Software Development (CS2500)

Lecture 15: Writing Methods

M.R.C. van Dongen

November 5, 2010

Contents

1 Introduction **1**

2 Methods **2**
2.1 Kinds of Methods 2
2.2 Side Effects 2
2.3 Managing Computations 3
2.4 Entry and Exit Points 4
2.5 Aim for Simplicity 5
2.6 Developing Methods 5

3 Case Study **6**
3.1 Requirements 6
3.2 High-Level Design 7
3.3 Refining Pseudo Code 8
3.4 Reflection 10

4 For Monday **10**

1 Introduction

These notes study two things. The first topic is good practice of method writing. The second topic is designing methods. By the end of these notes you should know how to write methods and develop algorithms using methods.

2 Methods

This section studies methods and method development. It start by recalling the difference between `void` methods, which don't return a value, and non-`void` methods, which do return a value. It continues by studying side effects of methods, managing computations with methods, writing methods, and designing methods.

2.1 Kinds of Methods

There are two kinds of methods:

`void` methods: These methods don't have a `return` statement, don't `return` values, and have the `void` "return type". The main purpose of these methods is to manage other computations.

Non-`void` methods: These methods do have a `return` statement, which they use to explicitly `return` a `value`. Usually, this is the only purpose.

2.2 Side Effects

In reality, the distinction between methods that "return" and don't "return" values is not always so clear. The reason for this is that methods may have *side effects* which change the state of the computation. For example:

- A class method may have access to class variables and it may change the value of such variables during a method call. The following is an example. The method `println` is a `void` method. As such it does not explicitly `return` a value. However, the value of the class attribute `printlnCalls`[1] is incremented as a result of each call.

```java
public class MySystem {
    private static int printlnCalls;

    public static void println( String str ) {
        printlnCalls ++;
        System.out.println( str );
    }

    public static int getPrintlnCalls( ) {
        return printlnCalls;
    }
}
```

- An instance method may have access to both class and instance variables. These methods may change the value of any such variable as a result of a method call.

[1] Why is this a *class* attribute?

- Even if you can't "see" state affecting changes as explicit assignments inside a method, this does not mean they are not there. For example, side effects may occur as the result of submethod calls inside the method.

2.3 Managing Computations

The main purpose of a method is to organise and manage its computation. The statements in the body of the method should have a natural order: first do this, then that, and so on. Each sub-computation should be well-defined. This includes the input, output, and purpose of the computation.

The purpose of previous sub-computations it to prepare the explicit/implicit input for subsequent sub-computations.

You should always aim for the maximum possible clarity in your computations. For example, your code is clearer if you don't use the `return` statement in `void` methods, and only have one `return` statement in non-`void` methods.

Arguably, the following is not clear because there are several exit points from the method. In general this makes it difficult to reason about the method. As we shall see in a few moments, the `return` statement in the middle of the method is not needed and omitting it simplifies the method.

Don't Try this at Home

```
public static void print( boolean condition, String str ) {
    if (!condition) {
        return;
    }
    System.out.println( str );
}
```

The following method is equivalent, but much clearer.

Java

```
public static void print( boolean condition, String str ) {
    if (condition) {
        System.out.println( str );
    }
}
```

The following method definition has two `return` statements. (Still this definition is crisp-and-clear.)

Java

```
public static int fib( int n ) {
    if (n <= 1) {
        return 1;
    } else {
        return fib( n - 1 ) + fib( n - 2 );
    }
}
```

The following method definition is equivalent to the previous definition. However, this definition uses only one `return` statement.

```Java
public static int fib( int n ) {
    final int result;

    if (n <= 1) {
        result = 1;
    } else {
        result = fib( n - 1 ) + fib( n - 2 );
    }

    return result;
}
```

2.4 Entry and Exit Points

The previous subsection stated that it is good practice to have one `return` statement in a method. This is a consequence of the following general rule:

> Each computation should have (exactly) one entry point and one exit point.

This rule can be enforced by avoiding the `break` statement.[2] The following is an example of poor programming style.

```Don't Try this at Home
while (condition1) {
    if (condition2) {
        break;
    } else {
        ⟨stuff⟩
    }
}
```

The following shows how to avoid this poor programming style. The result is much cleaner code, which is easier to understand and maintain.

```Java
while ((condition1) && (!condition2)) {
    ⟨stuff⟩
}
```

For the remainder of this course you are not allowed to use the `break` statement to break out of loops.

[2] Except for `breaks` in `switch` statement.

