Software Development (CS2500)

Lecture 20: Writing a Game

M.R.C. van Dongen

November 17, 2010

Contents

1 Introduction

This lecture we shall study a search algorithm and implement a game. The search algorithm is linear search. The game is a battleship-like game. However, we shall not implement the full-blown version. Instead, we shall implement a simplified version. Using the specifications as our input, we shall

- write prep code,
- write test code, and
- write real code.

On Monday we shall study `JUnit` testing, which should let us write proper test code. On Wednesday we shall implement the full-blown version of the game. The search algorithm is not covered by the book. The game is covered by Chapter 5 in the book.

2 Linear Search

Despite its simplicity, the *linear search algorithm* is one of the fundamental algorithms in computer science. Many efficient algorithms depend on linear search.

Basically, the linear search algorithm is a left-to-right search algorithm that traverses a sequence of items so as to locate an item in the sequence that satisfies a certain property. If the algorithm fails then all member in the sequence lack the property. Otherwise the algorithm locates the "first" item that satisfies the property.

For arrays, the algorithm is particularly pleasant to formulate. The termination and correctness proofs are also pleasantly easy.

The remainder of this section studies a simplified form of the linear search algorithm. For simplicity all arrays contain `int` values and a single property which is given by a method `boolean found( int value )`. In a future lecture we shall generalise the algorithm.

The following is the basic form of linear search for `int` arrays. The purpose of the algorithm is to find the left-most number in the array that satisfies our given property or decide that there is no number in the array that satisfies the property. The static method `boolean found( int number )` is `true` if `number` satisfies the property and `false` otherwise.

```java
int index = 0;
// index <= array.length and
// !found( array[ prev ] ) for 0 <= prev < index
while (index < array.length && !found( array[ index ] )) {
    index ++;
    // index <= array.length and
    // !found( array[ prev ] ) for 0 <= prev < index.
}
// index <= array.length and
// (!found( array[ prev ] ) for 0 <= prev < index) and
// (index >= array.length || found( array[ index ] ))
```

The reason why the implementation is presented using a `while` loop is that this makes it easier to state the proof.

Notice that if `found( )` always returns `false` then the algorithm enumerates all possible indices in the array and terminates with the value `arrays.length` as the final value for `index`. If `found( )` returns `true` for some index value then the algorithm has located an item that satisfies the property and the algorithm also terminates. These observations form an informal version of a termination and correctness proof. In the remainder of this section we shall provide more formal proofs.

As explained in Lecture 4 the condition after the `while` loop is constructed using the invariants before the loop, the invariants at the end of the body of the loop, and the condition of the `while` loop. The

condition of the loop is of the form 'A `&& !`B'. Negating it gives us '`(!`A`) ||` B'.

Let's first look at the termination proof.

Theorem 1. *The linear search algorithm terminates.*

Proof. Before the `while` loop and just before the end of the body of the `while` loop we have '`index <= arrays.length`'. Together with the condition of the `while` loop this guarantees that `index` cannot exceed `array.length`, which is non-negative. Initially, `index` is equal to `0` and each iteration increments `index`. Therefore, the maximum number of iterations is equal to `arrays.length`. □

The correctness proof is equally difficult.

Theorem 2. *The linear search algorithm is correct.*

Proof. As noticed before the condition after the `while` loop must hold. The termination proof guarantees that the algorithm will eventually reach the point where this condition holds.

The condition after the `while` loop is of the form 'E `&&` F `&&` G' and we've proved that it must hold. Therefore E, F, and G are all `true`. In particular this means that E — it is the condition '`index <= array.length`' — must be `true`. All we need to do is consider two cases which cover all possibilities that arise when `index <= array.length` is `true`.

Complete Failure: Assume that '`index == array.length`' holds. If this case holds then all calls to `found( )` failed for all values of `index` in `0`, `1`, ..., `array.length - 1`. We can prove this formally by substituting `array.length` for `index` in '`!found( array[ prev ] ) for 0 <= prev < index`'.

