of already existing ones, as in case of the three identical rectangles. Procedural decompositions can be made in a top-down manner as well, dividing an existing method into smaller components, as we will see now.

We notice that there are only two distinct strips in the rectangle: `+--+` and `|--|`. We define methods, `line` and `section`, that present these strips, respectively:

```
  public static void line()
  {
    System.out.println( "+---+" );
  }
```

and

```
  public static void section()
  {
    System.out.println( "|   |" );
  }
```

We can then rewrite `oneRectangle` as:

```java
public static void oneRectangle()
{
  line();
  section();
  section();
  section();
  line();
}
```

The overall program looks like this:

```java
public class Rectangle02
{
  public static void line()
  {
    System.out.println( "+---+" );
  }
  public static void section()
  {
    System.out.println( "|   |" );
  }
  public static void oneRectangle()
  {
    line();
    section();
    section();
    section();
    line();
  }
  public static void main( String[] args )
  {
    oneRectangle();
    oneRectangle();
    oneRectangle();
  }
}
```

Listing 4.5 A program that produces three rectangles. The final version

The way the method calls are handled is now two-tiered. Added benefits of using this structure are:

1. To change the width of the rectangle, the programmer only needs to edit the `String` literals appearing in `line` and `section`.
2. To modify the number height of the rectangle, the programmer only needs to change the number of times `oneRectangle` calls `section`.

4.1.2 Printing Quadrangles

Consider drawing four rectangles of same dimensions, two on top of the other two, where neighboring rectangles share their adjacent sides and corners, as shown next:

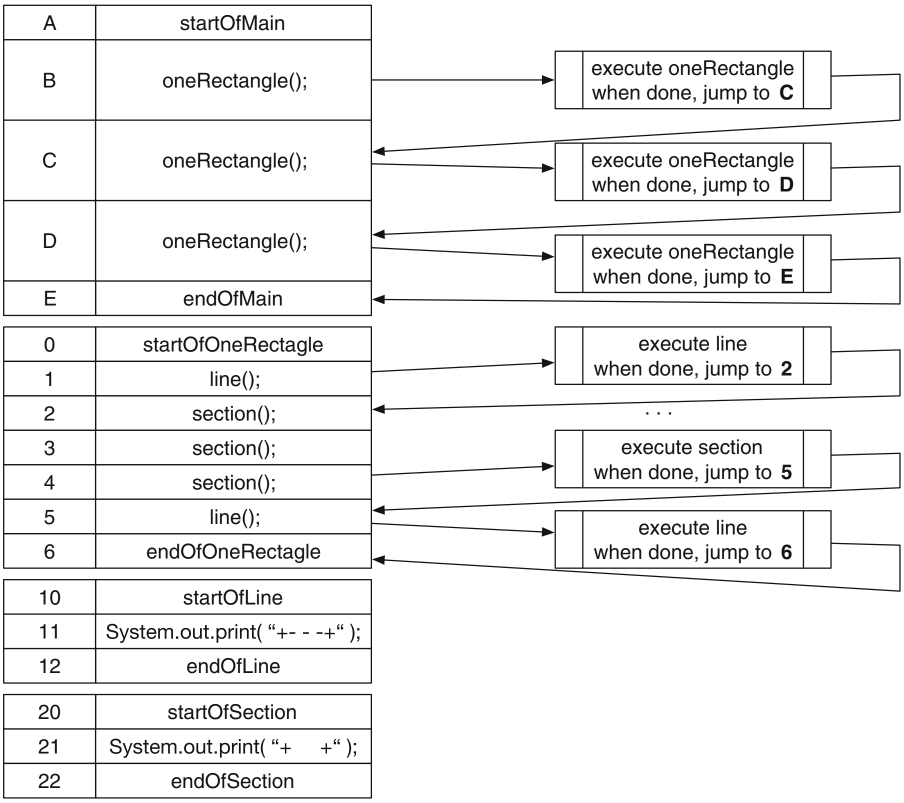
A
startOfMain
B
oneRectangle();
C
oneRectangle();
D
oneRectangle();
E
endOfMain
execute oneRectangle when done, jump to C
execute oneRectangle when done, jump to D
execute oneRectangle when done, jump to E
0
startOfOneRectagle
1
line();
2
section();
3
section();
4
section();
5
line();
6
endOfOneRectagle
execute line when done, jump to 2
. . .
execute section when done, jump to 5
execute line when done, jump to 6
10
startOfLine
11
System.out.print("+- - -+");
12
endOfLine
20
startOfSection
21
System.out.print("+   +");
22
endOfSection