2.5 Aim for Simplicity

Some programmers are always on the lookout for opportunities to combine sub-computations. Usually, this increases the complexity of their algorithms, making it more difficult to reason about the code. What is worse, the code becomes more difficult to maintain. In general, you should aim at simplicity. By adopting this rule, the quality of your methods will improve. In addition it makes your methods easier to write and maintain.

2.6 Developing Methods

Method definitions should be short and concise. Sub-computations in method definitions may be written using short, simple statements provided these statements don't require too many lines. No method should be longer than approximately 40 lines. If a method requires too many lines, then implement it as a series of submethod calls. Likewise, if a sub-computation is "long" then you should consider writing it as a submethod call.[3] Implementing the sub-computation as a method call simplifies the overall computation. For example, you can see exactly what goes into the sub-computation and comes out of it. This streamlines your code and makes it easier to maintain.

Your methods should be well-defined: input, output, and task. It is impossible to develop a complex algorithm without trial and error. Despite this observations, it is still possible to stay in control of the design process if you develop your methods in a top-down fashion. With a top-down design you:

- Start with a coarse version of the algorithm.
- You implement the algorithm as a method.
- Initially, you implement the basic steps in pseudo-code.
- When you're happy with the design of the method you substitute method calls and basic `Java` for the pseudo-code.
- You recursively *refine* your methods until you end up with "basic" `Java`.

The following spells out the top-down design process in more detail.

- If the task requires a few simple statements:
  - Write it down using simple statements.
- If the task requires many statements or is not easy to formulate:
  - Think of a sequence of sub-computations that carry out the overall computation.
  - Initially you state sub-computations using *pseudo-code*. Here *pseudo-code* is a mixture of English, mathematics, and `Java`.
  - Each pseudo-code statement should be well defined.

[3] Here the computation is long if it is difficult to tell the subcomputation's input and output.

  - This includes the main task, input, and output.
  - Easy pseudo-computations can be implemented "directly":
    * Either you use existing methods/classes/libraries.
    * Or you write simple statements without method calls.

    Possible candidates for easy tasks are the computations that are already expressed in `Java`.
  - Complicated pseudo-computations should be implemented using calls to new submethods.
  - By developing the new submethods in a similar way you can write your entire method.

3 Case Study

In this section we shall carry out a case study in method and class writing. In particular we shall implement a program that plays "guessing games".

3.1 Requirements

The following are the requirements of our guessing game.

- There are two contestants: John and Paul.
- John and Paul compete until there's a winner of the match.
- The program announces the winner and the final score.
- The match is won by the first player who wins 3 sets. Initially, each player starts with 0 sets won.
- The rules for playing a set are as follows:
  - Each player starts with 0 games won.
  - The players play a sequence of games.
  - The first player who wins 6 games wins the set.
- The rules for playing a game are as follows:
  - Each game has its own referee.
  - The game consists of a sequence of rounds.
  - The game is won by the first player that wins a round without ties.
  - Each round the referee and players guess a random `boolean`.
  - A tie occurs if both players guess the same `boolean`.
  - Otherwise, the winner is the player that guesses the same `boolean` as the referee.

3.2 High-Level Design

In the following we start by assuming that we only need a `Contestant` class. (This is reasonable since the contestants appear to be the actors. Furthermore, we can always add classes as needed.)

The following is a possible high-level specification.

Pseudo Code

```
Contestant john = new Contestant( "John" );
Contestant paul = new Contestant( "Paul" );

// Determine winner and loser of match.

// Announce winner and score.
```

It seems natural to introduce a method `playMatch( )` that plays a match and returns the winner. The implementation of "Announce winner and score" seems trivial. Of course this depends on how we implement `Contestant` and `playMatch( )`.

There are two obvious options for a signature for `playMatch`:

- The first option is '`Contestant playMatch( Contestant player1, Contestant player2 )`'. This method plays the match using `player1` and `player2`. Since it does not depend on other `Contestants` it is best implemented as a class method: `static Contestant playMatch( Contestant player1, Contestant player2 )`.
- The second option is '`Contestant playMatch( Contestant that )`'. Here the idea is that this method plays the match between `this` and `that`. This method depends on the implicit variable `this`, which corresponds to the "current" object: the object that called the method as an instance method. Therefore, this method has to be implemented as an instance method.

Let's opt for the second choice.