(Partial) Success: If '`index < array.length`' holds, then '`index >= array.length || found( array[ index ] )`' is equivalent to '`found( array[ index ] )`'. □

3 Specifications

This section presents the specifications of the game. The specifications are as follows. We have to implement a Battleship-style game called Sink-a-dot-Com. The game is played on 7×7 grid. The purpose of the game is to sink "dot.coms" instead of ships. Initially there are three dot.coms. Each dot.com occupies three cells on the grid. The program randomly places the dot.coms on the grid. While there are dot.coms left:

1. The program prompts the user to guess a cell.
2. The program reads in the user's guess.
3. The program checks the cell against the dot.com positions.
4. Finally, the program takes an appropriate action:
   - If the guess is a kill then the dot.com is deleted.
   - If the guess is a hit then the cell is deleted.

- Otherwise, the program reports a miss.

Following the book, we start by implementing a simplified version of the game. Later we'll scale this version to the full-blown game.

In the simplified version:

- We have only one dot.com.
- We represent it as a 3-valued `int` array.
- The values in the array are location cell numbers of the cells occupied by the dot.com.
- The location cells are consecutive numbers between 1 and 7.
- Rather that guessing cells, the user now guesses location cells.
- If the user guesses right we announce a hit.
- If there are three hits the game ends.
- Otherwise we continue.

4 Class Development

This time, we'll use the following methodology to writing our `SimpleDotCom` class.

1. Figure out what the class is supposed to do.
2. List the instance variables and methods.
3. Write *prep code* (also known as pseudo code) for the methods.
4. Write *test code* for the methods. Yes, you're reading this right. We're writing the test code before we implement the class. There are several advantages.
    - Thinking about the testing of the methods helps clarify what the methods themselves need to do.
    - Using the methods in the test code gives us a chance to play with the method API (Application Programming Interface). If there's something wrong with the API or if it's not easy to use, then we can adjust the API *before* we implement the methods themselves.
    - The test code acts as documentation/contract: it states the expected behaviour of the methods.
    - By writing test code early, we can use it straight away. It help prevent scenarios where you first write the entire class, have no time to test it, and end up with a poor/erroneous implementation.

5. Write *real code* for the methods: write the class.

6. Debug and reimplement as required.

Our first task is to figure out what the class is supposed to do. Sometimes playing a possible scenario helps clearing this up. Let's see:

1. **Game starts**: we start by creating a random `DotCom`. We continue by generating three random consecutive cell locations. For this example let's assume the cells are as follows: | 1 | 2 | 3 |.

2. **Game play begins**: The user starts guessing. This may lead to the following interaction.

```
$ java SimpleDotComGame
Enter a number: 2
hit
Enter a number: 3
hit
Enter a number: 4
miss
Enter a number: 1
kill
```
Unix Session

3. **Game finishes**: This may look like the following.

```
You took 4 guesses
```
Unix Session

Having figured out what the class is supposed to do, we continue with our second task, which is listing the instance variables and methods for our class. Following the book, we decide to opt for the attributes and methods that are listed in Figure 1.

SimpleDotCom
`int[] locationCells` `int hits`
`String checkYourself( String guess )` `void setLocationCells( int[] loc )`

Figure 1: Attributes and methods for `SimpleDotCom` game.

`locationCells:` This attribute stores the location cell numbers.

`hits:` This attribute counts the number of hits.

`checkYourself:` This method checks the user's guess and returns the program's answer.

`setLocationCells:` This method randomly initialises `locationCells`.

5 Prep Code

We've just decided which instance variables are needed in our `SimpleDotCom` class. The next thing on our list is to write *prep code* for our methods. Here *prep code* is the book's name for pseudo code.

In this section we shall provide a possible implementation for the methods. Note that this implementation is deliberately different from the implementation from the book. The main reason is that the book uses the `break` statement, which we're not allowed to use. To keep up with the book, the following implementation contains a deliberate error. If you spot the error, well done, but keep it to yourself. Otherwise, don't worry, we'll squash it in the next two lectures.

The following is a possible implementation for `checkYourself`.