Fig. 4.2 The method calls in `Rectangle02`

```
+---+---+
|   |   |
|   |   |
|   |   |
+---+---+
|   |   |
|   |   |
|   |   |
+---+---+
```

With all the actions placed in the `main` method, the code will look like:

```
public class Quadrant01
{
  public static void main( String[] args )
  {
    System.out.println( "+---+---+" ); // top line
    System.out.println( "|   |   |" ); // top section 1
    System.out.println( "|   |   |" ); // top section 2
    System.out.println( "|   |   |" ); // top section 3
    System.out.println( "+---+---+" ); // middle line
    System.out.println( "|   |   |" ); // bottom section 1
    System.out.println( "|   |   |" ); // bottom section 2
    System.out.println( "|   |   |" ); // bottom section 3
    System.out.println( "+---+---+" ); // bottom line
  }
}
```

Listing 4.6 A program that prints four rectangles with two on top of the other two

As in the case of the three rectangles, there are only two different printed strips that are printed: (a) the pattern appearing at the top, in the middle, and at the bottom of the shape and (b) the pattern appearing elsewhere. So, as before, we define two methods representing the patterns: `line` for the former and `side` for the latter. We can decompose the drawing of the shape as the following sequence:

`line`, `side`, `side`, `side`, `line`, `side`, `side`, `side`, `line`

We can group the series of `side` into one and assign the name `section` to this group. The whole sequence then becomes:

`line`, `section`, `line`, `section`, `line`

The above discussion is summarized in the diagram shown in Fig. 4.3. We then use this analysis to decompose the original source code to a new one that employs multiple methods, `Quadrant03`.

Appearing first in the program is the method `line` that prints the line corresponding to the top, the middle, and the bottom lines. Appearing next is the method `side`, which prints the line corresponding to one line of the middle section.

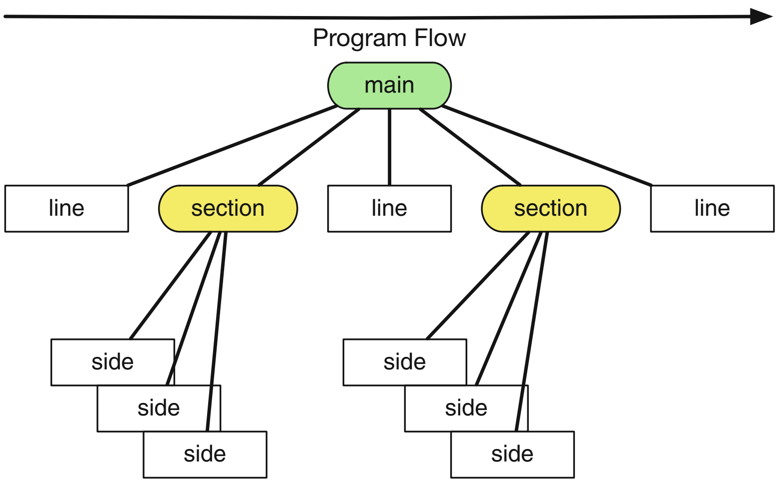

Fig. 4.3 The decomposition of actions in the generation of the quadrant

```java
public class QuadrantDecomposed
{
  // the horizontal line
  public static void line()
  {
    System.out.println( "+---+---+" ); // border line
  }
  // the side line
  public static void side()
  {
    System.out.println( "|   |   |" ); // one line of section
  }
```

Listing 4.7 A program that produces four rectangles using method class (part 1). The methods that are responsible for producing single lines

Appearing next is the method `section` that calls `side` three times. At the end, the method `main` appears and calls `line`, `section`, `line`, `section`, and `line` in this order.

```java
  // the middle block between the horizontal lines
  public static void section()
  {
    side(); // section 1
    side(); // section 2
    side(); // section 3
  }
  // the main
  public static void main( String[] args )
  {
    line();
    section(); // top section
    line();
    section(); // bottom section
    line();
  }
}
```

Listing 4.8 A new version of the quadrant generation program that uses method calls (part 2). The methods for printing the middle section and the method `main`

4.1.3 "Old MacDonald Had a Farm"

Suppose we are to write a code that produces on the screen the lyrics to a popular nursery rhyme "Old MacDonald Had A Farm". Each verse of the song introduces one new animal and then presents the sound that the animal makes as well as the sounds of all the other animals in the reverse order of introduction.