Pseudo Code

```
// Play match between this and that and return winner.
public Contestant playMatch( Contestant that ) {
    boolean matchOver = false;
    // Initialise numbers of sets won.
    while (!matchOver) {
        // Determine setWinner: the winner of the next set.
        // Increase the number of sets won of setWinner.
        // Set matchOver to true if setWinner has won the match.
    }
    return (this.winsMatch( ) ? this : that);
}
```

3.3 Refining the High-Level Pseudo Code

Our high-level pseudo-code formulation has the following tasks:

- Initialise *numbers of sets won*.
- Determine `setWinner`: the winner of the next set.
- Increase the *number of sets won* of `setWinner`.
- Set `matchOver` to true if `setWinner` has won the match.

It seems natural to introduce an instance variable `int setsWon` that records the number of sets won. It also looks the we're going to need a method which determines whether a given `Contestant` has won the match.

- Instance attribute: `int setsWon`. This attribute counts the number of sets won by `this Contestant`.
- Instance method: `boolean winsMatch( )`. This method returns `true` if and only if `this Contestant` has won the match.

Given our choice for `setsWon` and `winsMatch( )` we should be able to find an implementation according to the following lines.

```Java
public class Contestant {
    /**
     * Number of sets a {@code Contestant} needs to win
     * in order to win match.
     */
    private static int SETS_REQUIRED_FOR_MATCH = 3;
    /**
     * Number of times {@code Contestant} has won a set.
     */
    private int setsWon;

    /**
     * Determine if {@code Contestant} has won the match.
     *
     * @return {@code true} iff {@code this Contestant} has won the match.
     */
    public boolean winsMatch( ) {
        return setsWon == SETS_REQUIRED_FOR_MATCH;
    }
}
```

Notice that the choice for the visibility modifiers `private` and `public` is automatic:

- Instance and class variables should be encapsulated so they should be `private`.
- For the design to work it should be possible to call the method `winsMatch( )` outside the `Contestant` class. Therefore, the method should be `public`.

Implementing the getter and setter method for the attribute `setsWon` is left as an exercise for the reader.

It is also clear that contestants have a name. It seems reasonable to introduce an instance attribute `name` that represents the name and a constructor that initialises the name. While we're at it, we might as well override `toString`.

```Java
/**
 * Name of contestant.
 */
private final String name;

/**
 * Main constructor.
 *
 * @param name The name of the {@code contestant}.
 */
public Contestant( String name ) {
    this.name = name;
}

@Override
public String toString( ) {
    return name;
}
```

Let's see how far we've got. We were in the process of implementing the following:

- Initialise *numbers of sets won*.
  - This pseudo-code statement may be implemented by using the setter for the instance variable `setsWon`: `this.setSetsWon( 0 )` and `that.setSetsWon( 0 )`.
- Determine `setWinner`: the winner of the next set.
  - Playing a set and and a match is similar, so let's implement this pseudo-code statement along the same lines as our implementation of `playMatch`. For the moment we use a stub for this sub-computation: an instance method `Contestant playSet( Contestant that )`. The sole task of the method is to determine (return) the winner of the next set between `this` and `that`. Using the new method we can implement the pseudo code statement as '`Contestant setWinner = playSet( that )`'. For the moment we won't bother about the exact implementation of `playSet( )`. We know that stepwise refinement should allow us to complete the implementation of the method it later on.

- Increase the *number of sets won* of `setWinner`.
  - This pseudo-code statement may be implemented using the setter and getter of the method `setsWon`. For example, '`setWinner.setSetsWon( setWinner.getSetsWon( ) + 1 )`' does the trick.
- Set `matchOver` to true if `setWinner` has won the match.
  - This pseudo-code statement may be implemented as follows: `matchOver = setWinner.winsMatch( )`.

We've made great progress. All that remains doing is implement the method `playSet` and we're done.

3.4 A Moment of Reflection

How far have we got? We started with a pseudo-code implementation of the method `playMatch( )`. Using stepwise refinement we managed to substitute real `Java` for the pseudo-code. The result of this actually looks and feels as real `Java`:

```java
public Contestant playMatch( Contestant that ) {
    boolean matchOver = false;
    this.setSetsWon( 0 );
    that.setSetsWon( 0 );

    while (!matchOver) {
        Contestant setWinner = playSet( that );
        setWinner.setSetsWon( setWinner.getSetsWon( ) + 1 );
        System.out.println( "Set won by " + setWinner );
        matchOver = setWinner.winsMatch( );
    }

    return (this.winsMatch( ) ? this : that);
}
```

If we try to compile this, then the compiler will tell us that there's only one thing that's missing: the method `playSet( )`. This method can be developed in a similar way as we implemented the method `playMatch( )`.

4 For Monday

Study the notes and complete the design of the guessing game.