Pseudo Code
```
public boolean checkYourself( String guess ) {
    int cell = ⟨convert guess to int⟩;
    boolean found = ⟨find cell in locationCells⟩;
    ⟨increment hits if found⟩;
    return ⟨use found and hits and return result as String⟩;
}
```

The implementation of `setLocationCells` is equally straightforward. Note that we implement it as a `private` method because it is not and should not be used outside the `SimpleDotCom` class. By making the `public` we unnecessarily violates encapsulation, which should be avoided.

Pseudo Code
```
private void setLocationCells( ) {
    int cell = ⟨generate first cell number⟩;
    ⟨set locationCells to {cell, cell+1, cell+2}⟩;
}
```

6 Test Code

In this section we are supposed to write test code to test our methods. For the moment we postpone this because we're going to do this using the `JUnit` framework, which we haven't studied yet.

7 Real Code

Having written our prep code, and pretending we've written our test code, we're ready to turn our prep code and turn it into real code.

First let's have a look at the prep code for `checkYourself`. We have four well-defined and easy tasks. Using the strategy which we studied in Lecture 15, we'll turn the easy pseudocode into `Java` and the less straightforward pseudocode into method calls.

- The first pseudo statement is given by '`int cell = ⟨convert guess to int⟩`'. Using the method `Integer.parseInt( )`, which we've already used for Assignment 1, this part is easy.

  Java
  ```
  int cell = Integer.parseInt( guess );
  ```

- The second statement is '`boolean found =` ⟨`find cell in locationCells`⟩'. It is not clear how to do this, so let's turn this into a method call. If we implement the method as an instance method and pass the value of `cell`, then we should be able to implement this method. After all, the instance method can see the instance variable `locationCells`. (Of course, we could have also opted for a class method that takes both `cell` and `locationCells` as argument.) All we now need to do is choose a good name for the method — remember it has to be a verb. Let's choose `findLocation( )`.

```
boolean found = findLocation( cell );
```
Java

- This is going well. Next task: '⟨`increment hits if found`⟩'. This is trivial.

```
hits += (found ? 1 : 0);
```
Java

- Our final task at this level consists of turning '⟨`use found and hits and return result as String`⟩' into something meaningful. The task does not seem too difficult, and we could probably implement it using a few statements. Still we're going to implement it as a method call. Let's do this as follows:

```
return getResultAsString( found );
```
Java

Notice that arguably we should have turned `cell` and `found` into `final` variables.

Our method `checkYourself` is looking pretty cool:

```
public String checkYourself( String guess ) {
    int cell = Integer.parseInt( guess );
    boolean found = findLocation( cell );
    hits += (found ? 1 : 0);
    return getResultAsString( found );
}
```
Java

We still have to implement the methods `findLocation( )` and `getResultAsString( )` but these don't seem too difficult.

We're doing great. Let's have a look at the pseudocode for the method `setLocationCells`. There are only two tasks.

- The first task is '`int cell =` ⟨`generate first cell number`⟩'. The result has to be an `int` and it has to be a random number in the range `0`–`3`. We've seen this before. It's crying out for a `Random` object.

```
final int maxValue = MAX_CELL_VALUE - CELLS_IN_DOT_COM;
int cell = rand.nextInt( maxValue + 1 );
```
Java

  Here `MAX_CELL_VALUE` is the maximum possible cell number, `CELLS_IN_DOT_COM` is the number of cell numbers in a `dot.com`, and `rand` a `Random` object. We'll implement each of them as class

variables.[1]

```
Java
private final static Random rand = new Random( );
private final static int MAX_CELL_VALUE    = 6; // Maximum possible cell number
private final static int CELLS_IN_DOT_COM = 3; // Number of cells in a dot.com
```

- The last task is '⟨set `locationCells` to {cell, cell+1, cell+2}⟩'. We can implement this using a simple `for` loop.

  ```
  Java
  for (int index = 0; index != CELLS_IN_DOT_COM; index ++) {
      locationCells[ index ] = cell ++;
  }
  ```

  Note that using the enhanced `for` loop makes no sense because we don't need the current values in the array `locationCells` to initialise the array.