There are many variations of this rhyme, with regards to the number of animals and the order of appearance. Here is one version with four animals (a cow, a pig, a duck, and a horse) with no "repeats" from previous verses.

```
Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a cow
E-I-E-I-O
With a moo moo here
And a moo moo there
Here a moo, there a moo
Everywhere a moo moo
```

```
Old MacDonald had a farm
E-I-E-I-O

```

Listing 4.9 The lyrics to the rhyme "Old MacDonald Had A Farm" with four animals. Presented without repeats

```
Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a pig
E-I-E-I-O
With a oink oink here
And a oink oink there
Here a oink, there a oink
Everywhere a oink oink
Old MacDonald had a farm
E-I-E-I-O

Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a duck
E-I-E-I-O
With a quack quack here
And a quack quack there
Here a quack, there a quack
Everywhere a quack quack
Old MacDonald had a farm
E-I-E-I-O

Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a horse
E-I-E-I-O
With a neigh neigh here
And a neigh neigh there
Here a neigh, there a neigh
Everywhere a neigh neigh
Old MacDonald had a farm
E-I-E-I-O

```

Listing 4.10 The lyrics to the rhyme "Old MacDonald Had A Farm" with four animals. Presented without repeats (cont'd)

With the repeats from previous verses, the rhyme looks like:

```
Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a cow
E-I-E-I-O
With a moo moo here
And a moo moo there
Here a moo, there a moo
Everywhere a moo moo
Old MacDonald had a farm
E-I-E-I-O

Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a pig
E-I-E-I-O
```

Listing 4.11 The lyrics to the rhyme "Old MacDonald Had A Farm" with four animals (part 1)

```
With a oink oink here
And a oink oink there
Here a oink, there a oink
Everywhere a oink oink
With a moo moo here
And a moo moo there
Here a moo, there a moo
Everywhere a moo moo
Old MacDonald had a farm
E-I-E-I-O

Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a duck
E-I-E-I-O
With a quack quack here
And a quack quack there
Here a quack, there a quack
Everywhere a quack quack
With a oink oink here
And a oink oink there
Here a oink, there a oink
Everywhere a oink oink
With a moo moo here
And a moo moo there
Here a moo, there a moo
Everywhere a moo moo
Old MacDonald had a farm
E-I-E-I-O

Old MacDonald had a farm
E-I-E-I-O
And on his farm he had a horse
E-I-E-I-O
With a neigh neigh here
And a neigh neigh there
Here a neigh, there a neigh
Everywhere a neigh neigh
With a quack quack here
And a quack quack there
Here a quack, there a quack
Everywhere a quack quack
With a oink oink here
And a oink oink there
Here a oink, there a oink
Everywhere a oink oink
With a moo moo here
And a moo moo there
Here a moo, there a moo
Everywhere a moo moo
Old MacDonald had a farm
E-I-E-I-O

```

Listing 4.12 The lyrics to the rhyme “Old MacDonald Had A Farm” with four animals (part 2)

The structure of these verses is simple. Each verse consists of four parts:

part number	the lines	characteristic
1	“Old MacDonald ... -O”	common among all the verses
2	“And ... had a XXX ... -O”	unique to each verse
3	“With a YYY ... Everywhere a YYY”	cumulative
4	“Old MacDonald ... -O”	the first block, then one empty line

Based upon this observation, we design the following code:

```
public class OldMacDonaldDecomposed
{
  // start and end of each verse
  public static void macDonald()
  {
    System.out.println( "Old MacDonald had a farm" );
    System.out.println( "E-I-E-I-O" );
  }
  // possession of a cow
  public static void cowPossession()
  {
    System.out.println( "And on his farm he had a cow" );
    System.out.println( "E-I-E-I-O" );
  }
  // possession of a pig
  public static void pigPossession()
  {
    System.out.println( "And on his farm he had a pig" );
    System.out.println( "E-I-E-I-O" );
  }
  // possession of a duck
  public static void duckPossession()
  {
    System.out.println( "And on his farm he had a duck" );
    System.out.println( "E-I-E-I-O" );
  }
  // possession of a horse
  public static void horsePossession()
  {
    System.out.println( "And on his farm he had a horse" );
    System.out.println( "E-I-E-I-O" );
  }
```

Listing 4.13 A program that print the lyrics to “Old MacDonald Had A Farm” using decomposition (part 1). The method that produces the opening and ending lines of the verses and the methods for producing the lines about the animals

```
  // the sound of a cow
  public static void cowSound()
  {
    System.out.println( "With a moo moo here" );
    System.out.println( "And a moo moo there" );
    System.out.println( "Here a moo, there a moo" );
    System.out.println( "Everywhere a moo moo" );
  }
  // the sound of a pig
  public static void pigSound()
  {
    System.out.println( "With an oink oink here" );
    System.out.println( "And an oink oink there" );
    System.out.println( "Here an oink, there an oink" );
    System.out.println( "Everywhere an oink oink" );
  }
  // the sound of a duck
  public static void duckSound()
  {
    System.out.println( "With a quack quack here" );
    System.out.println( "And a quack quack there" );
    System.out.println( "Here a quack, there a quack" );
    System.out.println( "Everywhere a quack quack" );
  }
  // the sound of a horse
  public static void horseSound()
  {
    System.out.println( "With a neigh neigh here" );
    System.out.println( "And a neigh neigh there" );
    System.out.println( "Here a neigh, there a neigh" );
    System.out.println( "Everywhere a neigh neigh" );
  }
```

Listing 4.14 A program that print the lyrics to "Old MacDonald Had A Farm" using decomposition (part 2). The methods for printing the lines that introduce the sounds that the animals make

```
  // the cow verse
  public static void cowVerse()
  {
    macDonald();
    cowPossession();
    cowSound();
    macDonald();
  }
  // the pig verse
  public static void pigVerse()
  {
    macDonald();
    pigPossession();
    pigSound();
    cowSound();
    macDonald();
  }
  // the duck verse
  public static void duckVerse()
  {
    macDonald();
    duckPossession();
    duckSound();
    pigSound();
    cowSound();
    macDonald();
  }
  // the horse verse
  public static void horseVerse()
  {
    macDonald();
    horsePossession();
    horseSound();
    duckSound();
    pigSound();
    cowSound();
    macDonald();
  }
  // main
  public static void main( String[] args )
  {
    cowVerse();
    System.out.println();
    pigVerse();
    System.out.println();
    duckVerse();
    System.out.println();
    horseVerse();
  }
}
```

Listing 4.15 A program that produces the lyrics to the rhyme “Old MacDonald Had A Farm” (part 3). The methods that build individual verses and the method `main`