Looks like `fillLocationCells( )` is finished:

```
Java
private void setLocationCells( ) {
    final int maxValue = MAX_CELL_VALUE – CELLS_IN_DOT_COM;
    int cell = rand.nextInt( maxValue + 1 );
    for (int index = 0; index != CELLS_IN_DOT_COM; index ++) {
        locationCells[ index ] = cell ++;
    }
}
```

We still have to implement two methods: `findLocation( )` and `getResultAsString( )`. Let's start with `findLocation( )`.

The instance method `findLocation( )` is given an `int` as its argument. The task of the method is deciding whether this `int` is in the array `locationCells`, which is an instance attribute. We know how to do this: linear search. The following is a possible implementation.

```
Java
private boolean findLocation( int cell ) {
    int index = 0;
    boolean found = false;
    while ((index != locationCells.length) && !found) {
        found = locationCells[ index ++ ] == cell;
    }
    return found;
}
```

There are two more possible implementations, which rely on the fact that we know that the numbers

[1] The decision to use class constants for the constants is 100% justifiable because the constants do not change and there is no need to have a separate constant for each instance of the class. However, the decision to implement the `Rand` attribute into a class attribute is less obvious. On the one hand, there' no harm in sharing the attribute among all instances of the class, so implementing it as a class attribute should be fine. On the other hand, sharing the attribute may affect the sequence of random numbers when we create the `Rand` object with a fixed seed. This may unnecessarily hinder testing.

in `locationCells` are consecutive and start with the smallest possible number. The following is how we may implement it. This method is much cleaner, simpler, and efficient.

```Java
private boolean findLocation( int cell ) {
    return (loationCells[ 0 ] <= cell)
               && (cell <= locationCells[ locationCells.length - 1 ]);
}
```

We could have also written the method as follows.

```Java
private boolean findLocation( int cell ) {
    final int difference = cell - locationCells[ 0 ];
    return (0 <= difference) && (difference < locationCells.length);
}
```

Note that this implementation demonstrates that all the information we need is the first value in the array and the length of the array. In hindsight, using an array is a bit of an overkill.

The last method which we have to implement is `getResultAsString( )`. This is also an instance method. It is given a `boolean` which is `true` if and only if the user's guess is a hit. At this stage, the value of the instance variable `hits` should be correct (except for that bug which we shall ignore for the moment). It looks as if we can implement this method straight away.

```Java
private String getResultAsString( boolean found ) {
    String result;
    if (!found) {
        result = "miss";
    } else if (hits == CELLS_IN_DOT_COM) {
        result = "kill";
    } else {
        result = "hit";
    }
    return result;
}
```

Our class is implemented.

8 Debugging

Our last task is to debug our class and reimplement it as required. For the moment, let's do this "by hand". Let's use a `main` method in the `SimpleDotCom` class and change the method for a number of different scenarios. Clearly, this is not ideal. For the moment it'll have to do.

The following is the first scenario. (Notice that when we're testing we *have* to make sure it's reproducible: this is why we create the `Random` object with a fixed seed.)

```Java
private final Random rand = new Random( 0 );

public static void main( String[] args ) {
    SimpleDotCom dotCom = new SimpleDotCom( );
    System.out.println( dotCom.checkYourself( "0" ) );
    System.out.println( dotCom.checkYourself( "1" ) );
    System.out.println( dotCom.checkYourself( "2" ) );
    System.out.println( dotCom.checkYourself( "3" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
}
```

When we use this we get the following output: `miss` `miss` `hit` `hit` `kill`. This looks promising. Unfortunately, things are not always going that well.

```Java
public static void main( String[] args ) {
    SimpleDotCom dotCom = new SimpleDotCom( );
    System.out.println( dotCom.checkYourself( "3" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
    System.out.println( dotCom.checkYourself( "4" ) );
}
```

When we use this we get the following output: `hit` `hit` `kill`. It looks as if we've found a bug.

9 For Friday

Study the notes and study Chapter 5.