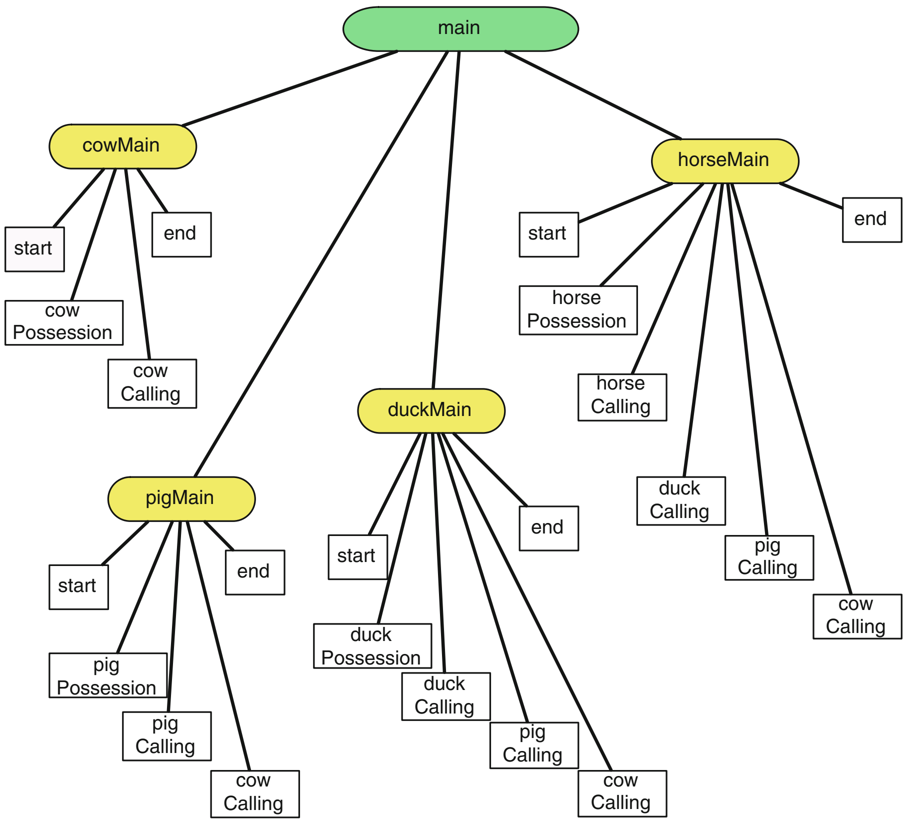

Fig. 4.4 The dependency among methods in `OldMacDonaldDecomposed.java`

Figure 4.4 presents the dependencies among the methods.

4.1.4 A General Strategy for Procedural Decomposition

While procedural decomposition helps better understand the code and makes future revisions easier, it does not necessarily reduce the length of the source code because each additional method has its own header and encompassing curly brackets.

In the next chapter, Chap. 5, we will study methods that take parameters and/or return a value. By combining procedural decomposition and the use of multiple source code files, we will be able to write a program divided into reasonably small units that are all easy to understand.

Since each method appearing in a source code can call each method appearing in the same source code, it is possible to create a circle of method calls.

Suppose we have the program:

```java
public class Parts123
{
  public static void partOne()
  {
    System.out.println( "One" );
    partTwo();
  }
  public static void partTwo()
  {
    System.out.println( "Two" );
    partThree();
  }
  public static void partThree()
  {
    System.out.println( "Three" );
  }
  public static void main( String[] args )
  {
    partOne();
  }
}
```

Listing 4.16 A program with methods that print 1, 2, and 3

The action of the entire code is simple: `main` calls `partOne`, `partOne` calls `partTwo`, and `partTwo` calls `partThree`. This produces the output of `"One"`, `"Two"`, and `"Three"`. The execution of the program produces the output:

```
One
Two
Three
```

By making a slight change to the code we can produce a bizarre effect.

```java
public class InfiniteCalls
{
  public static void partOne()
  {
    System.out.println( "One" );
    partTwo();
  }
  public static void partTwo()
  {
    System.out.println( "Two" );
    partThree();
  }
```

Listing 4.17 A program that produces a bizarre effect (part 1)

```
  public static void partThree()
  {
    System.out.println( "Three" );
    partOne();
  }
  public static void main( String[] args )
  {
    partOne();
  }
}
```

Listing 4.18 A program that produces a bizarre effect (part 2)

The code produces the following:

```
% javac InfiniteCalls.java
% java InfiniteCalls
One
Two
Three
...
...
Exception in thread "main" java.lang.StackOverflowError
  at sun.nio.cs.UTF_8$Encoder.encodeLoop(UTF_8.java:691)
  at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
  at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
  at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
  at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
  at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:129)
  at java.io.PrintStream.write(PrintStream.java:526)
  at java.io.PrintStream.print(PrintStream.java:669)
  at java.io.PrintStream.println(PrintStream.java:806)
  at InfiniteCalls.partOne(InfiniteCalls.java:3)
  at InfiniteCalls.partThree(InfiniteCalls.java:12)
  at InfiniteCalls.partTwo(InfiniteCalls.java:8)
  at InfiniteCalls.partOne(InfiniteCalls.java:4)
...
  at InfiniteCalls.partThree(InfiniteCalls.java:12)
  at InfiniteCalls.partTwo(InfiniteCalls.java:8)
  at InfiniteCalls.partOne(InfiniteCalls.java:4)
%
```

The actual output of the code is much longer. The `...` signifies the visual cut made to the output. The import thing to notice is the line

`Exception in thread "main" java.lang.StackOverflowError`

The error message states that the method calls have used up all the memory space available for JVM to run, so JVM had to abort the execution of the code. The direct cause of this termination is due to the method call structure from Fig. 4.5. You can see that there is a loop going from one to three. This loop repeats over and over again, which results in the exhaustion of the memory space. In general, we call a loop structure that makes the program run forever an **infinite loop**. Thus, we say that the code `InfiniteClass.java` has an infinite loop.

The word "Exception" appearing in the error message is a word that refers to a **run-time error**.

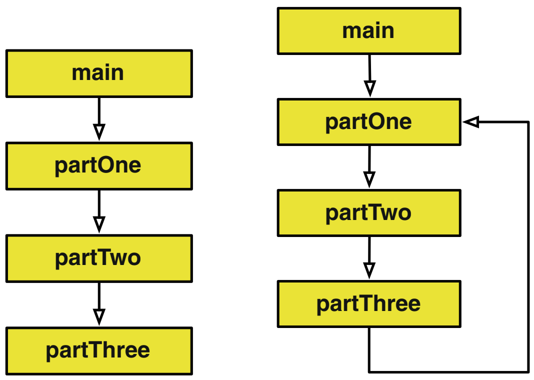

Fig. 4.5 The dependency among methods in the two source code. Left panel: the original code. Right panel: the modified code

4.2 Using Multiple Program Files

We extend the idea of extracting components from a method to create another method, and write multiple classes and use methods from one in another Java class.

The benefits of reusing existing source codes are twofold. First, the use of recycling code from another class saves the coder from having to write the same code from scratch again. Second, sharing the code among applications may make it easier to revise the code.

Consider the following class `Signature`.

```java
public class Signature
{
  public static void sign()
  {
    System.out.println();
    System.out.println(
      "+-----------------------------------------+" );
    System.out.println(
      "| THIS PROGRAM IS CODED BY MITSU OGIHARA  |" );
    System.out.println(
      "+-----------------------------------------+" );
  }
}
```

Listing 4.19 A program that produces a signature

`Signature.java` has one method, `sign`, which produces four lines of output as follows:

```

+-----------------------------------------+
| THIS PROGRAM IS CODED BY MITSU OGIHARA  |
+-----------------------------------------+
```

There is no `main` method in this class, so we cannot execute the code by itself. By attaching the class name `Signature` and a period before the method name, we can call this method from outside:

`Signature.sign();`

Knowing this capability we can write a new version of `HelloWorld`, which produces the signature lines along with the `"Hello, World!"` message.

```java
public class HelloWorldCall
{
  public static void main( String[] args )
  {
    System.out.println( "Hello, World!" );
    Signature.sign();
  }
}
```

Listing 4.20 A `HelloWorld` program that print a signature at the end

To run the above code, you need both `HelloWorldCall` and `Signature`. Since the former used the latter, one can compile the latter first and then compile the former. Alternatively, we may simply compile both at the same time:

```
javac Signature.java HelloWorldCall.java
```

Summary

- A class can have any number of methods in it.
- Methods are defined at depth 1 of the class in which they appear.
- Methods can appear in any order in a class.
- The process of creating a method that is in charge of performing a certain part of the actions another method performs is called procedural decomposition.
- The benefits of procedural decomposition include better readability and easier code modification.
- It is possible to write multiple classes with methods making calls across classes. When using multiple program files, the files can be compiled either individually or all at once.

Exercises

1. **Number manipulation** Suppose we are writing a program `PlayWithNumbersDecomposed`, in which we have two tasks:
   (a) The program receive two integers, `a` and `b`, from the user and then prints `a + b`, `a - b`, `a * b`, `a / b`, and `a % b` (we anticipate that the user will not enter 0 for the second number).
   (b) The program receives three integers, `a`, `b`, and `c`, from the user and then prints the result of `(a - b) / c` for each of the six possible permutations among the three numbers.
   Write the code for this program so that it has two separate methods that handle the two tasks. The method `main` calls the two methods one after the other. Each non-main method instantiates its own `Scanner` object with `System.in`. Here is an example of how the program may interact with the user.

```
Enter two integers: 1000435 345
a + b is equal to 1000780
a - b is equal to 1000090
a * b is equal to 345150075
a / b is equal to 2899
a % b is equal to 280
Enter three integers: 34325 79 -40
(a - b)/c is equal to -856
(a - c)/b is equal to 435
(b - c)/a is equal to 0
(b - a)/c is equal to 856
(c - a)/b is equal to -435
(c - b)/a is equal to 0
```

2. **Shape Presentation** Suppose we are writing a program `HouseShape` that produces the following output on the screen:

```
     /\
    /  \
   /    \
  / +--+ \
 /  |  |  \
/   +--+   \
-+--------+-
 |        |
 |  +--+  |
 |  |  |  |
 |  +--+  |
 |        |
-+--------+-
This is my house!
```

The action of the program can be divided into three parts:
- printing the roof (including the bottom of the roof),
- printing the body of the house, and
- printing the message.

Write a program with three methods (in addition to `main`) which correspond to the above three tasks, where the method `main` simply calls the three methods in order.

3. **Forward slashes** Previously we wrote a program that produced on the screen:

```
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
/ / / / / / / / / / / /
 / / / / / / / / / / /
```

This output consists of five repetitions of the pattern of the first two lines. Write an alternate version, `SlashesWithMethodCalls`, that performs this task with five identical method calls (in the method `main`) to a method, `twoLines`. The method `twoLines` produces the consecutive pair of two lines.

Programming Projects

4. **This Old Man** "This Old Man" is a popular children rhyme that consists of ten verses. All the verses are identical, except for the two words, which we present below as `XXX` and `YYY`, where `XXX` goes from `one` to `ten`

```
This old man, he played XXX,
He played knick-knack on his YYY;
With a knick-knack paddywhack,
Give the dog a bone,
This old man came rolling home
```

Here is the list of ten words for `YYY`:

`drum`, `shoe`, `knee`, `door`, `hive`, `sticks`, `heaven`, `gate`, `spine`, `again`

We can decompose the common parts of the verses into:
(a) The first line excluding `XXX,`. In other words, it is `"This old man, he played "`
(b) The second line excluding `YYY;`. In other words, it is `"He played knick-knack on his "`
(c) The third to fifth lines plus one empty line.
Let `first`, `second`, and `rest` be methods that print the three parts, where `first` and `second` use `System.out.print` and `rest` uses `System.out.println`. Using this decomposition, write a program that prints the first three verses of the rhyme.

5. **Die face printing** Consider printing the six sides of a die using in a 5 × 5 grid as follows:

```
+---+
|   |
| o |
|   |
+---+

+---+
|   |
|o o|
|   |
+---+

+---+
|  o|
| o |
|o  |
+---+

+---+
|o o|
|   |
|o o|
+---+
+---+
|o o|
| o |
|o o|
+---+

+---+
|o o|
|o o|
|o o|
+---+
```

Write a program named `Dice` that prints the six faces using a method call to print each line of the faces.

6. **Digits in** 5×5 **rectangles** Consider writing a program that prints digits $0, \ldots, 9$ in 5×5 rectangles using the following design: Here is the code:

```
+----+      |  ----+  ----+ |    | +----  +----   ----+ +----+ +----+
|    |      |      |      | |    | |      |           | |    | |    |
|    |      | +----+  ----+ +----+ +----+ +----+      | +----+ +----+
|    |      | |           |      |      | |    |      | |    |      |
+----+      | +----   ----+      |  ----+ +----+      | +----+      |
```

To save space, the digits are placed side by side, but in the actual code, the digits will be stuck on top of each other. There are only six patterns appearing in the digits:

```
+----+
+----
 ----+
|    |
|
     |
```

We can give these six patterns unique pattern names and write six methods that print the individual six patterns. With the addition of one more method that prints one empty line, the seven methods can be used as the building blocks for digit printing. Write a program named `Digits` that accomplishes this task. The program should have the seven building blocks as methods, ten methods that print the ten digits with one blank line as their sixth lines, and the method `main` that calls the ten methods one after another.

7. **A Pop song** Select one of the No. 1 hits by *The Beatles* (e.g., "I Want To Hold Your Hand") and write a program that prints the lyrics to the song line by line. A popular song lyrics search engine can be used to find the lyrics. Consider the following points:
   - If a line is repeated more than once, define a method for printing that line alone.
   - If a series of lines are repeated more than once, find a maximally long stretch for that series and define a method for printing that series.
   - It is natural to define a method for each verse or bridge (a verse and a bridge are the units that are presented with no blank lines in them).